Platform-Independent QoS Parameters and Primitive APIs for Automotive Software

BaekGyu Kim[1] Chung-Wei Lin[2] Eunsuk Kang[3] Nobuyuki Tomatsu[1] Shinichi Shiraishi[4]

***Abstract*— Modern ITS (Intelligent Transportation System) applications are becoming increasingly sophisticated and diverse, following the growing trends of connectivity and autonomous driving. For example, a vehicle can be equipped with advanced sensors (*e.g.,* LIDAR, camera or radar) to perform complex driving maneuvers by itself and/or communicate with other vehicles or road-side infrastructures to assist drivers in safer driving. Since the operation of an automotive application is closely tied to the underlying platform characteristics, it requires significant integration effort when the application needs to operate on a wide range of platforms such as other vehicle types or road-side infrastructures. To reduce such effort, we propose a method to implement automotive applications in a platform-independent way. Our approach is to define the automotive domain-specific QoS (Quality of Service) parameters that allow an application to specify the expected quality of sensor input or actuator output independent of a particular platform for safe operation. The application interacts with a platform through several primitive APIs to inform QoS requirements, to read/write sensors/actuators values according to the specified QoS, and to handle exceptions upon any QoS violation. We implement a multi-mode ACC (Adaptive Cruise Control) application on a RC-car platform to demonstrate how an automotive application can be implemented based on the proposed QoS parameters and primitives.**

I. Introduction

Software plays a key role in the implementation and deployment of ITS (Intelligent Transportation System) applications, such as autonomous driving or traffic management. These types of software rely on not only on-board sensors and actuators, but also those that are equipped on remote vehicles or road-side infrastructures. Many examples can be found in the ARC-IT (Architecture Reference for Cooperative and Intelligent Transportation) project sponsored by USDOT [16]; for example, an ITS application may utilize the speed sensor input of remote vehicles to implement adaptive cruise control via wireless communication such as DSRC; another type of application may provide road-side lighting systems with location and motion information to control their brightness on demand. Such aspects of ITS applications show that modern vehicle software needs to interact with a wider range of sensors and actuators, in contrast to traditional vehicles that typically rely on on-board sensors and actuators only.

[1]Toyota InfoTechnology Center, U.S.A., Inc., Mountain View, California {bkim, ntomatsu}@us.toyota-itc.com
[2]National Taiwan University, Taipei, Taiwan cwlin@csie.ntu.edu.tw
[3]Carnegie Mellon University, Pittsburgh, Pennsylvania eskang@cmu.edu
[4]Toyota InfoTechonology Center Co., Ltd., Tokyo, Japan sshiraishi@jp.toyota-itc.com

This trend poses a significant challenge to automotive engineers, since control software needs to be integrated with a wide range of platforms such as other vehicle types or road-side infrastructures. As the reliable operation of the control software heavily depends on the quality of the underlying sensor input, the challenge is to achieve the same quality of control across different types of platforms that the software may be deployed on. For example, the adaptive cruise control needs to measure the distance of a front vehicle to determine acceleration or deceleration rates to maintain appropriate speeds and distances. The distance can be measured with various sensors (*e.g.,* on-board sensors or remote sensors) that provide different levels of accuracy, such as a millimeter wave radar sensor, a stereo camera or a LIDAR (Light Detection and Ranging). Therefore, engineers may be required to spend a significant amount of time to manually modify (or tune) software configuration parameters to the platform-specific sensor in order to achieve the same behavior across different platforms. The need to perform this time-consuming, manual activity for each new platform is a major bottleneck in developing a wide range of ITS applications.

A better method of ITS software development would be to allow engineers to implement an application in a platform-independent manner, while platform-specific details are hidden from them. Then, a separate mechanism can be provided to automatically check if the integration of the software with the particular platform can preserve the level of quality and accuracy that is demanded by the application uses.

Many prior academic works proposed general approaches to decouple the software development from the platform integration, for example, by providing reconfigurable components [4], designing APIs to hide the platform details [7], specifying the performance requirements [14] or preserving timing guarantee [6]. In particular, this trend can be also found in the recent development of Adaptive AUTOSAR standard [1] by providing a concept of manifest to explicitly specify the desired configuration of the platform on which the application can perform its intended operations under dynamic integration scenarios. Such a change from the classic AUTOSAR intends to accommodate the nature of ITS applications where the software needs to interact with a wide range of platforms. However, those works are rather intended for establishing a framework for general systems, and we believe that additional research needs to be done to refine such a general decoupling process for ITS applications in automotive domain.

To this end, we introduce the platform-independent QoS (Quality of Service) parameters and associated API primitives with which automotive software can be implemented in a platform-independent way. First, we categorize sensors and actuators according to their usages and extract their common properties that typically affect the performance of software.

Based on those hardware-specific properties, we identify QoS parameters that can be specified by software to guarantee its performance in a platform-independent way. Second, we introduce a set of primitive APIs for the software to interact with the next level of system layers (*i.e.,* a platform) according to the specified QoS parameters. The primitives are defined as set/get methods to read/write sensor/actuator data according to the specified QoS. With these primitives, software can request QoS requirements to the platform needed to appropriately control the vehicle. Then, the platform delivers the sensor data to the software as long as the QoS requirements are preserved. If the platform fails to meet the QoS requirements, it triggers an exception so that the software can handle it in an appropriate way (*e.g.,* deactivate the ADAS (Advanced Driver-Assistance Systems) feature or mode switching).

The contributions of this paper are as follows:

- A set of approximately 40 platform-independent QoS parameters for common automotive applications (Section III),
- A prototype implementation of APIs that provide primitive operations for the manipulation of these parameters (Section IV-A), and
- The demonstration of the feasibility of the proposed platform-independent development approach through a case study on a radio-controlled (RC) car (Section IV-B).

In addition, we provide a discussion of potential research problems to be tackled in order to make this style of development practical and applicable to a wide range of automotive applications (Section V).

II. Trends on software platform technology in automotive domain

Decoupling an application development process from a platform integration is an important trend in the automotive domain [5]. In particular, a future vehicle is expected to operate aftermarket applications developed by third party or to update and add new safety critical functions via the over-the-air update. This trend will fundamentally change the way how current OEMs (Original Equipment Manufacturers) develop software where engineers assume most software is pre-installed providing little rooms for end-users to reconfigure the vehicle system. We introduce two software platform technologies aligned with this trend.

A. AUTOSAR Adaptive

AUTOSAR Adaptive [1] is a new automotive architecture standard that offers flexible configurations of vehicle software. This standard is designed as an evolution of the AUTOSAR Classic standard, which allowed only static configurations. To provide this additional flexibility, AUTOSAR Adaptive explicitly distinguishes the application development process from its deployment process. An application code is allowed to be developed independently of a particular deployment scenario. Hence, the developed code can be deployed on a range of platform configurations later on.

Even though such a decoupling makes the application development process easier by hiding the details of the deployment scenario, it may also cause some issues due to lack of information of the underlying platform. For example, the application is not aware of available resources from the platform, such as computation resource (*e.g.,* the number of CPU cores, memory) and communication resource (data bandwidth between different electronic control units), to guarantee its correct operation; when it is deployed to a platform that has insufficient resources, the system cannot guarantee its intended operations.

To bridge this gap, AUTOSAR Adaptive introduces the manifest to specify the desired configuration of the platform on which the application can perform its intended operations. With this information, the deployment stage determines if the application can operate as intended on a particular deployment scenario. In addition, this manifest plays a role of limiting any non-intended behavior of an application at run time. That is, a platform utilizes the manifest to monitor the unintended behavior of an application at runtime, and limit its operation upon violation, such as exceeding allowed computation resource usages.

B. Publish-Subscriber Communication

The publisher-subscriber communication model is designed to decouple the communication entity that produces data (*i.e.,* publisher) from another entity that consumes the data (*i.e.,* subscriber). More specifically, a publisher may produce data without knowing who will consume it, and a subscriber may receive the data without knowing who produced it. This compares to the client-server communication model where clients and a server should know each other to exchange any information. Such decoupled communication increases flexibility in building a large scale distributed system in that one can easily add a new publisher or subscriber without requiring any changes of the existing system.

Even though both publisher and subscriber are not aware of each other in exchanging data, there are some cases, especially for the safety critical applications, where the communication needs to be performed according to some Quality-of-Service (QoS) requirements such as latency, delivery confirmation and so on. This implies that there should be a way to specify such non-functional requirements for the communication between publishers and subscribers.

There are two standards that adopt the publisher-subscriber communication model with such QoS consideration: Data Distribution Service (DDS) [13] and Message Queuing Telemetry Transport (MQTT) [12]. In MQTT, one can specify QoS parameters for the message delivery confirmation.

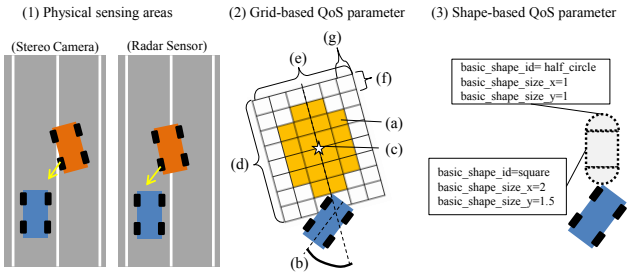

Fig. 1. Example characterization of QoS parameters for sensing area.

When a publisher and a subscriber need to exchange data on unreliable network, this parameter allows the communication to happen with a certain level of guarantee such as the message will be delivered at most once (*i.e.,* QoS level 0), at least once (*i.e.,* QoS level 1) and exactly once (*i.e.,* QoS level 2). In DDS, one can specify more diverse QoS parameters including the maximum acceptable delay (*i.e.,* LATENCY_BUDGET), the frequency of sample update (*i.e.,* DEADLINE) or the maximum duration of validity of the published data (*i.e.,* LIFESPAN).

As introduced in the earlier two standards, it is necessary to specify QoS requirements in a quantitative format so that the deployment process can use the information to check the correctness of the integration. However, the above QoS parameters are rather domain independent, which is necessary but not sufficient enough for automotive engineers to implement safety critical application in a platform-independent manner. In the rest of section, we introduce our preliminary result in defining automotive-domain specific QoS parameters that can be used to implement platform-independent software for vehicle systems.

III. QoS Parameter Identification

A. Categorization of Sensors and Actuators

Many automotive applications needs to guarantee various QoS requirements. For example, the adaptive cruise control (*i.e.,* an application) should correctly detect the distance of the front vehicle and maintain an appropriate distance and speed. Engineers need to take into account many hardware-dependent parameters to realize desirable QoS (*i.e.,* maintaining the distance and speed with a certain tolerance). Even though many sensors are capable of detecting the distance of the front vehicle, each has different detection area as illustrated in Fig. 1-(1). A stereo camera can sense a wider and shorter detection area compared to a radar sensor. In order to have the adaptive cruise control the same performance with different sensor types, engineers spend significant effort to finely tune the control parameters of the software to accommodate the sensor variation.

From the software engineering perspective, it is better to separate the programming concerns from such hardware dependent aspects. By doing so, the software will become reusable across a range of sensors and actuators. In addition, engineers can focus on implementing control logic without worrying about the changes of sensor and actuator specifications throughout the development process. To allow engineers to program at a higher level of abstraction, we define platform-independent QoS parameters.

TABLE I
CATEGORIZATION OF SENSORS AND ACTUATORS

Device Category	Examples Sensors and Actuators
Surrounding Environment	LIDAR, radar, camera (stereo, infrared), ultrasonic sensor, temperature/humidity/brightness/rain sensor.
Vehicle Control	Power train (engine, motor), yaw rate sensor, wheel speed, brake, brake/acceleration pedal, steering, shift position, suspension.
Cabin Control	Power seat, seat position, air conditioner, door sensor, door lock, side window, moonloof, room lamp, display, speaker, mic, side mirror indicator, mirror display, trunk opener/closer, seat belt, air bag, side mirror.
Diagnostics	Fuel tank, battery, tire pressure, device health.
Communication	DSRC, Bluetooth, WiFi, cellular network.

The platform-independent QoS parameters quantify the degree of quality assurance in reading or writing the environment input/output necessary to implement a certain automotive function. The parameters are platform-independent and the quantity is not associated to a particular sensor or actuator, but is associated to the functions to be implemented.

To extract such QoS parameters, we first categorize sensors and actuators according to their common purposes. This categorization is performed based on our prior knowledge of vehicle systems and the information of existing vehicle simulation tools [15]. Table I shows the example sensors and actuators that we consider to define the five category as follows.

Surrounding environment: Many automotive applications require a vehicle to recognize objects, such as preceding vehicles for the adaptive cruise control, lanes for the lane keep assist or pedestrians for pre-collision systems. The surrounding environment category includes sensors to recognize such objects in the close proximity of the vehicle, which allow applications to operate appropriately. Those applications are typically safety-critical functions that require recognition of the surrounding environment to be performed in a timely manner with high precision.

Vehicle control: Those safety-critical applications need to control the vehicle based on the sensor input, such as deceleration as a preceding vehicle reduces its speed, steering the wheel of the vehicle or applying a brake as a pedestrian steps onto the road. The vehicle control category includes sensors and actuators involved in controlling the vehicle according to the objective of each application. The vehicle control software typically calculates a new control input (*e.g.,* new speed), and the input needs to take in effect through the actuators with a certain level of quality assurance (*e.g.,* the set speed should be reached within x seconds).

Cabin control: Some applications also involve vehicle control for driver's comfort purposes, such as temperature control, door, mirror or light control. The cabin control category includes sensors and actuators engaged to provide in-vehicle

driving comfort. The cabin controls are typically non-safety critical so the precision of the sensors may require less strict quality assurance compared to the vehicle control category.

Diagnostics: There are vehicle components that need to be regularly monitored for its proper operation, such as fuel levels, battery or tire pressure. The diagnostics category includes sensors that can monitor such information to alert drivers to maintain the vehicle condition at a desirable level. The monitoring features typically do not require input as frequently as that of the surrounding environment category. However, sensor values should precisely reflect the current status of vehicle components.

Communication: Modern vehicle is equipped with communication modules to not only allow drivers to communicate with the vehicle with their hand-held devices (*e.g.,* smart phone), but also allow the vehicle to communicate with other vehicles or infrastructure, such as vehicle-to-vehicle (V2V) communication or vehicle-to-infrastructure (V2I) communication. The communication category includes the device that allows the vehicle to exchange information with other systems. Each application that utilizes the information from other systems requires the communication module to deliver information with a certain level of latency.

B. Defining QoS Parameters

We believe that the five categories reasonably characterize the intention of sensors and actuators that are typically equipped in a vehicle. Sensors and actuators from each category need to deliver the information to applications with their respective quality assurance. Based on this categorization, we define the QoS parameters by hiding the details of specific sensor or actuator characteristics.

Each category requires multiple quantities to be measured. For example, the surrounding environment category needs to measure speed, orientation/size of other vehicles or pedestrian for collision avoidance. It would be ideal to obtain those quantities with a perfect precision and zero latency so that an application can perform its functions as intended. However, there are many sources that make applications unable to utilize ideal quantities. For example, a camera cannot detect all objects with a perfect accuracy at nights; a camera needs additional computation time to decode and classify the image, which makes an application use the input after some delay. We define QoS parameters over such quantities to be measured. The QoS parameters need to be specified in a different way depending on the type of quantity. Here, we explain the example QoS parameters of the sensing area that is an important aspect to implement many autonomous features.

As illustrated in Fig. 1, a camera and a radar have different sensing areas. Table II shows two different ways to specify the sensing area illustrated in Fig. 1. The first one is the grid-based representation that allows an application to specify the sensing area in the form of a grid. It defines a rectangular area where multiple grid units are located with their unit sizes. The position of the area is specified as the center position and the orientation with respect to the vehicle position. An application can determine the necessary sensing area by specifying units in the grid. For example, each unit is assigned to an identification and the application can pass a set of identifications to the platform to express the sensing area. Then, the platform needs to check if the sensor has enough capability to detect the objects within this area. If it matches, the platform allows the application to utilize the quantity generated from the sensor; otherwise, the platform rejects the request from the application or generates warning signals, which implies that the sensor cannot detect objects according to the given QoS. The shape-based representation can be interpreted in a similar way, but we do not give details here. Engineers can choose an appropriate sensing area representation for the control objective.

Other QoS parameters may be specified in a relatively simpler form. For example, the QoS parameter of distance from the preceding vehicle can be specified in terms of how much errors can be tolerable in meters (*e.g.,* $\pm$ 3 meters). In the next section, we introduce API primitives that allow an application to interact with a platform to use the QoS parameters.

IV. Platform-Independent Primitive APIs

A. Three Primitive APIs

An application specifies the expected quality of the environment input and output for its safe operation in the form of the QoS parameters. The QoS parameters can then be used by a platform to deliver the corresponding environment input to the application and the application output to the environment according to the specified quality. We introduce three types of API primitives that make such an interaction possible between an application and a platform.

Get/Set value primitives: This primitive intends to set a new control output to the actuator (set-value primitive) or to get new sensor input (get-value primitive). An application calls the get-value primitive, and then the platform delivers the corresponding sensor input to the application. For example, we defined *getObjectInfo (v-area)* primitive for an application to receive the object information located in the area of interest specified as *v-area*. Note that the object detection can be done through a camera, a LIDAR or a radar sensor, but the application does not need to specify such a particular sensor type when it calls the primitive. The object information returned by a platform may include the object type (*e.g.,* pedestrian or vehicle), size or position.

An application calls the set-value primitive, and then the platform commands the actuator according to the set value. For example, we defined *setSpeed (p-spd)* primitive for an application to set a new vehicle speed specified as *p-spd*. Note that there are several ways to adjust vehicle speed, such as controlling a throttle, a gear or a brake. But, the application does not need to specify a specific way to reach the new speed when it calls the primitive.

Get/Set QoS primitives: This primitive intends to set QoS requirements to the value primitive or get QoS that a platform can currently provide. An application calls the set-QoS primitives to inform the platform of QoS requirements over the

TABLE II

EXAMPLE QOS PARAMETERS OF SENSING AREA

Variation	Item	Description
Grid-Based Representation	area_grid	Grid representation of area (Fig. 1(a))
	grid_orientation_local	Angle between in-vehicle local axes and grid orientation (Fig. 1(b))
	grid_orientation_global	Angle between global axes and grid orientation
	grid_center_position_local (x,y)	Position in in-vehicle local coordinates (Fig. 1(c))
	grid_center_position_global (longitude, latitude)	Position in global coordinates
	number_of_grid_x	Number of x-axis grids (Fig. 1(d))
	number_of_grid_y	Number of y-axis grids (Fig. 1(e))
	grid_length_x	Length of an x-axis grid (Fig. 1(f))
	grid_length_y	Length of a y-axis grid (Fig. 1(g))
Shape-Based Representation	number_of_basic_shape	Number of shapes
	basic_shape_id	Array of IDs of shapes (e.g., square: 0, circle: 1, half circle: 2)
	basic_shape_size_x	Array of sizes along x-axis of shapes
	basic_shape_size_y	Array of sizes along y-axis of shapes
	basic_shape_position_x	Array of positions along x-axis of shapes
	basic_shape_position_y	Array of positions along y-axis of shapes
	basic_shape_orientation	Array of orientations of shapes

quantity to be measured or to be set. For example, we defined *setQoS-ObjectInfo (q-area)* primitive for an application to specify the quality of the area to be observed as described in Table II. With this information, the platform checks if the equipped sensor can detect objects within the area; if the sensor can cover the area, the platform returns the object information when the corresponding value primitive (*i.e., getObjectInfo*) gets called by the application. Similarly, we defined *setQoS-setSpeed (q-spd)* primitive for an application to specify the quality of the speed. One can specify the QoS parameter (*q-spd*) as to, for example, how fast the target speed should be reached (*e.g.,* the target speed should be reached within 5 seconds) or how much difference between the actual speed and the set speed is tolerable (*e.g.,* the speed difference should be within $\pm$ 1 mph). The platform checks if the equipped actuator can reach the set speed according to the QoS requirement when the corresponding value primitive (*i.e., setSpeed*) gets called by the application.

On the other hand, an application calls the get-QoS primitives to obtain the QoS that a platform is currently providing. The platform returns this information in a similar form that the set-QoS primitive passes as a parameter. Note that the return value of the get-QoS primitive is not same as the QoS requirement specified in the set QoS primitive. This result of the get-QoS primitive can be used by an application that need to react differently depending on the QoS level at run time (*e.g.,* a mode change in the control algorithm). We introduce this usage in detail in the case study.

QoS exception primitives: This primitive intends to inform an application of any violation of QoS that has been requested through the set-QoS primitive. An application first registers an exception handler that needs to be asynchronously called when the platform cannot guarantee QoS in delivering sensor/actuator value. The platform detects any QoS violation and then calls the registered exception handler. The application is responsible for specifying what needs to be done upon such an exception occurrence. Considering a pre-collision system that automatically applies a brake upon detection of any object in the frontal area of the vehicle,

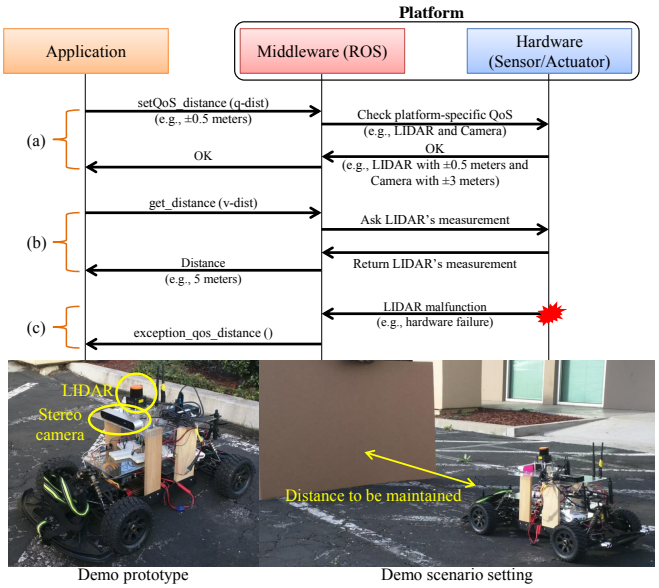

Fig. 2. Example of the primitive usage and the experimental setup. A video can be viewed at `https://youtu.be/uxkVmeA5GWQ`.

the platform may deliver the object information through a stereo camera. However, the camera may not be able to detect objects as the surrounding environment becomes darker due to the changing weather conditions, encounter other obstacles or hardware malfunctions (*e.g.,* a dirty camera lense). In this case, the application can implement an exception handler that triggers alarm sounds to inform drivers to perform more cautious driving since the pre-collision system cannot work as intended.

B. Demo Prototyping

We implemented the prototype system to demonstrate the usage of the aforementioned API primitives. Fig. 2 illustrates the RC-car demo system and the usage of the primitives in the case study. The demo platform consists of the RC car chassis (1/5 scale of actual car) and NVIDIA Jetson TK1 board running ROS (Robot Operating Systems) that controls

the RC car motor; additional sensors are integrated to the board, such as a LIDAR, a stereo camera, an IMU sensor and an encoder.

We implemented a multi-mode ACC (adaptive cruise control) application to demonstrate the usage of the primitives. The main objective of this application is to maintain a constant distance from the preceding vehicle. Our application implements three modes depending on the quality of sensor input that determines the distance from the preceding vehicle. If the sensor input guarantees the distance accuracy on the order of centimeters (*e.g.,* the distance error is within 0.5 meters), the ACC stays in the mode to maintain a short distance. If the sensor input guarantees distance accuracy on the order of meters (*e.g.,* the distance error is less within 3 meters), the ACC switches to another mode to maintain a wider gap. If the sensor input cannot guarantee any bounded accuracy, the ACC is deactivated.

Fig. 2 shows the example primitive calls made in the demonstration. In Fig. 2(a) the ACC application sets the QoS requirements to maintain the short distance (*e.g.,* $\pm$ 0.5 meters) through the set QoS primitive. Then, the platform checks if the equipped hardware is able to provide the distance measurement according to the QoS requirement, then acknowledge the application. We use two sensors to measure the distance, a LIDAR and a stereo camera. A LIDAR is known to have a higher accuracy in detecting distances from obstacles compared to a stereo camera. Before starting the demonstration, we perform an experiment to calibrate and obtain the accuracy of the LIDAR and camera; we have the platform maintain this information. When the application calls the set QoS primitive, the platform uses this information to decide if the QoS requirements can be satisfied. Since both LIDAR and camera are available to use in the beginning, the application is informed with OK.

In Fig. 2(b), since the QoS can be guaranteed, the application can maintain the short distance based on the LIDAR input (note that camera input is ignored by the platform at this moment since its measurement does not match with the QoS requirements). In Fig. 2(c), we detach the LIDAR physically to simulate a potential hardware malfunction. This makes the platform unable to receive LIDAR input any more, but the camera input is still available. However, the camera input cannot guarantee the QoS requirement requested by the current mode of ACC application, so the platform triggers the exception to let the application automatically switch to the conservative mode to maintain a larger gap. We further detach the camera physically to simulate another potential hardware malfunction, and the ACC is deactivated for safety guarantee. A video can be viewed at `https://youtu.be/uxkVmeA5GWQ`.

Through this case study, we showed how the application can be designed using the platform-independent primitives by hiding the details of the hardware dependent aspects. We believe this way of implementing automotive software will make engineers implement applications easier and in a more reusable way across various hardware platforms.

V. Discussion

While the preliminary experiments show a promise for the proposed platform-independent approach, a number of research challenges remain to further improve and make this style of development applicable to a wide range of automotive applications.

This paper has focused primarily on QoS parameters, but there are other types of system properties that may be desirable to specify in a platform-independent manner, such as security, availability, and reliability. For instance, depending on the security requirements of an application (e.g., confidentiality or integrity), the engineer may specify certain data elements to be encrypted or signed before being transmitted over a network. The middleware would then select and apply appropriate cryptographic primitives (e.g., RSA or SHA) to fulfill the given specification, while hiding the details of these primitives from the engineer. Depending on the computational capacity of the underlying hardware, cryptographic primitives of varying strengths would be considered for selection (e.g., some resource-constrained devices may be limited to shorter key lengths). Prior works on security-aware platform mappings [17] may be a promising place to start for this problem.

Certain QoS parameters, such as latency and accuracy of sensor data acquisition, may be in conflict with each other, and so it may not always be possible to fully satisfy the engineer's requirements. In such cases, it may be useful for an API to expose options for the engineer to indiciate priorities among different parameters. Alternatively, when conflicting parameters yield multiple parameter choices, the middleware may perform a multi-objective optimization procedure [11] to enumerate the space of pareto-optimal solutions and present them to the engineer.

With the identified QoS parameters, another research direction would be to utilize them to automate some of the development process. For example, one can use the QoS parameters to automatically generate platform-specific source code in a way that meets timing requirements [9] or architectural requirements [10]. In addition, when a platform-independent software needs to be integrated with a particular platform, one can also use the platform-specific verification technique to formally prove the correctness of the integration [8].

Better language support is needed for specifying and reasoning about QoS guarantees provided by the underlying platform. For example, the developer of an API may annotate its key operations with pre- and post-conditions that describe the expected properties of QoS parameters (similar to functional specifications in languages like Java [3] or C# [2]). This specification could then be leveraged by a static analysis tool to reason about the overall properties of the application that makes use of the API; conversely, the same specification may be leveraged by the API developer to analyze its implementation for potential bugs.

Currently, in our approach, an API primitive throws an exception when it fails to fulfill the given QoS requirements.

However, in certain use cases (especially those that involve the control of safety-critical hardware), it may not be appropriate to rely solely on the client application to handle such exceptions. Instead, in the case of an unexpected hardware failure, the underlying platform may perform its own fault-tolerant actions by, for example, enabling a backup hardware device or disabling the application access to certain actuators. More generally, it may be necessary to design the platform with certain safety invariants that it must attempt to maintain at all times, regardless of application requirements or the status of hardware devices.

VI. Conclusion

Many modern ITS applications rely on various sensor inputs, and sensor performance is being improved rapidly to which those applications need to adapt accordingly. Furthermore, the concept of sensors is further expanding with the recent trend of connected vehicles. Any source, such as another vehicle or road-side infrastructure, can play a sensing role by providing useful information over the network. To meet this trend, it is necessary to appropriately virtualize the platform so that automotive applications can be developed in an independent way. Our platform-independent QoS parameters allow applications to specify their own requirements independently of a particular sensor. Then, the application can use the proposed primitive APIs to communicate with the platform and perform its operation according to the specified QoS. As a result, the application can be easily and systematically integrated with a wide range of sensor types to assure the system-wide quality assurance. In the future, we plan to extract more QoS parameters for connected vehicles to facilitate the development of more intelligent applications.

References

[1] AUTOSAR. AUTOSAR Adaptive Platform. `https://www.autosar.org/standards/adaptive-platform`. [Online].

[2] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[4] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The runes middleware: a reconfigurable component-based approach to networked embedded systems. In *2005 IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications*, volume 2, pages 806–810 Vol. 2, Sept 2005.

[5] D. Gangadharan, J. H. Kim, O. Sokolsky, B. Kim, C.-W. Lin, S. Shiraishi, and I. Lee. Platform-based plug and play of automotive safety features: Challenges and directions (invited paper). In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 76–84, Aug 2016.

[6] T. A. Henzinger and C. M. Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):33, 2007.

[7] A. A. Jerraya and W. Wolf. Hardware/software interface codesign for embedded systems. *Computer*, 38(2):63–69, Feb 2005.

[8] B. Kim, L. Feng, L. T. X. Phan, O. Sokolsky, and I. Lee. Platform-specific timing verification framework in model-based implementation. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 235–240, March 2015.

[9] B. Kim, L. Feng, O. Sokolsky, and I. Lee. Platform-specific code generation from platform-independent timed models. In *2015 IEEE Real-Time Systems Symposium*, pages 75–86, Dec 2015.

[10] B. Kim, L. T. X. Phan, O. Sokolsky, and L. Lee. Platform-dependent code generation for embedded real-time software. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10, Sept 2013.

[11] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, Apr 2004.

[12] OASIS. MQTT version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, 2014. [Online].

[13] Object Management Group. DDS version 1.4. http://www.omg.org/spec/DDS/1.4, 2015. [Online].

[14] D. C. Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6):43–48, 2002.

[15] TASS International. PreScan. `https://www.tassinternational.com/prescan`. [Online].

[16] USDOT. Architecture reference for cooperative and intelligent transportation.

[17] B. Zheng, C.-W. Lin, H. Yu, H. Liang, and Q. Zhu. CONVINCE: a cross-layer modeling, exploration and validation framework for next-generation connected vehicles. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016*, page 37, 2016.