Dynamic and Execution Views to Improve Validation, Testing, and Optimization of Autonomous Driving Software

Miguel Alcon[1,2], Hamid Tabani[1], Jaume Abella[1], Francisco J. Cazorla[1]

[1]Barcelona Supercomputing Center

[2]Universitat Politecnica de Catalunya

Abstract

The adoption of autonomous driving (AD) software executed on high-performance Multi-Processor Systems on Chip (MPSoCs) contributes to increasing overall system's safety and efficiency. However, existing AD software frameworks are provided as complete implementations that do not follow any domain-specific safety-requirement centric development process. In this paper, we develop, for the first time, ISO 26262 *dynamic views* of a representative AD framework, *Apollo*. Dynamic views are a key element of software architectural design that links safety software requirements with their implementation, and are the basis to verify that all casuistics are properly considered in the design and tested in the validation tests. We also show that dynamic views miss key information of the execution parallelism of Apollo, needed to assess and improve execution efficiency to meet performance-related safety requirements and reduce resource utilization. We cover this gap by proposing *execution views* that capture the parallelism exploited by the analyzed application on the target MPSoC. Execution views, improve greatly resource usage testing, which is required by ISO 26262, and allow better resource utilization contributing to the stringent cost-reduction requirements in automotive domains.

I. Introduction

Achieving high accuracy and performance while containing maintainability costs has driven the design of navigation frameworks for decades [1]. Navigation frameworks, typically based on machine learning, have recently been tailored for several domains, including autonomous driving (AD). In each of these domains, different requirements apply which, however, may clash with

This is a post-peer-review, pre-copyedit version of an article published in *Software Quality Journal.* The final authenticated version is available online at: http://dx.doi.org/10.1007/s11219-022-09609-x

the principles driving the development of navigation frameworks. In the case of automotive (road vehicles) these include:

- Tight margins impose drastic limits on the costs that can be incurred. On the computing side, this calls for increasing hardware resource efficiency by consolidating the navigation software in the most efficient way so that other applications can be consolidated onto the same hardware platform, reducing hardware procurement. This is key for the competitive edge as the cost of electronics in cars is projected to reach 45% by 2030 [2].
- Another key requirement for automotive relates to safety by fulfilling certain validation and verification (V&V) requirements and providing evidence of the adherence of the development process to the safety guidelines in relevant functional safety standards such as ISO 26262 [3] and ISO/PAS 21448 (a.k.a. SOTIF) [4].

While some initial efforts have been made in the area of V&V of AD frameworks [5], they are however designed without properly factoring safety [6] and cost aspects. An architectural design is needed to describe dynamic design aspects of the software components. Those include their functionality and behavior, control flow and concurrency of processes, data flow between software components and at external interfaces, and temporal constraints (ISO 26262 [3] see part 6, clause 7.4.5). This information is captured in specific diagrams referred to as *dynamic views* that are the basis for multiple steps of the V&V process of safety-related automotive software, and respond to a explicit requirement of the software development process in ISO 26262 (see part 6, clause 7.4.5). This includes identifying application processes and end-to-end paths that need to execute within specific timing bounds, and relevant scenarios for testing (both functional and timing). However, to the best of our knowledge, dynamic views have not been produced for AD frameworks, creating a gap for their certification and leading to potentially inefficient application deployments. Besides, the adoption of MPSoCs in critical embedded systems requires understating how underlying cores are used, which is not provided by any current view of the software architectural design[1].

In this work, we study and experiment with the Apollo [7] AD software system as an example of AD framework. Apollo is an industrial-quality open-source AD software framework with over 120+ industrial partners, most of them top-tier AI companies and car manufacturers. Apollo

[1]MPSoCs have been used in mainstream domains for years, but they are not fully adopted yet in embedded critical domains due to some of the difficulties they bring to the whole V&V process.

has indeed been installed on a variety of prototype vehicles (including self-driving trucks) and supports cutting-edge hardware such as Velodyne and other suppliers' latest LiDARs and cameras, as well as GPU acceleration.

Overall, the long-term viability of software AD frameworks in automotive, and all the potential benefits they can bring in increased safety and efficiency, requires reconciling their design principles and the requirements intrinsic to the automotive market. This includes the analysis of the architecture of the software of AD frameworks from an automotive safety view, and the maximization of hardware resource efficiency. In this line, our contributions are:

1) We generate, for the first time, the dynamic views of Apollo to comply with ISO 26262. The main challenge faced is the gap between dynamic views, originally designed to capture simple and hierarchical software, and Apollo's (and AD frameworks in general) complex structure with non-obvious control flow including plenty of event- and time-triggered callback functions, its advanced object oriented programming, and its hundreds of classes and files with limited documentation in many cases.
2) We show that dynamic views help in understanding the software architecture (e.g. hierarchy of modules and the processes and classes defined in each module). Yet they omit information on the execution parallelism and MPSoC usage, which are key to understand software execution on MPSoCs.
3) To cover this gap, we propose *execution views* that report on the parallelism exploited by the analyzed application, including detailed information on which cores/threads processes run throughout their execution. We generate Apollo's execution views revealing how processes are mapped to cores expansively, keeping all cores used periodically, but largely underutilized. Such consolidation is against efficiency challenging the deployment of other applications or setting cores to low-power states since all cores are busy.
4) The information from execution views can be leveraged to improve resource usage testing, e.g. the use of computing resources (CPUs), i.e. how the operating system underneath Apollo manages processes. We also propose and enforce an enhanced consolidation (process-to-core mapping) of Apollo processes into 4 out of the 8 cores available in the target MPSoC that allows using idle cores to schedule other applications or to apply aggressive power saving techniques (e.g. power gating [8]). Our results show that, under our enhanced consolidation, (a) Apollo preserves its high average performance; and (b) it is less affected by the interference generated by other applications that we run simultaneously.

This makes contention experienced by Apollo easy to bound since we remove several fine-grain interference channels in the cores and L1 caches.

Overall, this paper shows the dynamic views of Apollo AD software (Section III) and also contributes with a new tool to validate and optimize automotive software deployment: the execution views (see Section IV) that are used as part of the design and verification process of automotive software running on complex multicore (MPSoC) processors to complement *components*, *dynamic*, *interface*, *operating modes*, and *error behavior views*.

The rest of the paper is organized as follows. Section II introduces the main modules of Apollo and the software architecture design phase of ISO 26262 in which dynamic views are generated. Section III presents the dynamic views we generated for the Apollo modules. Execution views are introduced in Section IV along with those derived for Apollo. Section V shows our proposed Apollo enhanced consolidation and evaluates it. Related works are described in Section VI. Finally, Section VII summarizes the main conclusions of this paper.

II. Background

A. Software Architectural Design in ISO 26262

The software development process in the context of the reference functional safety standard for the automotive domain, ISO 26262 [3], starts when software safety requirements are derived from the full system design (see top-left corner of Figure 1). However, existing AD frameworks do not follow this safety-requirement centric design, implementation, verification, and validation approach. In fact AD frameworks are provided as complete implementations that carry no architectural design at all, hence creating a gap between software requirements and implementation.

The software architecture, which links safety software requirements with software implementation, is the most important document for the software part of the system, especially in safety related projects. The software architecture provides key information of the software needed for its actual implementation, verification and validation. In particular, IEC 61508 [9] – the parent safety standard of ISO 26262 – states the following in Part 3, section 7.4.3: *The software architecture defines the major elements and subsystems of the software, how they are interconnected, and how the required attributes, particularly safety integrity, will be achieved. It also defines the overall behavior of the software, and how software elements interface and interact. Examples of*

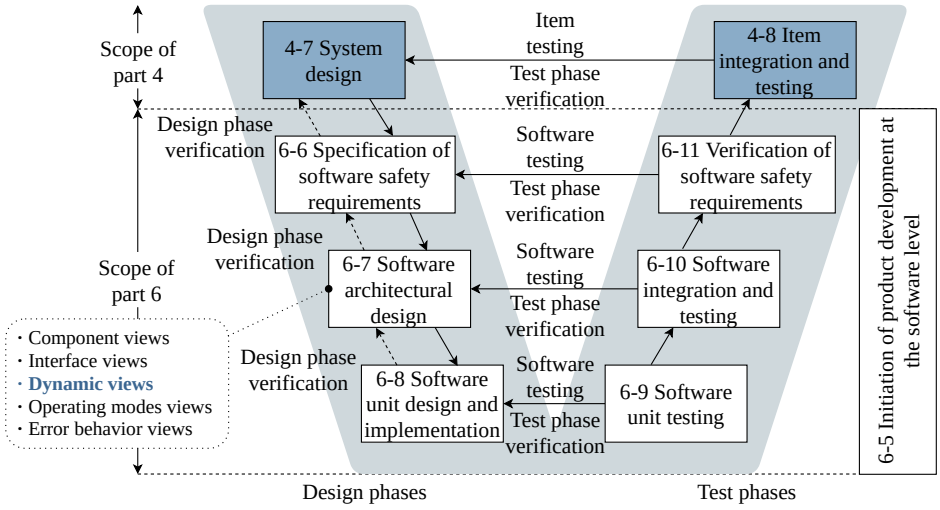

Fig. 1: Phase model for the software development as part of ISO 26262 [3].

major software elements include operating systems, databases, EUC[2] *input/output subsystems, communication subsystems, application program(s), programming and diagnostic tools, etc.*

In the specific case of ISO 26262, software architectural design, which must favor modularity, encapsulation and simplicity as stated in part 6 clause 7.4.3, includes the description of static and dynamic design aspects, as detailed in ISO 26262 part 6 clause 7.4.5. In particular, such clause states the following: *The software architectural design shall describe: (a) the static design aspects of the software components. Static design aspects address: the software structure including its hierarchical levels; the logical sequence of data processing; the data types and their characteristics; the external interfaces of the software components; the external interfaces of the software; and the constraints including the scope of the architecture and external dependencies. (b) The dynamic design aspects of the software components. Dynamic design aspects address: the functionality and behavior; the control flow and concurrency of processes; the data flow between the software components; the data flow at external interfaces; and the temporal constraints.*

Those descriptions are provided in the form of software views, which include the following:

- Static aspects: Components views consisting in block diagrams depicting the different components.
- Dynamic aspects: Dynamic views describing how components interact during operation

[2]Equipment Under Control.

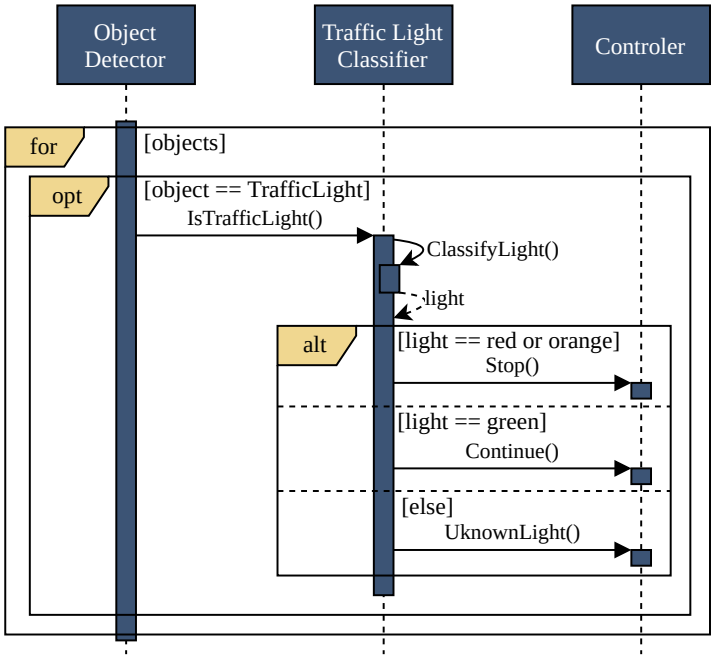

Fig. 2: Dynamic view example with a naive process to handle traffic lights.

by means of specific diagrams that we detail later; operating modes views depicting the different states for the software and the existing transitions and conditions across states; and error behavior views describing how components interact upon the occurrence of an error until it is properly managed (e.g., reaching a safe state).

Overall, static aspects are restricted to software structure, logical sequence of data processing, data types, and interface information. Dynamic aspects, instead, provide richer information, and include information such as control flow, data flow, and concurrency of the processes. Among the different views used for dynamic aspects, dynamic views allow identifying individual processes, their communication mechanisms, and how data and control is exchanged over time. Dynamic views represent time in the vertical dimension, and objects (e.g. processes) in the horizontal one (see Figure 2). They are expressive allowing to represent *Functions* (solid arrows), *optional* (opt) fragments for conditional (e.g. if-then-else) execution, as well as different types of loops indicated with backward arrows in the vertical dimension.

Dynamic views are instrumental in multiple steps of the software development process to reach an efficient and safety compliant system, including:

- End-to-end paths from sensors to actuators can be identified in the dynamic views, even if

they span across multiple processes with any degree of concurrency. This provides critical information to obtain end-to-end execution time bounds (e.g. building on WCET estimates of individual runnables) that can be compared against response time bounds (e.g. the brake must respond within 100 ms since the braking pedal is activated).

- Dynamic views are the basis for the generation of test scenarios by exposing all relevant interactions across processes, which must be triggered by the validation tests. Note that test scenario generation spans across runnables and tasks, and aims at guaranteeing that all operation modes of the software are triggered.
- Dynamic views also provide some limited information about application's processes and their dependencies in the automotive certification process.

Failing to describe all relevant aspects of the software by means of dynamic views can easily lead to design errors due to the inability of the software to manage specific casuistic which otherwise could have been detected by means of dynamic views. It can also lead to the inability of the validation tests to trigger design flaws because the lack of some views challenges the systematic generation of tests for all relevant operation scenarios. Moreover, the increasing complexity of software and computing platforms beneath, which enable increasing concurrency by, for instance, providing larger core counts, call for additional views for the verification and testing of emerging aspects related to execution concurrency. Hence, execution views like the ones proposed in this work become increasingly instrumental along with the dynamic views for the V&V of safety-relevant complex software frameworks, and less instrumental but still useful, to improve performance by detecting performance inefficiencies.

Despite the benefits of dynamic and execution views, AD frameworks in general, and Apollo in particular, do not provide them, therefore hampering achieving ISO 26262 compliance.

B. Apollo AD Framework

Apollo is structured as a collection of *processes* that run on a regular basis. The execution is divided into stages, each of which corresponds to a different functional step in each module. In Figure 3 we present the main software architecture and *modules* in Apollo. Each module is in charge of a specific functionality of the AD framework, from perceiving the surrounding of the vehicle to steering or braking. Next, we describe the main functionality of each module, starting with *HD Map*, which functions as a library and operates as a query engine support that offers ad-hoc organized information about the roads to *Planning, Prediction*, and *Perception*. *Perception*

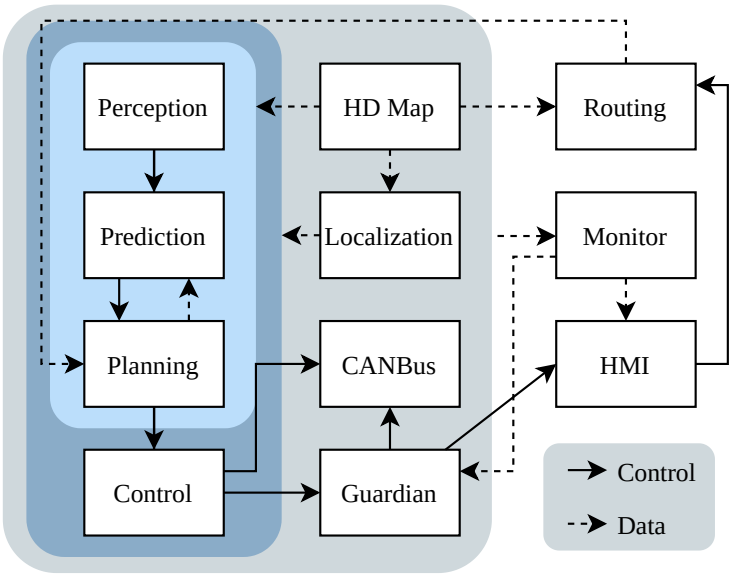

Fig. 3: Apollo's interface view.

detects obstacles (e.g., pedestrians, vehicles, or barriers), traffic signs, and drivable regions in the vicinity of the autonomous vehicle. To increase accuracy, it combines the output of many types of sensors, including LiDAR, radar, and cameras. It is the most important and complicated module in an AD system. On the other hand, *Localization* uses various information sources, such as GPS, LiDAR, and IMU, to estimate where the autonomous vehicle is located. With the obstacles detected by *Perception* and the results of *Localization*, *Prediction* anticipates the future motion trajectories of these obstacles. The predicted trajectories, along with the vehicle's route calculated by *Routing*, are used by *Planning* to create a safe and comfortable spatio-temporal trajectory for the autonomous vehicle to take. Then, *Control* generates control commands such as acceleration, braking, and steering to carry out the scheduled spatio-temporal trajectory. Finally, *CanBus* actions these commands. In particular, this module is the control interface that sends commands to the vehicle's hardware, and also communicates chassis data to the software system.

Modules and processes are independent pieces of code in Apollo. Apollo v3.0 (the target of this work) relies on the Robot Operating System (ROS) [10] to build the communication system between modules and sensors. None of the modules calls a ROS routine directly and they use a specific class for this purpose: *adapters*. There is one adapter per each kind of message (also referred to as *topic*) which takes care of the subscriptions, publications, and execution of the routines linked to the topic. For this reason, all the event- and time-triggered functions start

within an Adapter.

Modules comprise one or several *nodes*, all with the same structure, but each one in charge of a single functionality of the module. Apollo offers different nodes that can be used depending on the target hardware. The selection of the utilized nodes and the communication between them is defined in input configuration files.

One of the key components of the AD framework is the perception module, given (i) its complexity, which is necessary to deal with inputs from various components (e.g., video cameras, short- and long-range radars and LiDARs), and (ii) its long execution time that represents a large fraction of the overall execution time of the AD framework. For this reason, we focus mainly on this module with detailed analysis. Note that camera-based object detection is part of the perception module, which is fused with the LiDAR-based object detection and Radar processes. For our experiments, as depicted in Figure 5, we use (Apollo) configurations comprising the following perception nodes.

- *Camera* processes the images from the camera sensor to detect obstacles and lanes.
- *Radar* processes the signals of the radar sensor to detect the surrounding obstacles.
- *Fusion* fuses the outputs from the camera and the radar nodes to generate consistent information of the perceived obstacles with both sensors' data.
- *Lane post-processing* processes the data obtained by the camera node regarding the lane lines.
- *Motion* gathers and processes information from the camera sensor and the Localization module, which is then used by the *Lane post-processing* node.

III. Apollo Dynamic Views

Generating dynamic views for Apollo is a challenging task due to the complexity and limited information:

- Complexity. Apollo is a large software framework including 373 C++ and 240 header (.h) files spread across 121 folders. The classes of objects implemented are tightly coupled hindering the identification of process interactions.
- Limited documentation. The lack of documentation for some modules, which provide no comments in the code for many functions, and the use of non-obvious names for functions and variables further challenges the comprehension of the existing processes and their interactions to generate the dynamic views.

When producing the dynamic views, one of the main decision points is to select the right level of detail. Insufficient detail limits the benefits of developing dynamic views. Excessive detail makes dynamic views too complex and hence, hard to understand and to use effectively. In fact, the purpose of dynamic views is to enable engineers to identify end-to-end paths, generate test scenarios, and optimize the use of resources, hence making readability a must. As a result, the particular functions, variables, and control flow constructs that need to be depicted require careful analysis to show only those relevant for process interaction.

We address this challenge by building the dynamic views through an iterative process combining code inspection and execution tracing, which provides also some form of redundancy in the process, and hence self-validation. We proceed hierarchically, in line with automotive software architectural design, going from general dynamic views showing coarse-grain components, to dynamic views of each specific component. We start from an Apollo general dynamic view, move to module-level dynamic views, and continue with node (process) dynamic views where classes, functions, and calls are detailed.

In this section we show representative dynamic views at the module level rather than exhaustively covering all modules. In particular, after showing the dynamic view of the overall Apollo, we focus on the perception module as one of most complex modules in AD frameworks.

A. View of the Overall Apollo

Apollo comprises several modules with predefined interactions among them. Figure 4 shows the general dynamic view we developed for Apollo focused on those modules with more complexity and, consequently, with longer execution times, namely localization, perception, planning, control and prediction. The others are not included to ease readability. In Figure 4 green rounded boxes represent Adapter processes (related to communication/ROS). Blue square boxes are Apollo processes. Black arrows are adapter functions that can trigger modules' functions, while green arrows are module publish functions, namely functions limited to the generation of new messages of their specific topic.

Adapters perform event- and time-triggered callbacks to each module, including event communication from modules to adapters that, in turn, perform callbacks to other modules. For instance, as shown in Figure 4, the "sensor/radar" adapter (3rd column) can trigger callbacks for the perception module (5th column). We can classify adapters by the source of the topic they handle i) mainly from sensors (external input); and ii) Apollo's modules (internal input). Besides

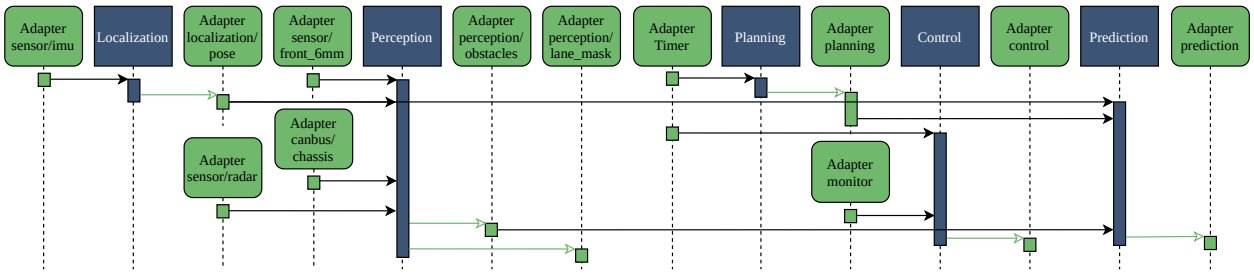

Fig. 4: Developed dynamic view of Apollo and its modules.

these two types of adapters, there is a special one, the "Timer" adapter (8th column), that takes care of triggering functions that must be periodically executed during the whole execution of the Apollo framework. The user can determine the triggering period of each function. Figure 4 shows all three types of adapters. For example, the adapter in charge of "localization/pose" (3rd column) handles the messages from Localization (2nd column), and "sensor/front_6mm" (4th column) the ones from the front camera sensor (sensors are not shown in the figure).

Each module can be triggered by one or more adapters. In particular, each adapter triggers a specific function of the module. Once the module finishes processing messages, it publishes one or more messages with the result of this processing, which will be received and handled by the corresponding adapters.

Let's now review the complete flow starting with the "sensor/imu" adapter (1st column) triggering Localization (2nd column) upon the reception of the corresponding input from sensors by the former. Once the module finishes its work, it generates and publishes the corresponding message to the "localization/pose" adapter (3rd column). Then, this adapter triggers the corresponding function of Perception (5th column). Once Perception has received and processed all messages, it generates and publishes two messages for "perception/obstacles" (6th column) and "perception/lane_mask" (7th column) adapters. After that, the "perception/obstacles" adapter handles the new message, triggering the Prediction's function (13th column), and so on. In other words, as explained in Section II-B, Perception detects the obstacles that surround the vehicle with the data from the camera and radar sensors, from the chassis, and from the localization of the vehicle, which is calculated by Localization. These obstacles, along with data from Planning and Localization, are then used by Prediction to anticipate the future motion trajectories of each of the obstacles.

Following the explanations given in Section II-B, the Planning module (9th column) needs

the predicted trajectories to plan the new vehicle's trajectory, i.e., the "prediction" adapter (14th column) sends this data to Planning, but it is not reflected in the figure. The reason behind is that the "prediction" adapter does not trigger the Planning's callback because it is triggered periodically. In the same line, adapters such as "prediction" (14th column) or "perception/lane_mask" (7th column) do not trigger any module, but the data they carry is used in other processes. These data dependencies are added in the module-specific dynamic view so that this information is properly reflected without overloading the overall Apollo's dynamic view for readability.

B. Views of the Nodes of the Perception Module

We analyze perception's code to identify the most relevant functions and obtain their structure (functions invoked, inputs and outputs, etc.). We create one dynamic view for the module and one per node with the information obtained from this analysis. In this work, we focus on the views of the most complex node, the camera node.

1) Perception Module: The dynamic view of the whole perception module, Figure 5, depicts perception's processes (a.k.a. nodes). Perception consists of 8 processes that run independently. Some of them correspond to callback functions that are called upon the occurrence of specific events (i.e. event-triggered processes), whereas the remaining processes run repeatedly in an infinite loop and process the most recent input data they find. In particular:

- The set of event-triggered processes correspond to *ImgCallback-Camera*, *ImageCallback-Motion*, *OnChassis-Fusion*, *OnLocalization-Motion*, *OnLocalization-Radar* and *OnRadar-Radar*.
- The remaining processes, namely *ProcEvents-Fusion* and *ProcEvents-Lane*, run in an infinite loop in their corresponding nodes.

Besides those processes, we can also find *Adapter* processes that just call event-triggered processes and hence are not further analyzed in the discussion in the rest of the paper.

The processes that run just once for initialization purposes have been omitted in the dynamic view for the sake of clarity. Also note that perception's dynamic view has been tailored to its particular instantiation with the input data available for Apollo. In particular, we chose the input data set (referred to as *bag* in ROS'/Apollo's terminology) that includes data for the camera and the radar, but not for the LiDAR. Alternative bags exist with data for other combinations of

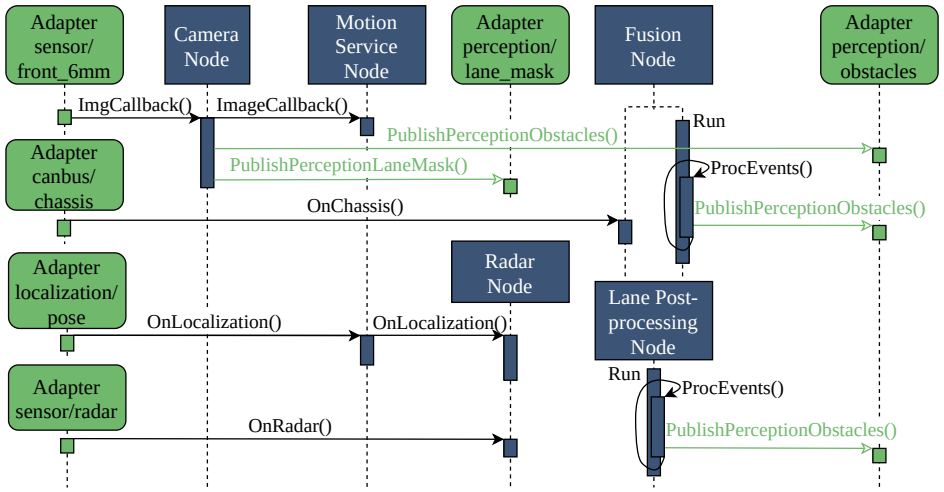

Fig. 5: Dynamic view of the full Perception module.

sensors, but none of them includes data for all sensors simultaneously, namely camera, LiDAR and radar.

This module-level view is particularly useful to understand the different processes that form the perception module. Note, however, that such root view does not provide information on the internals of each process (node). Also note that, while not shown in this work, we generate analogous views for the remaining and simpler Apollo modules, which only have between 1 and 3 event-triggered processes each.

2) Camera Node: The Camera node is mainly formed by an event-triggered process, *ImgCallback*. Following the dynamic view depicted in Figure 6, the execution of *ImgCallback* starts when a message of the topic "sensor/front_6mm" (front camera) enters the system. More precisely, when the Adapter in charge of the topic triggers the function. *ImgCallback* is a large function that relies on many other C++ classes (such as YOLO camera detector [11], Caffe [12] Net, Geometry Camera Converter, Tracker, and Filter) to perform specific work. For instance, YOLO camera detector detects relevant information of the car's surrounding taking the images that come from the camera(s) as an input. This class is first used to detect the obstacles (objects) and the lane lines (mask) that can be observed on the image through its *Multitask* function.

Then, if in the configuration file the user enables the "use_whole_ lane_line" option, YOLO Camera Detector would trigger *Lanetask*, which processes the image again, however, focusing only on the lane lines in order to gather more information about them (masks). Both functions, *Multitask* and *Lanetask*, manage their neural networks through Caffe [12], making calls to

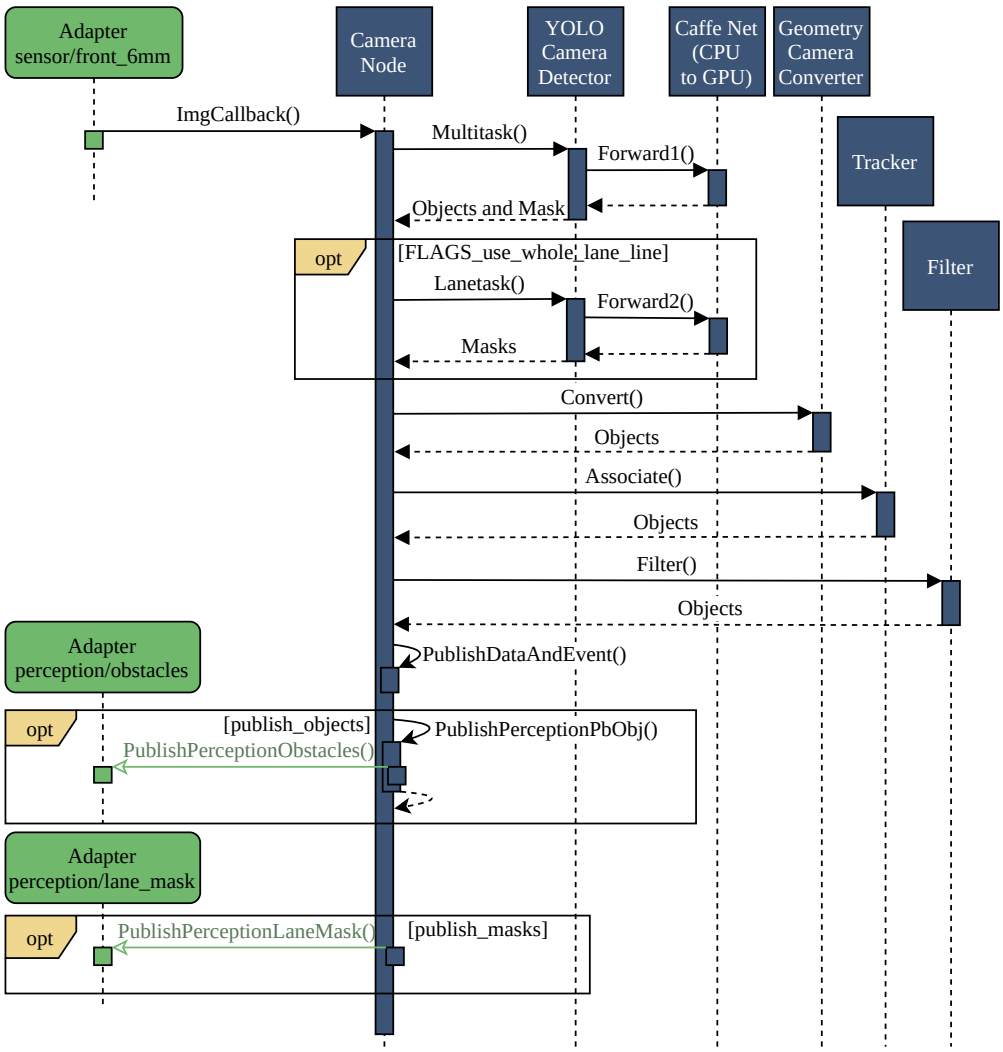

Fig. 6: Perception module's camera node dynamic view.

Forward1 and *Forward2* respectively. Both functions are executed on the GPU, however, it is possible to configure perception to run them on the CPU.

Once the necessary data from the image has been taken, perception calls *Convert*, *Associate* and *Filter* to preprocess the data. *Convert* converts the bounding boxes of the detected obstacles from 2D to 3D; *Associate* relates the detected obstacles with other ones that were previously detected; and *Filter* checks whether each of the obstacles was previously associated.

Finally, when *ImgCallback* detects an object and publications are enabled, it calls the adapters of "perception/obstacle" and "perception/lane_mask" topics to publish the results (objects and masks).

It is worth mentioning that the output of the camera and radar nodes is consumed by the fusion node. This is not reflected with explicit dependencies in the dynamic views since data communication occurs asynchronously. In particular, camera and radar related processes are event-triggered and execute every time new data from their corresponding sensor is available. Instead, the process consuming their outputs in the fusion node runs in an infinite loop, and hence, executes periodically using the last set of results available from the camera and radar nodes, therefore without any explicit synchronization.

3) Summary: Overall, the dynamic views we have generated for Apollo provide structured information and insights on elements like the hierarchy of modules, the processes (nodes) existing in each module, and the classes and functions called for each node. Those views, provide key control flow information and relationships among nodes, classes and functions as required in ISO 26262's software development process.

C. Dynamic Views Automation and Generalization

Due to the complexity of AD frameworks, having some form of automation to generate dynamic views is key to make the endeavour attainable. Moreover, automation also facilitates generalization by allowing generating those views for other AD frameworks using the same approach.

1) Automation: Dynamic views are typically generated building on views like interfaces views, which define what components interact with each other component, and the operating modes views, which provide additional information about the interactions of components in each operating mode. For AD frameworks, since the frameworks already exist (i.e. are designed and implemented before generating these views), these views can be generated following a different approach. In particular, the framework can be instrumented with appropriate debugging tools that allow tracing calls and returns across functions (i.e. the call tree).

For the subset of casuistics triggered by the tests, this process provides a first approach to generate dynamic views that can be complemented using static analysis tools processing the code or less automated approaches like code inspection. In fact, one could generate tests triggering untested casuistics based on expected dynamic interactions from the interface and operating modes views to crosscheck that, effectively, those interactions occur as expected.

General purpose tools like Valgrind [13] or *Perf* (part of Linux) could be easily used to that end. This is also the case for commercial timing analysis tools like RapiTime [14]). Overall,

automating the generation of the initial call tree on which to build to produce dynamic views of Apollo is a feasible challenge, hence avoiding the need for manual code instrumentation. As we stated at the beginning of Section III, in our particular case, we used an iterative process combining code inspection (manual) and execution tracing (automated) to build dynamic views. We could not rely only on automatic processes from existing tools because they could not handle Apollo's complexity.

Note that, given that AD frameworks inherit safety requirements, the tools used to generate traces to produce dynamic views must be qualified unless a strict verification process is performed later on to confirm that the information retrieved from the tools is correct, and complement it. In that case, the verification process, typically carried out by independent engineers for the most stringent integrity levels, backs up the automated process, which then can be performed with non-qualified tools. However, for efficiency reasons, we recommend using appropriate qualified tools, which can build on top of, for instance, those assessing function call coverage, which already need being qualified.

2) Generalization: Using software tools to produce the call-return traces of dynamic views eases the portability of the approach across different AD frameworks reducing the manual intervention. Yet the generation of dynamic views involves some manual steps in which they are generated from less complex ones (e.g., components views) to more complex ones (e.g., dynamic views), and from the highest abstraction levels (e.g., full tasks) to the lowest ones (e.g., individual functions) hierarchically to mitigate potential errors in their generation.

IV. Execution Views

AD frameworks are executed on MPSoC-based computing platform to achieve the required performance, and Apollo is not an exception to that. MPSoCs comprise several cores and each core can be multi-threaded, i.e. having several *hardware threads*. In order to understand how Apollo uses the hardware platform, it is instrumental to understand which software threads (i.e. processes or functions) run in parallel.

Dynamic views, however, are hardware platform agnostic and omit important information related to execution parallelism and process mapping to computing resources. More in detail, dynamic views provide some insight on which processes (or functions thereof) could theoretically run in parallel. However, whether those processes can effectively run in parallel in the specific MPSoC in which the AD framework is deployed, or what computing resources should each

process use is not part of the dynamic views. For instance, dynamic views do not provide information on how often each process is executed, for how long, and whether some processes are triggered simultaneously or not. This information, which is platform dependent, and hence, it is not included in dynamic views, requires other types of views to be exposed and enable optimizing how efficiently computing resources are used.

We contend that dynamic views can be augmented with *execution views* to provide specific information on the parallelism exploited by the analyzed application. This can be achieved with visual performance analysis tools oriented to ease the profiling of the execution of applications. The specific features to be provided by the tool are as follows:

- Provide information on the specific computing resource (e.g. hardware thread of a core) in which each instance of each node is run.
- Provide sufficient granularity to enable optimization of hardware resource efficiency (e.g. start and end execution time for each job and/or function).
- Cause negligible probe effects due to instrumentation and data collection by adding light instrumentation at a sufficiently coarse grain (e.g. at node and/or function granularity).
- Ease instrumentation and data collection of the AD framework software implementation.

In this work we use the Paraver performance visualization and analysis tool [15] to meet all these requirements. Paraver provides a visual representation of the program's execution that we use to develop its execution views.

A. Execution Views of Apollo

Modules and their nodes are run as independent processes with some nodes being invoked from event-triggered callback functions by ROS (i.e. those called through an *Adapter* wrapper). This is the case of the camera node of the perception module (see Figure 6). Internally, that node may use multiple classes like *Caffe Net*, *Geometry Camera Converter*, *Tracker* and *Filter* as shown in the figure. Other functions, like *ProcEvents* in the fusion node in perception (see Figure 5), run repeatedly in an infinite loop processing their latest input data available. Hence, some processes may need to run in parallel though dynamic views are inconclusive on this.

- All event-triggered processes in each module, such as the camera node of perception, are managed explicitly by ROS and hence, the particular hardware thread where they run is not directly controlled by the end user. Instead, the user can only provide ROS with a pool of hardware threads in which callback processes of a given module are scheduled by

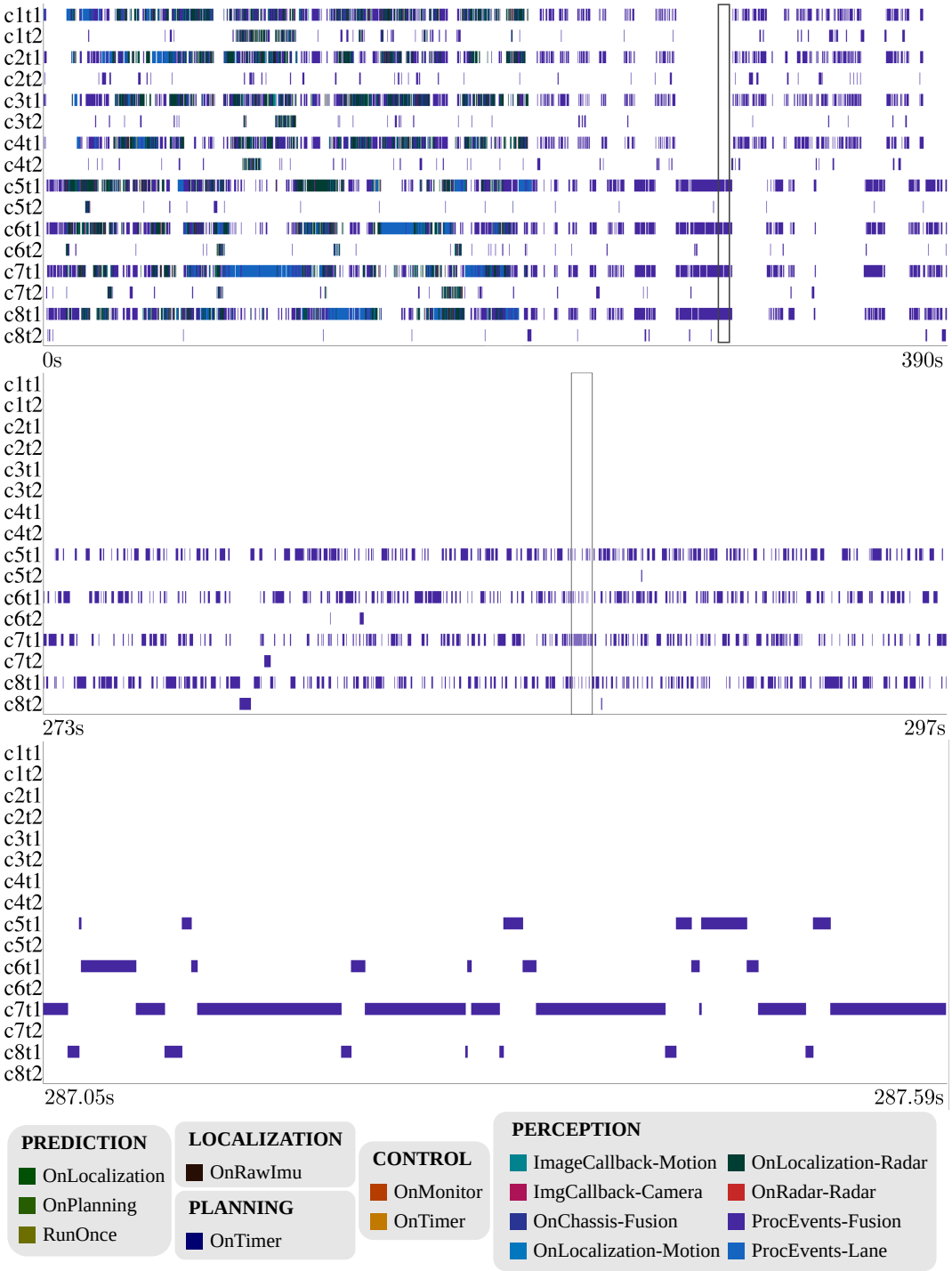

Fig. 7: Full execution profile of the perception module. The second view is the zoom-in of the first. Y-axis define in which core (c) and thread (t) is running each process. On the legend, notice that only “ProcEvents” functions are not event-triggered (callback) functions.

ROS. For instance, we can provide ROS with a set of hardware threads for perception's event-triggered processes in which ROS freely schedules those processes. While we can provide ROS with a disjoint set of hardware threads for event-triggered processes of the different modules, exercising some control on the hardware threads in which processes run, we cannot exercise control on a per-process basis for the set of event-triggered processes of a given module.

- End users can bind non-event-triggered process (e.g. the *ProcEvents* process in the fusion node from the perception module) to a particular hardware thread superseding Apollo.

In the execution view of perception, Figure 7 (top plot), we show the 16 hardware threads of the platform (8 cores with dual-thread multi-threading each, further details in Section V-A) in the y-axis, and time in the x-axis. Each pair of hardware threads in the y-axis (e.g. starting from top to bottom) corresponds to the 2 hardware threads for a given core. In the top plot, we observe a gray rectangle spanning across all hardware threads in the center-right of the figure. Part of such rectangle is zoomed in the center plot.

We see that all hardware threads execute at least an instance of some of the processes of the perception module. Hardware thread 1 for each core is more loaded than hardware thread 2 (i.e. from top to bottom, odd rows are busier than even ones). Also, while the top figure suggests that all processes run in parallel, the zoomed-in plot shows more clearly that threads do not necessarily overlap their execution. In this particular time window, both hardware threads for the top 4 cores (cores 1, 2, 3 and 4) are idle, as well as hardware thread 2 for cores 5, 6, 7, and 8. Also hardware thread 1 for cores 5, 6, 7, and 8 execute processes in a non-concurrent manner. Those hardware threads execute different instances of the same process (*ProcEvents* of the fusion node) during the time window, which are called one after the other in an infinite loop. Each instance runs in different hardware threads because Apollo does not control where its processes are executed. Since they are individual processes, they can run on any hardware thread as dictated by the OS.

When analyzing the semantics of Apollo, we realize that, as explained before, *ProcEvents* process in the fusion node is expected to run uninterruptedly, but requiring one hardware thread at a time. However, this information is not clearly exposed by dynamic views, and only when using execution views, as proposed in this paper, this situation becomes evident.

When analyzing other processes, such as those called through callback functions, we can conclude, based on dynamic views, that they are run upon the occurrence of specific events.

However, whether those events may lead to long idle periods between invokations, or to concurrent executions of multiple instances of the same callback function, cannot be determined at all based on dynamic views. Instead, one must resort to execution views to observe the frequency and duration of those processes, and whether behavior changes over time. Such information allows understanding how much computation resources are needed to run those processes, and hence, what the best way is to allocate hardware threads to ROS so that callback processes run without relevant mutual delays, but using hardware resources efficiently. In this work, we analyze the behavior of the callback functions by collecting sufficiently long execution views reflecting the behavior of each process over time so that hardware threads can be allocated as efficiently as possible. By releasing multiple hardware threads and cores, they can be disposed to run other applications or for power savings.

As explained before, the complexity and execution time requirements of the perception module is higher than other Apollo modules, which we also analyzed reaching analogous conclusions about process to hardware thread mapping. Therefore, we do not further review other modules since they do not provide further insight.

B. Integrating Execution Views into ISO 26262's Development Process

Execution views provide unique information (i.e. not available in other views) about task scheduling that is key for timing verification and validation. In particular, execution views provide information needed in the following steps of the software development process:

- **Verification**: ISO 26262 part 6 clauses 7.4.11 and 7.4.12, and Annex D, focus on the existence of *freedom from interference* or *sufficient independence* between software components, which includes functional aspects like those related to memory corruption and also timing aspects related to multicore timing interference. Verifying those requirements carries as a first step determining the software components that can execute simultaneously in different processor cores (or other computing units) with some hardware shared resources that could lead to mutual timing interference. However, dynamic views just provide information about how software components and their tasks are serialized due to functional dependencies, but lack information on whether tasks are effectively executed concurrently and, if so, whether they do it in computing components (e.g., cores) sharing some hardware resources. That information is provided by execution views that allow identifying when and where each task runs, and hence, whether two tasks can create timing interference on each other.

- **Validation**: ISO 26262 part 6 clauses 9.4.3 introduces *resource usage tests* as a means to validate that resource budgets allocated are not exceeded. The aspects to be tested include execution time (e.g., validating that deadlines are not missed). Resource usage tests related to timing build on developing relevant test cases, which in the context of multicores requires understanding the tasks that can run concurrently in different computing cores to develop appropriate tests that trigger feasible but stressful scenarios. Execution views reveal such information and hence, become a fundamental tool to develop appropriate tests to prevent any misbehavior from passing the validation phase unnoticed.

Interestingly, many of these effects are not relevant for single-core processors, so it is understandable why existing views do not account for them. However, the adoption of multicores as the reference computing platform in the automotive domain, and the trend towards increased hardware resource sharing, bring new challenges for the V&V of automotive software that make execution views a key element of the ISO 26262 development process for software.

C. Execution Views Automation and Generalization

1) Automation: Execution view automation build on the use of tools that enable instrumenting software to collect detailed execution information about when each task starts and ends each one of its jobs, and the computing element (i.e. the core/thread) where it is run. Instrumentation tools, moreover, must be as little intrusive as possible to mitigate any probe effect hence making the information obtained representative of the behavior of the system under analysis.

Another aspect to consider of the instrumentation tools is qualification, which depends on the integrity level of the software under analysis. Qualified tools for timing analysis like RapiTask [16] could be conveniently extended to provide execution views since they already provide instrumentation capabilities. This just requires generating information like the start and end times of each job, identification of the task to which each job belongs, and specific computing component where each job is run. Other tools from other domains already provide such information when configured appropriately. This is the case of Paraver [15], though it is not qualified. As illustrated before in Figure 7, Paraver gathers the information needed and also provides visualization capabilities. Overall, we conclude that there is not stumbling block for the generation of execution views with affordable changes to existing qualified tools.

2) Generalization: The tools used to generate execution views can be easily configurable to the type of instrumentation needed. Those tools typically produce traces in an internal representation

format, and such traces can be visualized with the tools themselves. Hence, no application-specific information is needed for this process, which can be applied, a priori, on any AD framework or any other type of automotive software.

V. Improving Apollo's Resource Usage and Its Integration with other Applications

The information derived by execution views extending that provided by dynamic views provides insight on how Apollo uses the underlying platform. We leverage this information to optimize Apollo's execution when run in isolation and when it is integrated with other applications on the target platform with the goal of increasing resource efficiency.

A. Experimental Setup and Platform

We run Apollo on an x86 platform using an AMD Ryzen 7 1800X CPU [17] with 8 cores (each one being 2-threaded) and 64 GB of DDR4 RAM operating at 2133 MHz. The processor has three levels of caches, L1, L2, and L3 which are 96 KB, 512 KB, and 16 MB respectively. The L1 and L2 caches are private per each core while the L3 is shared among all the cores. In order to satisfy the computational needs of Apollo our platform is equipped with a Pascal-based high-end GPU (the NVIDIA GeForce 1080 Ti [18] with 3584 CUDA cores and 11 GB of GDDR5 memory). Our setup resembles state-of-the-art automotive Systems on Chip (SoCs) targeting the automotive AD market such as the two variants of the NVIDIA Drive PX2 platform [19], AutoCruise and AutoChauffeur. They both have similar CPU and GPU configurations to ours with a single NVIDIA Tegra X2 SoC [20] containing 4 ARM Cortex-A57 and 2 Denver cores, combined with an integrated 256-core Pascal GPU. The latter contains two Tegra X2 SoCs and 2 discrete Pascal-based GPUs. Moreover, the ARM Cortex-A57 CPUs used in these platforms exhibit similar hardware complexity as that of the x86 cores in our platform since both are superscalar, out-of-order CPUs, with several levels of cache. The GPU in our configuration is discrete similar to latest automotive platforms such as NVIDIA Drive Pegasus [21], thus requiring data transfers. However, we have verified that data transfers account for less than 1% of the total execution time of Apollo. Therefore, the multiprocessing capabilities in the CPU side and the GPU architecture of our platform are representative of the automotive domain.

Due to the software dependencies of Apollo, the framework is executed on a Linux environment and it is built on top of ROS [10], as described earlier. In order to minimize the jitter

stemming from outside of the application, i.e. from the operating system or hardware behavior, we follow standard guidelines for real-time execution under Linux. In particular, we minimize the running services of the system to the bare minimum, stopping services such as mail services or the window manager. In addition, we assign real-time priorities from the Linux kernel to all scheduled tasks under analysis. We further pin tasks on specific cores in order to prevent costly task migration, and map interrupts to fixed cores.

B. Resource Usage

Interestingly, while the coarse-grain execution views (top chart in Figure 7) seem to indicate that there is parallel execution across processes in different hardware threads, a closer look (bottom chart in the same figure) reveals that in many cases processes do not really run in parallel across the different hardware threads. In particular, our analysis reveals that the perception module has two types of processes: event-triggered ones invoked with callbacks and the rest of them. Event-triggered processes (e.g. *ImgCallBack* in Figure 6) perform some work upon invocation and finish their execution whenever the callback finishes. Instead, the remaining processes are called repeatedly in an infinite loop, see *ProcEvents* processes in Figure 5.

1) Perception module: The analysis of the execution profiles of perception (Figure 7) shows that event-triggered processes have relatively low computing requirements and, apparently, they could be scheduled in a single hardware thread. Also, the different execution instances for each of the two *ProcEvents* processes (the one for Fusion and the one for Lane nodes) are scattered across different hardware threads/cores by Apollo. However, the different instances of any given *ProcEvents* process run serially as they are called sequentially in an infinite loop. Therefore, each *ProcEvents* process should be able to run properly on a single hardware thread.

Building on these conclusions, we consolidate all event-triggered processes into a single hardware thread. Note that, since our goal is illustrating the impact on performance and efficiency achieved building on execution views' information, which is achieved with these simple modifications, we do not enter into considerations related to functional isolation and to the degree of achievement of freedom from interference.

As shown in the resulting execution view, Figure 8, we consolidate all event-triggered processes into hardware thread 1 of core 1, (referred to as c1t1 for short) by setting such single hardware thread as the only one that ROS can use for perception's event-triggered processes (callbacks). Also, we allocated each one of the *ProcEvents* processes to a single hardware thread (*ProcEvents*-

Fig. 8: Execution view the enhanced consolidation of the perception module.

Fusion to c2t1 and *ProcEvents-Lane* to c3t1). As it can be seen, c1t1 has low utilization whereas c2t1 and c3t1 are fully utilized.

Note that the goal of such improvement in the scheduling is illustrating that execution views provide key information to detect suboptimal task consolidation, which could otherwise challenge not only average performance and power, but safety-related real-time requirements due to increased execution times produced by inter-core communications of any sort that could otherwise be managed intra-core is tasks are scheduled properly. The alternative consolidation proposed, while showing that it is more efficient in practice in our case study, might be outperformed by others in different metrics. Yet this is irrelevant for our work since our goal is illustrating that the use of dynamic and, especially, execution views is instrumental to detect some forms of application misbehavior in terms of performance.

2) Full Apollo framework: In this section we cover the dynamic views of perception, as shown next. We also provide in the Appendix the dynamic views of other modules, in particular Control, Prediction, and the rest of Perception's nodes.

1) All Apollo modules except perception use exclusively event-triggered processes. This is analogous to the camera node of perception. Hence, only the perception module has two *ProcEvents* processes.
2) The default consolidation scenario (*DefaultCS*) for the whole Apollo causes no impact on perception's *ProcEvents* processes, which behave analogously to the case where perception is run as a standalone module. In other words, running perception together with the rest

of modules does not affect the execution time of *ProcEvents* processes.

3) The execution time of the *ImgCallback-Camera* event-triggered process of perception increases.
4) While all event-triggered processes of perception still lead to limited overall computing time theoretically fitting in a single hardware thread, the chances of any of them being significantly delayed by *ImgCallback-Camera* are high due to the longer duration of the latter.
5) The cumulative execution time of the processes of all other modules could theoretically fit into a single hardware thread, and none of them runs for too long, so if two such processes are triggered simultaneously, but serialized to run, the impact on the termination time for each one can only be tiny.

Hence, we use the following strategy for our proposed consolidation scenario (*ProposedCS*):

1) Perception's event-triggered processes. We allocate the two hardware threads of one core (e.g. core 1) so that ROS can schedule those processes there. In particular, we allocate c1t1 and c1t2 for those processes. ROS schedules those processes in those hardware threads without further user intervention. This strategy grants parallelism across different such processes while limiting resource utilization to a single core since their cumulative execution time is expected to fit.
2) *ProcEvents* processes. Each *ProcEvents* process is allocated to a specific hardware thread in a core, leaving the other hardware thread idle to limit impact due to contention to resources shared across cores, thus avoiding interference in the core pipeline, L1 and L2 caches. In particular, we allocate *ProcEvents-Fusion* to c2t1 and *ProcEvents-Lane* to c3t1.
3) We allocate a hardware thread of a different core to run the remaining modules' event-triggered processes. By providing such hardware thread (c4t1 in particular) to ROS for each one of the remaining modules, ROS schedules all those processes there.

To reach the *ProposedCS*, we followed the next methodology, which is straightforward and general enough to be applied in other AD frameworks:

1) Assign one hardware thread to each software process that runs all the time (e.g. *ProcEvents* functions in Apollo).
2) Fit as many processes that handle event-triggered functions as possible into the same hardware thread without affecting the execution time of any of them. Use as many hardware

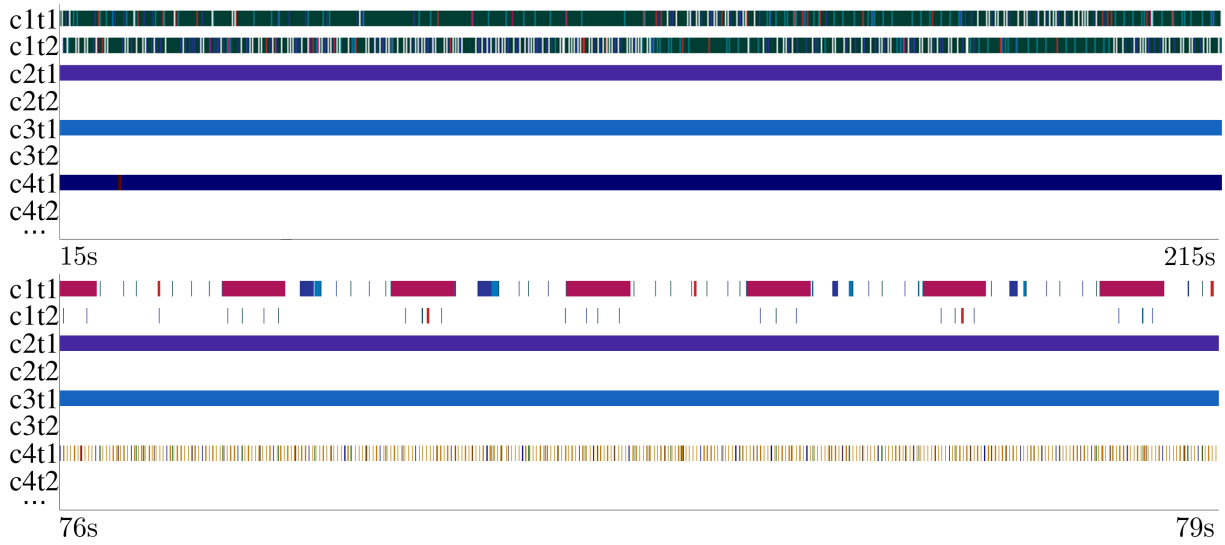

Fig. 9: Execution view of the entire Apollo with the proposed consolidation. The second view is the zoomed-in version of the first.

threads as needed until all these processes have a hardware thread assigned.

3) If some of the processes that are mapped to different hardware threads can reuse data from each other, put them in the same core if possible.

Figure 9 provides the execution view of the *ProposedCS*. As shown, perception's event-triggered processes run smoothly in c1t1 and c1t2. By granting both hardware threads instead of only one, some processes are allowed to run in c1t2 in parallel to *ImgCallback-Camera*. c2t1 and c3t1 are fully utilized, as expected. Finally, c4t1 suffices to run all those short-duration event-triggered processes of all modules but perception.

3) Performance analysis: We assess the performance impact on a per-process basis of the *ProposedCS* normalized w.r.t. the *DefaultCS* when Apollo is run in isolation. To that end, for each process, we use the median of all runs of 4,000 executions of Apollo. Results are shown in Figure 10, which excludes the *OnMonitor* process from the Control module since its execution time, as shown in Figure I, is extremely low (below 1μs) and hence within the margin of error of the measurement precision.

We observe that the *ProposedCS* in isolation often provides lower execution times (slowdown below 1.0x) than those of the *DefaultCS*. This occurs for short duration processes since, by running most of the times in the same core, they often reuse instructions and data that remain in cache across different invocations. Instead, in the *DefaultCS*, execution may often migrate

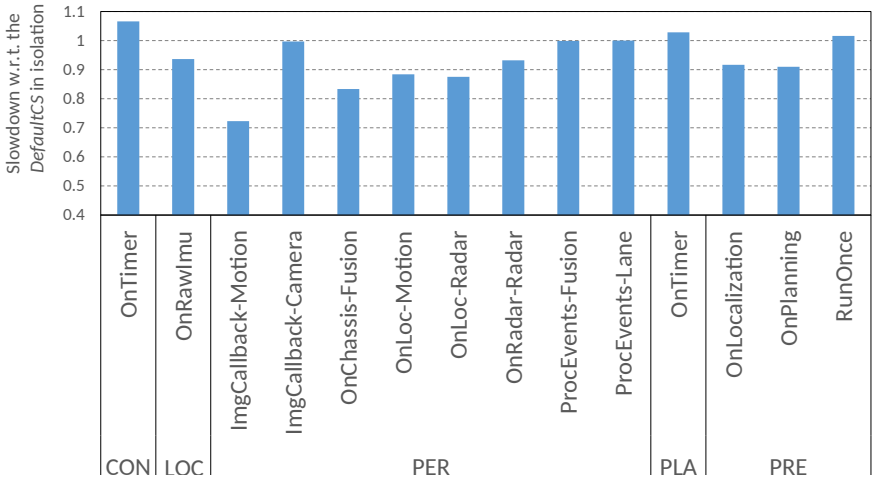

Fig. 10: Normalized execution times for the processes of the different modules of Apollo (excluding OnMonitor from the Control module) under *ProposedCS* w.r.t. *DefaultCS* when Apollo is run in isolation.

across different cores, therefore losing opportunities to reuse cache contents. Only two processes, *OnTimer* and *RunOnce*, experience some execution time increase (48 and 12 μs respectively). However, as shown in Figure I for the *DefaultCS* and *ProposedCS* in isolation (left 2 columns), reductions in other processes are larger in absolute terms (note that the table on the right shows the standard deviation). For instance, the difference between both scenarios $621 = 179990 - 179369$ μs for *ImgCallback-Camera*. Moreover, end-to-end execution times decrease because the execution time of the heaviest processes (perception ones) decreases.

		Execution Time						Standard Deviation					
		Isolation		OtherApps - L2		OtherApps - L3		Isolation		OtherApps - L2		OtherApps - L3	
		Default CS	Proposed CS	Default CS	Proposed CS	Default CS	Proposed CS	Default CS	Proposed CS	Default CS	Proposed CS	Default CS	Proposed CS
CON	OnMonitor	0	0	0	0	0	1	0	0	1	1	0	0
	OnTimer	725	773	1358	1046	1285	1470	3115	3034	4468	2802	1635	1610
LOC	OnRawImu	63	59	86	67	69	87	572	46	78	80	572	69
PER	ImgCallback-Motion	24	17	24	17	18	24	9279	5715	10388	4787	2932	4563
	ImgCallback-Camera	179990	179369	195540	184848	186201	191039	13385	9081	43239	23998	10151	11064
	OnChassis-Fusion	6	5	8	6	7	8	2	1	3	2	2	2
	OnLocalization-Motion	86	76	117	95	96	119	10176	6340	16371	9992	4498	4289
	OnLocalization-Radar	16	14	19	17	18	19	3	3	7	6	5	5
	OnRadar-Radar	5777	5382	6124	5439	5475	6161	11269	4239	19631	12371	3782	5731
	ProcEvents-Fusion	1120	1119	1123	1119	1119	1123	2253	2127	2499	2306	2104	2142
	ProcEvents-Lane	500078	499975	500085	500287	500384	499601	9043876	9044791	9042447	9042801	9044006	9043055
PLA	OnTimer	1179	1212	1532	1426	1485	1648	15340	8253	19557	12387	5105	9340
PRE	OnLocalization	24	22	32	27	27	33	4	4	9	8	8	10
	OnPlanning	210	191	250	211	216	241	36	4	281	329	57	65
	RunOnce	746	758	987	917	942	1012	529	4	1038	1256	614	645

TABLE I: Absolute execution times (in μs) for the processes of the different modules of Apollo: control, localization, perception, plannning, and prediction.

Overall, we manage to reduce resource utilization to less than half (only 5 hardware threads out of the 16 available and 4 out of the 8 cores available are used) while providing the same or even better performance. The unused cores can be put in low-power mode if they are not used or can run other software applications. The latter helps consolidating more software applications onto the same MPSoC reducing the overall hardware procurement costs, hence increasing margins.

C. Improved Integration with Other Applications

We also assess the benefits of deploying our *ProposedCS* when running several other applications along with Apollo with respect to Apollo's *DefaultCS*. This contributes to increasing resource utilization and hence reduce hardware procurement adhering to automotive cost-reduction requirements. We model this scenario by deploying a set of benchmarks that stress a specific shared resource in the CPU.

- *L2B*. This benchmark performs load/store operations that hit in the L2 cache of one core. We schedule 4 copies of this benchmark, which in the *DefaultCS* we let the operating system schedule freely. For the *ProposedCS*, we statically allocate each benchmark to one hardware thread of each of the 4 idle cores (cores from 5 to 8).
- *L3B*. This benchmark performs load/stores mostly hitting in the shared L3 cache. As for *L2B*, 4 copies are used and scheduled analogously.

Figure 11 provides normalized results for each process and configuration. As for Figure 10, we focus on the median of 4,000 runs and normalized w.r.t. the *DefaultCS* when running Apollo in isolation. For completeness, Figure I provides absolute execution time values for the median and standard deviations (STD) in μs.

When run against L2B, we observe how Apollo's execution time for the *DefaultCS* increases noticeably (the two middle columns in the left table of Figure I show absolute results and the left chart in Figure 11 speed up results). Since there is no control on how Apollo processes are scheduled, they end up using all cores. As a result, it is often the case that L2B benchmarks may end up running in the very same core as an Apollo process simultaneously, hence generating interference in the pipeline, and L1 and L2 caches. Also, even if Apollo processes do not execute simultaneously with L2 benchmarks in the same core, they may interleave their execution in some cores so that cache contents reuse across executions drastically decreases. Instead, for our *ProposedCS*, neither pipeline, nor L1, nor L2 cache interference is possible.

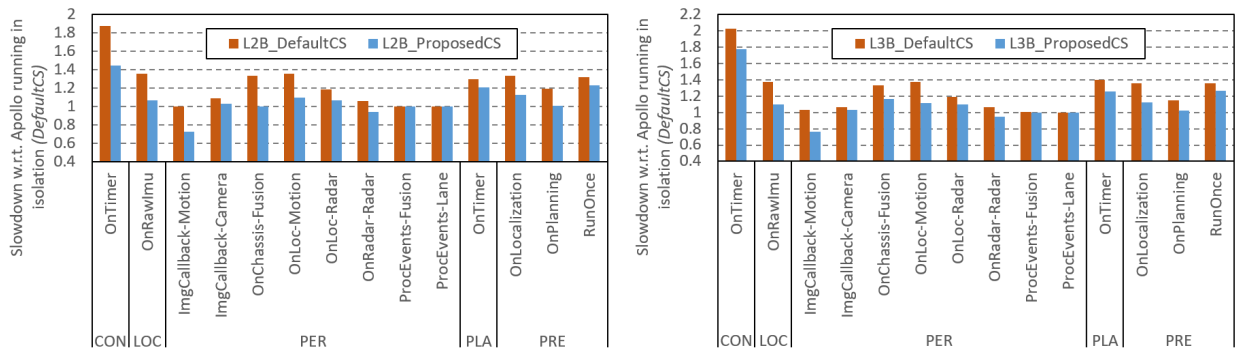

Fig. 11: Normalized execution times for the processes of the different modules of Apollo (excluding OnMonitor from the Control module). Values are normalized w.r.t. the *DefaultCS* when run in isolation.

When other applications are consolidated instead of the used benchmarks, the benefits will be mutual for all applications by limiting the interference those other applications can cause on Apollo, as well as the interference that Apollo can cause on the other applications. It is particularly interesting noting that, by binding processes to cores – and hence reducing the degrees of freedom to use the platform for Apollo – the execution time sensibly improves in most cases or remains roughly constant. This holds for one of the two largest processes (*ImgCallBack-Camera*), so gains do not only occur in relative terms, but also in absolute terms. Also note that the standard deviation generally decreases by building on our *ProposedCS*, thus increasing time predictability.

Running against the copies of L3B (two right columns in the left table of Figure I and right chart in Figure 11) leads to similar trends on Apollo's execution time to those observed for L2B, but with lower gains. This effectively reflects that the extra interference experienced by the *DefaultCS* w.r.t. our *ProposedCS* is caused by sharing core pipelines, and L1 and L2 caches, which are local per core. Instead, the L3 cache is shared across all hardware threads and cores, so interference in the L3 cache occurs regardless of the process-to-hardware thread mapping.

VI. Related Work

There are several AD frameworks though few of them are publicly available. Among those, Apollo [7] and Autoware [22] are the most popular ones. In this work, we focus on Apollo due to its industrial maturity and use in real-life systems, as explained in Section II-B.

V&V (including testing) of automotive software systems is a key concern in industry. The main reference safety concept for automotive systems, the E-Gas concept [23], includes the strategies to follow for V&V of automotive software systems. It fact, it is the basis upon which the reference safety standard, ISO 26262 [3], was developed.

Specific solutions for the V&V of automotive systems are abundant. The spectrum is broad including commercial tools for fault injection at hardware level to assess functional safety aspects across the whole platform, such as Synopsis's Certitude [24] (formerly SpringSoft's), practical experiences applying fault injection and mutation testing for software defect detection even for AD systems [25], testing approaches exploring different inputs of the system [26], [27], [28], and model-based design and testing solutions for classic automotive software [29] to name a few.

Some works consider V&V for AD frameworks. Authors in [30] expose the complexities of performing resource usage testing for AD frameworks, and present some guidance and realistic evidence about how resource usage testing can be successfully achieved, allowing end users to verify their safety-related real-time AD frameworks. Authors in [31] discuss that the overarching goal of developing an edge computing environment for autonomous vehicles is to have sufficient computing power, redundancy, and protection to ensure autonomous vehicle safety. Then, they examine state-of-the-art methods in these fields and consider possible solutions to these problems. Authors in [6] discuss their experience applying ISO 26262, as the practical safety standard for road vehicles, software safety standards to Apollo. They have quantitative and qualitative compliance metrics with certain ISO 26262 product architecture, deployment, and testing guidelines. Authors in [5] provide a literature review by analyzing and grading related works in order to analyze the state-of-the-art in software V&V in autonomous vehicles. Based on this study, authors chose a group of primary research for more in-depth review based on relevant parameters. Behles [32] examines the emerging industry best practices for assessing autonomous cyber-physical devices, as determined by interviews with experts in the field.

Our work departs from usual verification and validation concerns, being orthogonal to existing literature on that area. Instead, the target of our work is two-fold: we contribute to align Apollo software to ISO 26262 specific validation and verification (V&V) requirements; and we assess the efficiency of the deployment of AD frameworks and how resource efficiency can be improved while keeping functionality unaffected. This is achieved through the generation of dynamic views, as part of the automotive software development process, and augmenting such process with the

generation of execution views.

VII. Conclusions

The software architecture of AD frameworks is generally designed with high performance and maintainability as key objectives. Unfortunately, those principles often play against automotive software requirements such as safety-friendly automotive software architectures and resource efficiency to shave margins. This paper investigates those concerns for an industrial-level AD framework, Apollo, and exposes the challenges to generate dynamic views, a mandatory step of the architectural design of automotive software. We produce dynamic views for Apollo and further enrich the information they provide with *execution views*, which we propose to incorporate into automotive software development processes. Building on both, dynamic and execution views, we show Apollo's inefficiencies using computing resources and propose alternative process consolidation scenarios leading to more effective use of resources that brings two key benefits: (i) availability of computing cores to safely consolidate other applications onto the same platform, and (ii) higher performance by increasing isolation and reusing cache contents. Overall, our work contributes to an efficient validation, testing, and optimization of AD software, hence reducing development and deployment costs.

Declarations

- **Funding.** This work has been supported by the Spanish Ministry of Science and Innovation under grant PID2019-107255GBC21/ AEI/10.13039/501100011033, and the European Research Council (ERC) grant agreement No. 772773 (SuPerCom).
- **Conflict of interest/Competing interests.** The authors have no relevant financial or non-financial interests to disclose.
- **Data availability statement.** The data supporting the results reported in this work are available from the authors upon reasonable request.

References

[1] W. Ochieng and K. Sauer, "Urban road transport navigation: performance of the global positioning system after selective availability," *Transportation Research Part C: Emerging Technologies*, vol. 10, no. 3, pp. 171–187, 2002.

[2] Deloitte, *Semiconductors – the Next Wave Opportunities and winning strategies for semiconductor companies*, 2019. [Online]. Available: https://www2.deloitte.com/content/dam/Deloitte/cn/Documents/technology-media-telecommunications/deloitte-cn-tmt-semiconductors-the-next-wave-en-190422.pdf

[3] International Organization for Standardization, *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[4] ——, *ISO/PAS 21448. Road vehicles – Safety of the intended functionality*, 2019.

[5] N. Rajabli, F. Flammini, R. Nardone, and V. Vittorini, "Software verification and validation of safe autonomous cars: A systematic literature review," *IEEE Access*, vol. 9, pp. 4797–4819, 2021.

[6] H. Tabani, L. Kosmidis, J. Abella, F. J. Cazorla, and G. Bernat, "Assessing the adherence of an industrial autonomous driving framework to iso 26262 software guidelines," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[7] "Apollo, an open autonomous driving platform." http://apollo.auto/, 2018.

[8] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. Vijaykumar, "Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories," in *Proceedings of the 2000 international symposium on Low power electronics and design*, 2000, pp. 90–95.

[9] International Electrotechnical Comission, *IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Edition 2.0*, 2009.

[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, 2009.

[11] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[13] Valgrind Developers. (2019) Valgrind. [Online]. Available: http://valgrind.org/

[14] RapiTime, *www.rapitasystems.com*, 2008.

[15] (2021) Paraver: a flexible performance analysis tool. [Online]. Available: https://tools.bsc.es/paraver

[16] RapiTask, *https://www.rapitasystems.com/products/rapitask*.

[17] AMD. (2021) AMD Ryzen 7 1800X Processor. [Online]. Available: https://www.amd.com/en/products/cpu/amd-ryzen-7-1800x

[18] Nvidia. (2021) GEFORCE GTX 1080 Ti. [Online]. Available: https://www.nvidia.com/en-sg/geforce/products/10series/geforce-gtx-1080-ti/

[19] ——. (2021) NVIDIA DRIVE PX. [Online]. Available: https://www.nvidia.com/content/nvidiaGDC/sg/en_SG/self-driving-cars/drive-px/

[20] ——. (2021) Jetson TX2. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/

[21] ——. (2021) NVIDIA DRIVE AGX Developer Kit. [Online]. Available: https://developer.nvidia.com/drive/drive-agx

[22] "Autoware. An open autonomous driving platform." https://github.com/CPFL/Autoware/, 2016.

[23] GAS Workgroup, *Standardized E-Gas Monitoring Concept for Gasoline and Diesel Engine Control Units (version 6.0)*, 2015. [Online]. Available: http://docplayer.net/31264302-Standardized-e-gas-monitoring-concept-for-gasoline-and-diesel-engine-control-units.html

[24] Synopsys, "Certitude functional qualification system," 2017. [Online]. Available: https://www.synopsys.com/

[25] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, and F. Törner, "Early verification and validation according to iso 26262 by combining fault injection and mutation testing," pp. 164–179, 2014.

[26] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," *Commun. ACM*, vol. 62, no. 11, p. 137145, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3361566

[27] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Testing autonomous cars for feature interaction failures using many-objective search," in *International Conference on Automated Software Engineering*, 2018.

[28] C. Berger, "Accelerating regression testing for scaled self-driving cars with lightweight virtualization: A case study," in *International Workshop on Software Engineering for Smart Cyber-Physical Systems*, 2015.

[29] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz, "What is the benefit of a model-based design of embedded software systems in the car industry?" pp. 343–369, 2012.

[30] M. Alcon, H. Tabani, J. Abella, L. Kosmidis, and F. J. Cazorla, "En-route: on enabling resource usage testing for autonomous driving frameworks," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1953–1962.

[31] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[32] C. Behles, "A dissertation on the testing approaches of autonomous cyber-physical systems," Ph.D. dissertation, The University of Memphis, 2020.

APPENDIX

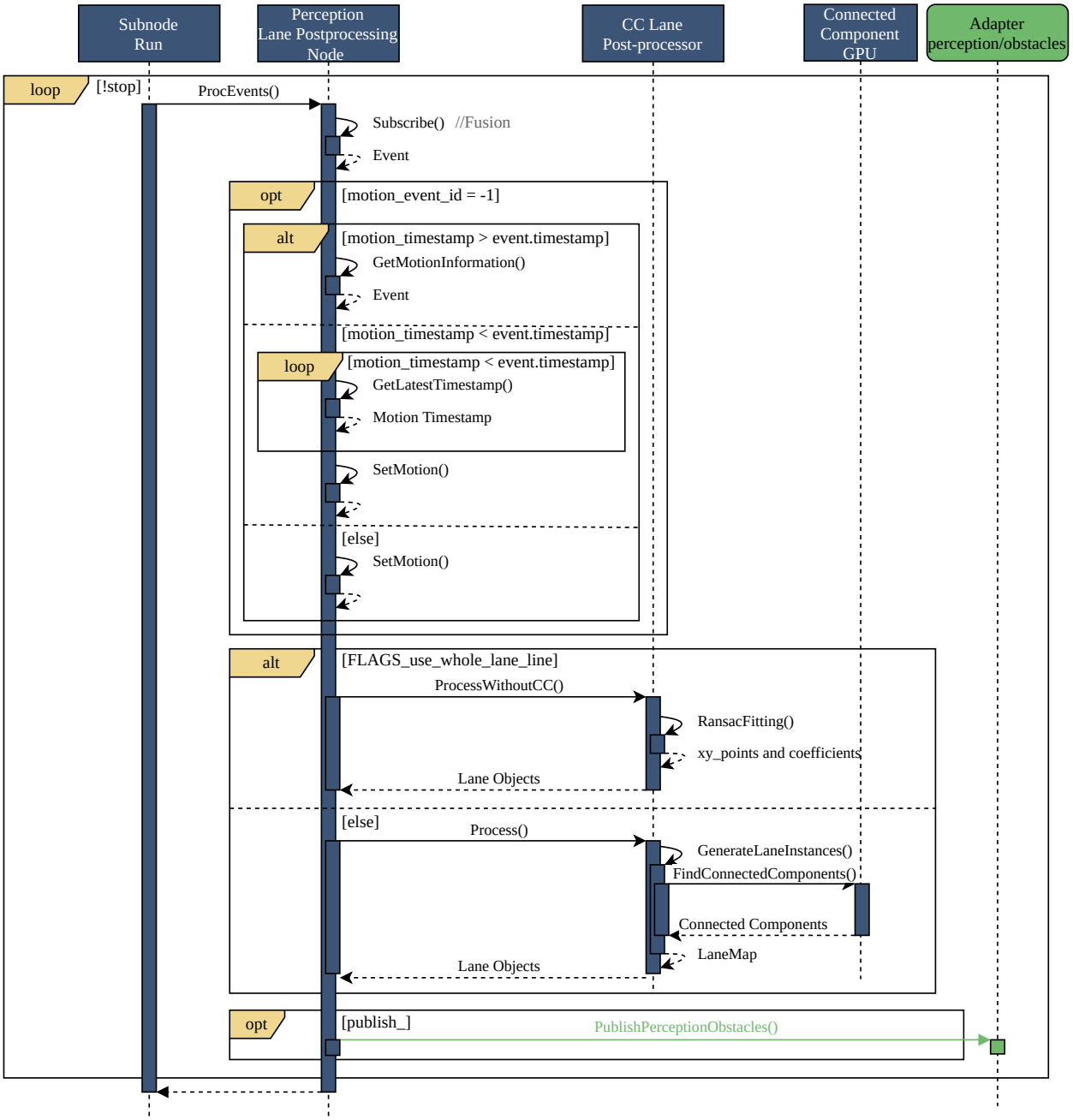

Fig. 12: Dynamic view of Perception's Lane post-processing node.

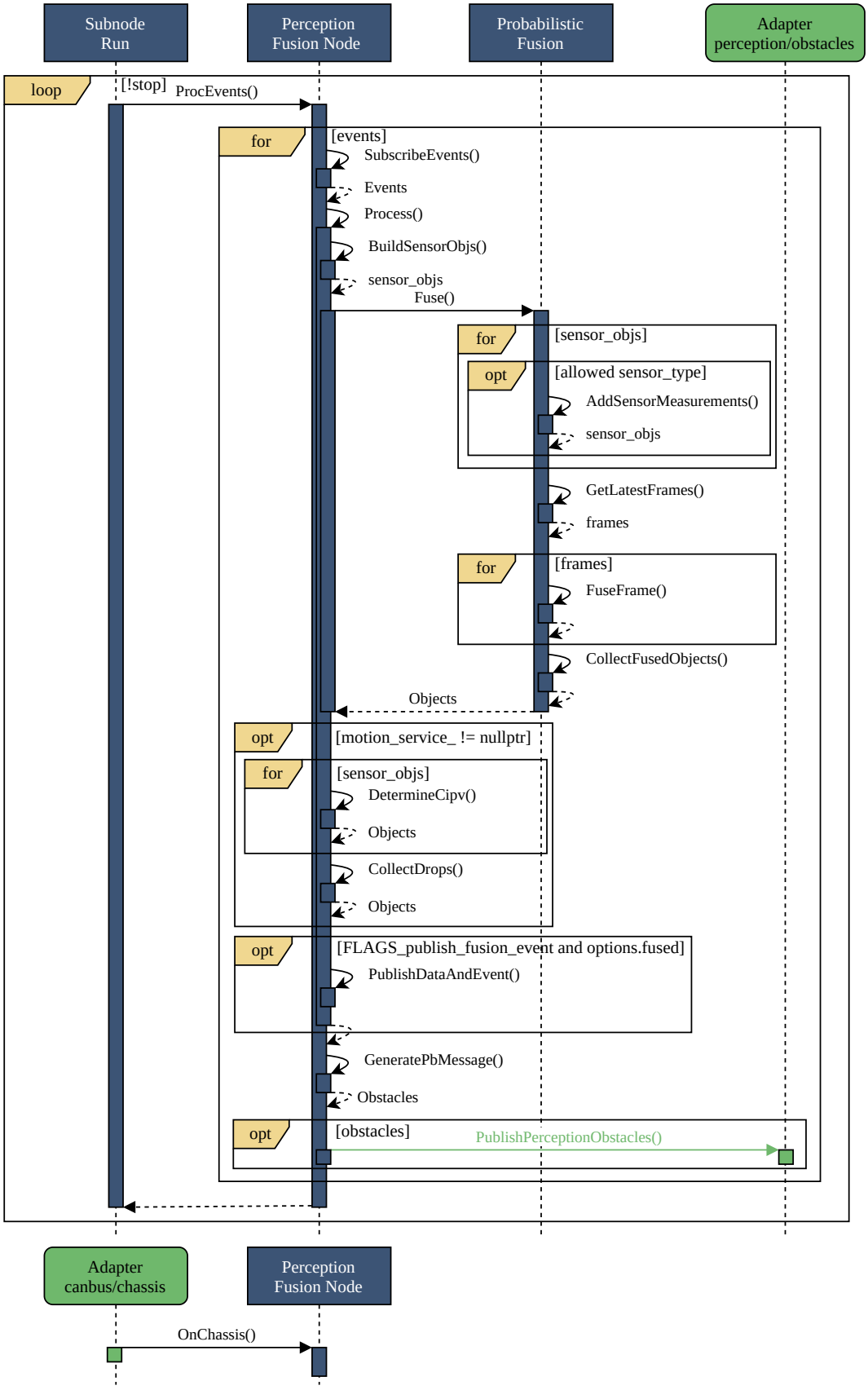

Fig. 13: Dynamic view of Perception's Fusion node.

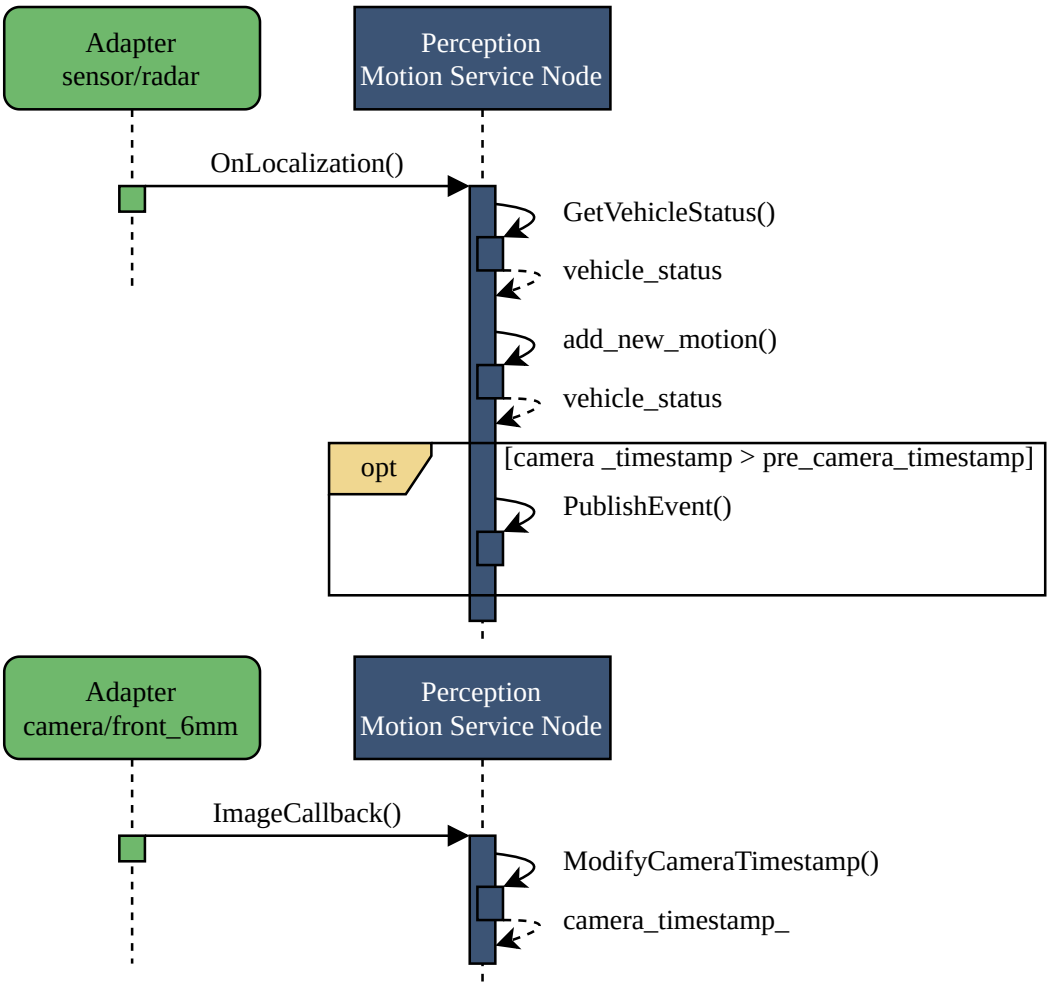

Fig. 14: Dynamic view of Perception's Motion Service node.

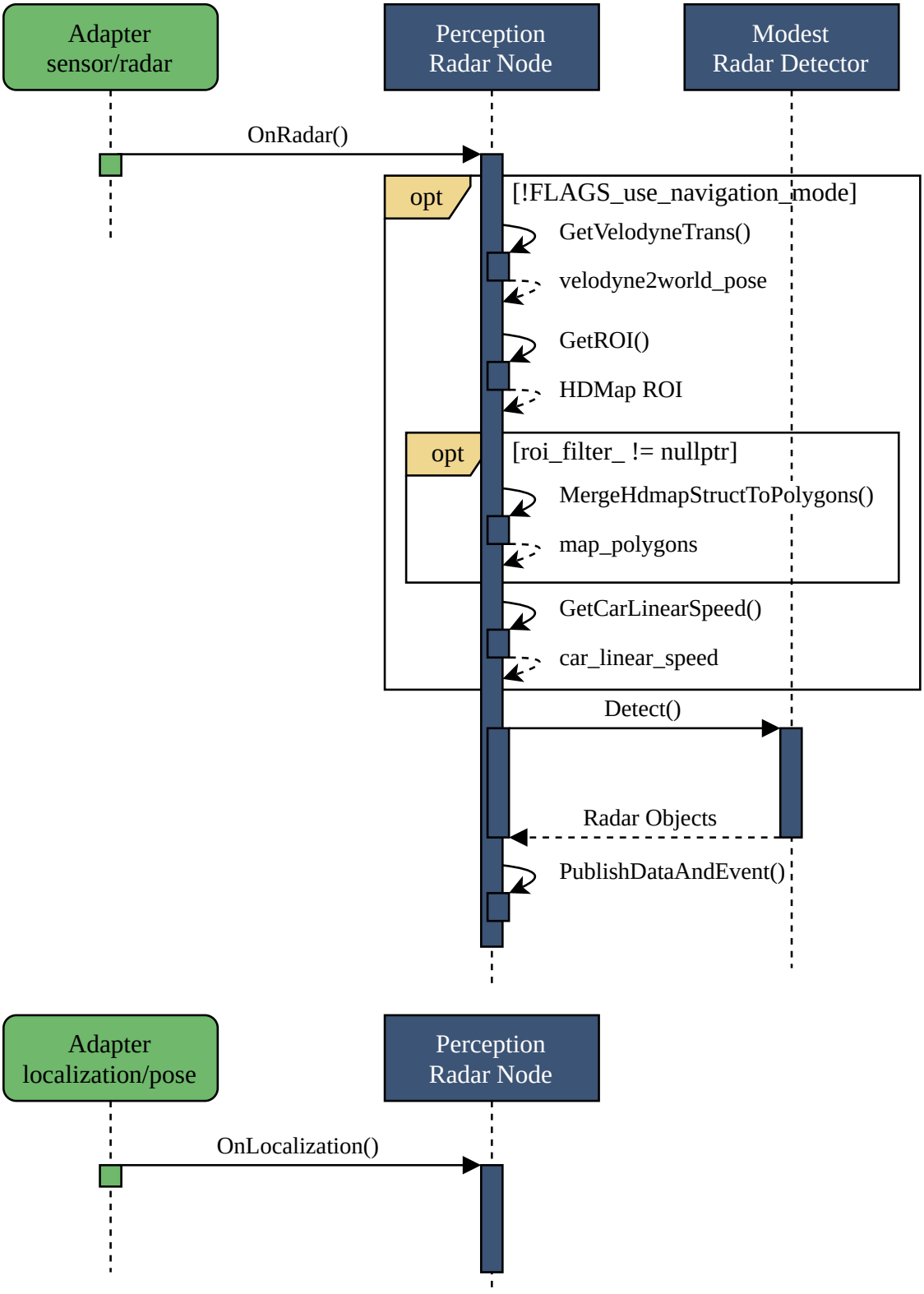

Fig. 15: Dynamic view of Perception's Radar node.

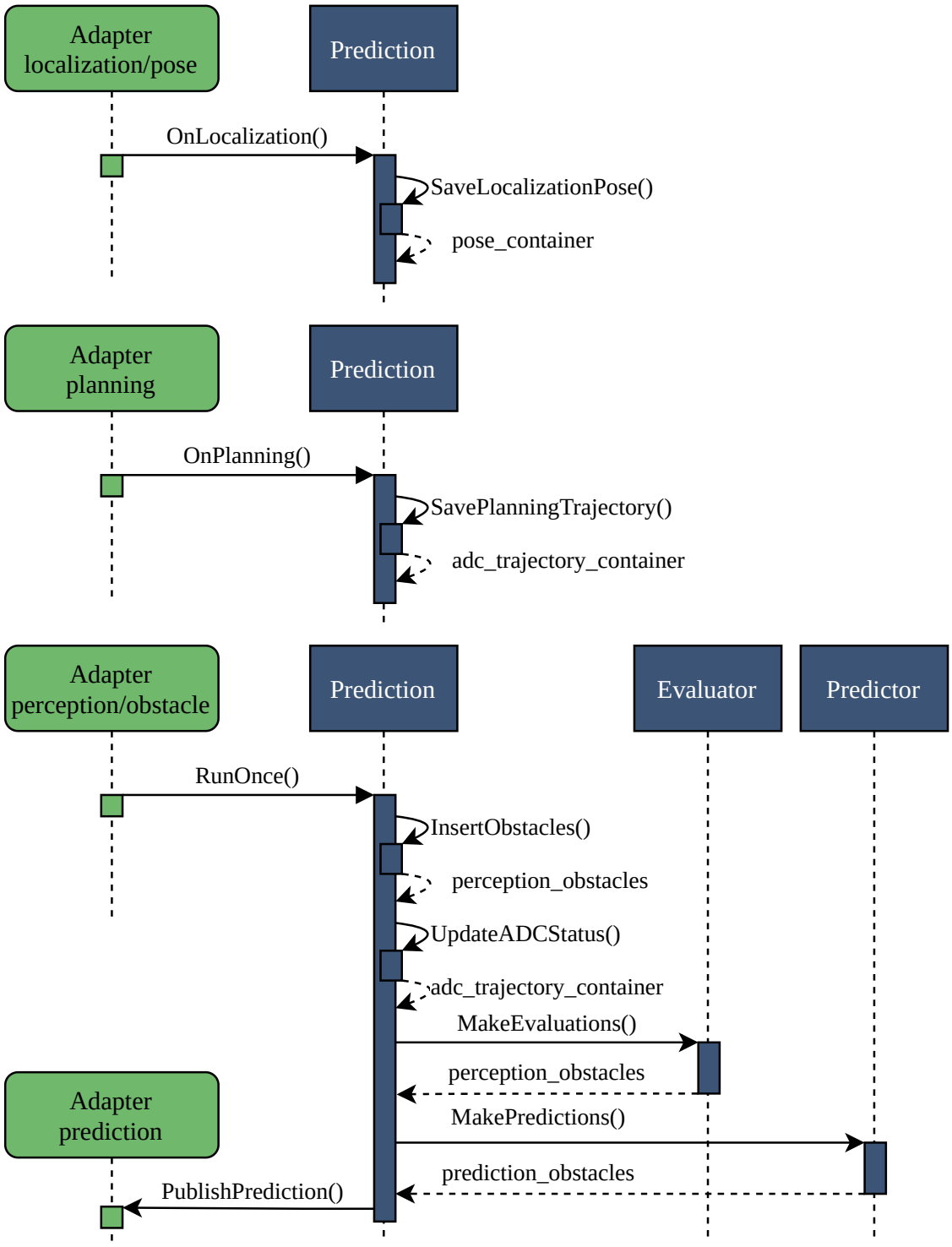

Fig. 16: Dynamic view of the Prediction module.

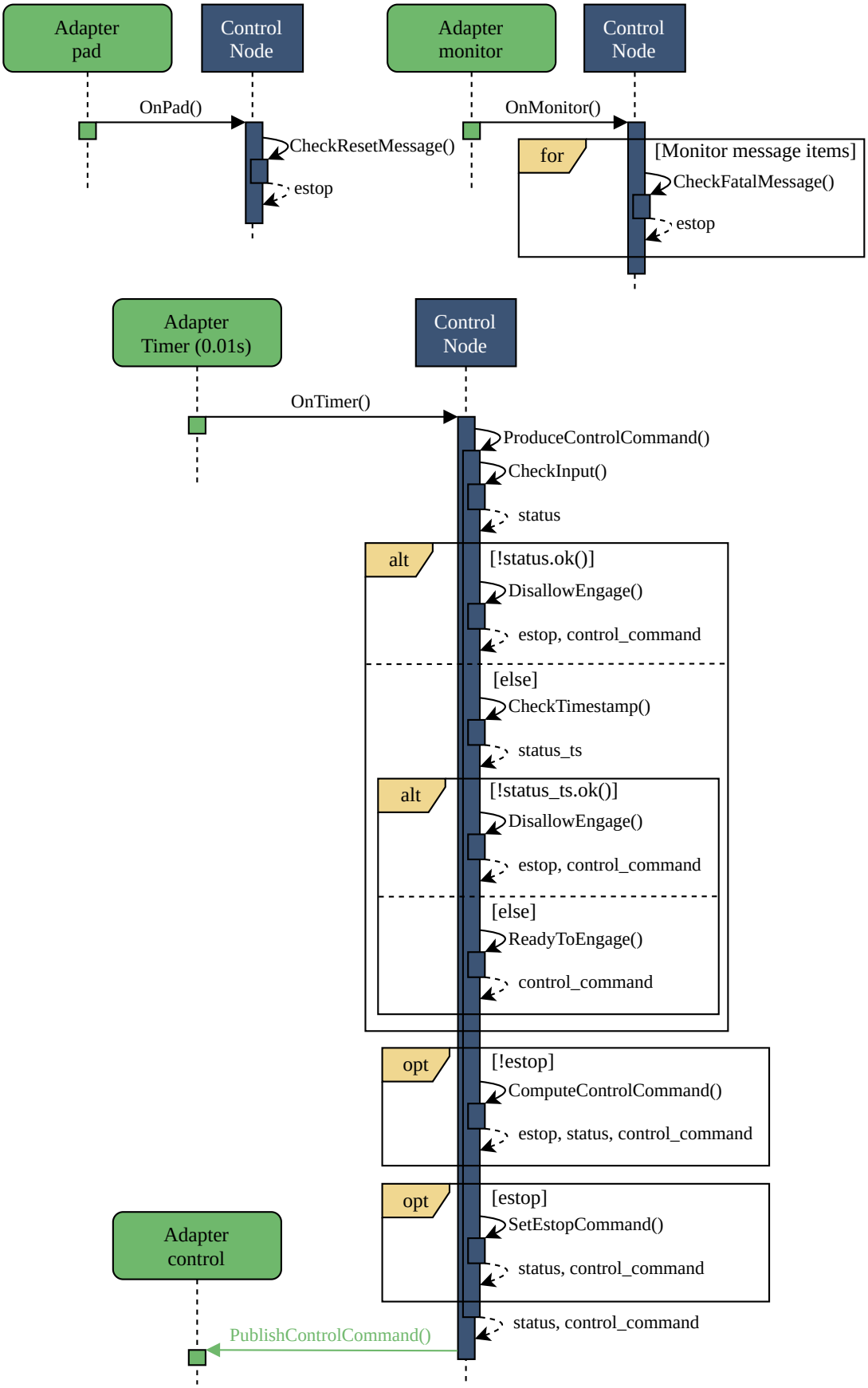

Fig. 17: Dynamic view of the Control module.