INVITED: Runtime Monitoring for Safety of Intelligent Vehicles

Kosuke Watanabe, Eunsuk Kang, Chung-Wei Lin, Shinichi Shiraishi
Toyota InfoTechnology Center, USA, Inc.
{kwatanabe,ekang,cwlin,sshiraish}@us.toyota-itc.com

ABSTRACT

Advanced driver-assistance systems (ADAS), autonomous driving, and connectivity have enabled a range of new features, but also made automotive design more complex than ever. Formal verification can be applied to establish functional correctness, but its scalability is limited due to the sheer complexity of a modern automotive system. To manage high complexity and limited development resources, one alternative is to apply *runtime monitoring* techniques to detect when the system transitions into an unsafe state (i.e., one where it violates a critical safety requirement). In this paper, we report on our experience integrating runtime monitoring into a development workflow and present practical design considerations on languages and tools from an industrial perspective. Using signal temporal logic (STL) [12] and the Breach [6] monitoring tool, we perform a case study showing how monitoring can be used to detect undesirable interactions between two ADAS features called *Cooperative Pile-up Mitigation System* (CPMS) and *False-Start Prevention System* (FPS). This is an initial step to utilize runtime monitoring to achieve high assurance in the design of intelligent vehicles.

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; • **Software and its engineering → Software verification and validation**; *Software safety*; Specification languages;

KEYWORDS

Automotive systems; autonomous vehicles; connected vehicles; formal verification; runtime monitoring; safety.

ACM Reference Format:
Kosuke Watanabe, Eunsuk Kang, Chung-Wei Lin, Shinichi Shiraishi. 2018. INVITED: Runtime Monitoring for Safety of Intelligent Vehicles. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3199856

1 INTRODUCTION

Automotive design has become more complex than ever due the advances of advanced driver-assistance systems (ADAS), autonomous driving, and connectivity between a vehicle and its surrounding environment. As a safety-critical system, safety is always one of the top goals in automotive systems, and addressing the safety of a modern vehicle with an increasingly complex design has become an emerging issue that brings many challenges to engineers.

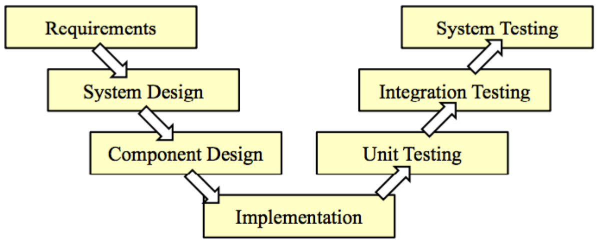


Figure 1: The V-model of automotive system development.

The traditional system development process in automotive industry is the V-model as shown in Figure 1 (from [19]). Usually, an *Original Equipment Manufacturer* (OEM) defines a set of system requirements and constructs a high-level design. The OEM also defines the specifications of individual components to be implemented accordingly by *suppliers*. If needed, the OEM and suppliers may negotiate the specifications and adjust the designs of the components. After implementation, different levels of testing are performed to check if the implemented components and their composition satisfy the corresponding specifications and system-level requirements.

To establish the satisfaction of a specification or requirement, formal verification can be applied to design models and implementations. However, formal verification is a computationally expensive task, and its scalability limits its applicability to systems of high complexity. As it is still challenging and too costly to apply verification to complex systems like an intelligent vehicle with autonomy and connectivity, *runtime monitoring* becomes a practical alternative to achieve safety goals. If a component or a system cannot be formally verified to satisfy its specifications or requirements, runtime monitoring can be deployed to detect and notify when there is any specification or requirement violation during runtime. In case of such a violation, the user may take appropriate actions, such as switching to a safe mode or stopping the system altogether.

In particular, runtime monitoring can be applied to support two types of development activities:

- **Type 1: Monitor a specification violation by a component**. Although suppliers have an incentive to design and implement components correctly to meet its specification given by the OEM, there are still several scenarios where safety violations may happen: (1) ill-defined or ambiguous specifications communicated to the suppliers, (2) design or implementation errors inside a component, and (3) hardware failures during operation. Note that, if applicable, formal verification is possible to mitigate (1) and (2) scenarios, but (3) cannot be prevented even with formal verification.

DAC '18, June 24–29, 2018, San Francisco, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3199856

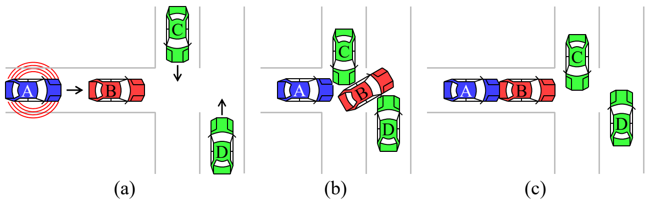


Figure 2: An illustration of CPMS. (a) Vehicle A fails to stop in time and informs Vehicle B the coming collision. (b) Without CPMS, Vehicle B is pushed into the intersection. (c) With CPMS, Vehicle B increases its brake torque and prevents from being pushed into the intersection.

- **Type 2: Monitor an assumption violation**. An assumption violation means an unexpected scenario or a scenario for which a component is not designed. This is not regarded as a specification violation, but it is still highly-related to safety as appropriate actions should be taken. For example, a specification of an autonomous function can be that "if it is not snowing, then the autonomous function must work." Even if a supplier implements the autonomous function, a runtime monitor which detects snowing and notifies the system or the user is still essential. It should be mentioned that this yields another desirable specification that "if it is snowing, then certain actions must be taken." However, it is not a specification of the autonomous function, but rather assigned to another component that uses the output of a monitor to trigger further safety actions.

In this paper, we report on our experience on the development of a prototype monitoring framework for vehicle safety systems. The main contributions of this paper include:

- We introduce our runtime monitoring workflow and the rationale behind our selections of Signal Temporal Logic (STL) [12] as the property specification language and Breach [6] as the monitoring tool.
- With STL and Breach, we perform a case study and create runtime monitors for the integration of two ADAS features, *Cooperation Pile-up Mitigation System* (CPMS) and *False-Start Prevention System* (FPS).
- We show that the runtime monitors successfully detect an assumption violation (Type 2 as mentioned above) within the integration of two ADAS features. It is an initial but necessary step for safety of future intelligent vehicles.

The paper is organized as follows. Section 2 and Section 3 introduce the two ADAS applications and the monitoring workflow, respectively. Section 4 presents our case study. The paper concludes with discussion of the related work in Section 5, and challenges of using runtime monitors and future research directions in Section 6.

2 MOTIVATING EXAMPLE

Modern vehicles are increasingly being equipped with the capability to modify or replace existing software applications with new ones via over-the-air (OTA) updates. Along the benefits of OTA, there are risks that newly updated software may interact with an application in unexpected ways, possibly resulting in a violation of the latter's assumption. Predicting all such interactions at the design time is extremely challenging, and thus monitors are needed to detect and address potential violations after the vehicle is deployed into the market. We describe an example of such interactions between two ADAS features called CPMS (an existing application developed by an OEM) and FPS (newly downloaded after deployment).

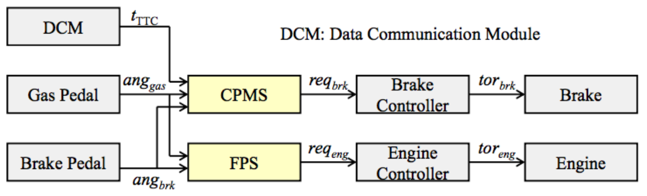


Figure 3: The architecture with the CPMS and FPS features installed. Edge labels indicate the type of data transmitted.

2.1 Cooperative Pile-up Mitigation System

CPMS is an ADAS application designed to leverage the V2V communication capability to improve safety. As shown in Figure 2, the objective of CPMS is to reduce secondary (but maybe more critical) damage of a coming-from-behind collision while waiting for a signal, stopping in a traffic jam, or running at slow speed. CPMS reduces the running distance after a collision and also notifies a potential collision in advance so that the human driver can take avoidance actions, where CPMS should not hinder the actions of the driver. The architecture is shown in Figure 3. Time-to-collision (TTC) is calculated by the following vehicle and sent to the ego vehicle via V2V communication. CPMS examines the received TTC and calculates the amount of brake torque as a request to the brake controller, which then processes the request and manipulates the brake actuator accordingly. However, if the driver intends to run away from the following vehicle by pressing the gas pedal, then CPMS cancels its prior brake request by sending an additional request with the brake torque set to 0. In our system architecture, we regard the sensors, actuators, and controllers except CPMS as "the environment of CPMS." Then, we define specifications that are realized by CPMS as *guarantees* and specifications that are realized by its environment as *assumptions*.

2.2 False-start Prevention System

FPS is another safety application that may be installed onto a vehicle. The objective of FPS is to prevent unexpected sudden start by an accidental press of the gas pedal. It detects the accidental press while the ego vehicle is stopping or running at slow speed and then controls the engine torque to prevent the vehicle from accelerating unintentionally. In particular, FPS determines whether the press is intended or accidental using the current vehicle speed and the degree and rate of changes in the angle of the gas pedal, and generates a request to the engine controller to reduce the engine torque to a predefined minimal value.

3 RUNTIME MONITORING WORKFLOW

Figure 4 illustrates the proposed workflow that leverages a runtime monitor to improve the safety of a vehicle during both development and deployment stages. Given a set of system requirements that a vehicle must be designed to satisfy, the engineer constructs contracts that formally specify the behavior of components to be

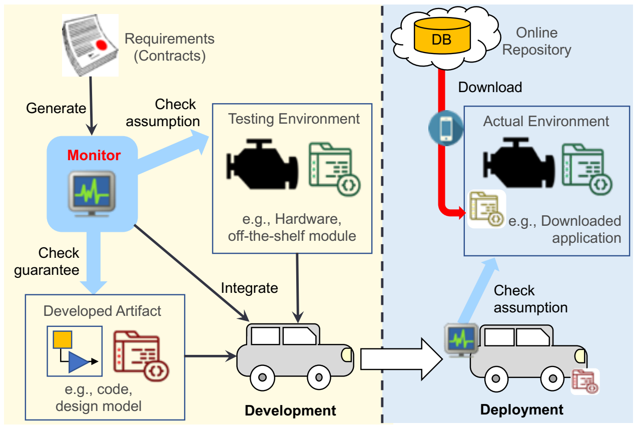


Figure 4: Overview of the runtime monitoring workflow.

implemented (in terms of guarantees) and the expectation on the environment (assumptions). From these contracts, a runtime monitor is derived using an automatic synthesis tool. The resulting monitor may be used to support two types of the development activities: (1) during the testing of a component implementation, to detect potential faults that may lead to a violation of its guarantee (Type 1 in Section 1), and (2) during system integration, to ensure that components do not violate the assumptions made by each other (Type 2 in Section 1).

During deployment, the monitor itself may remain integrated into the vehicle and be used to detect potential violations of an existing assumption by an application that is downloaded from an online repository. If such a violation is found, the monitor may perform an additional safety procedure to disable the new application and trigger a notification to the OEM, which may then produce a software update to mitigate this conflict.

In the rest of this section, we list the major elements of the proposed monitoring workflow and, for each one of these, describe a concrete technique or tool that we selected in order to realize the workflow, along with the rationale behind our decisions.

3.1 System Design Methodology

Design by contract [14] is a methodology for achieving high system assurance by assigning, to each component, a contract that describes its expected behavior and applying an analysis technique to check that the component satisfies its contract. *Assume-guarantee (AG) contracts* are a type of contract specification that structures the component behavior in two distinct parts—an *assumption* about its environment and a *guarantee* on its output [1]. AG contracts provide an intuitive way of structuring system specifications, with well-defined operators for manipulating and reasoning about satisfaction of contracts (including composition, decomposition, and refinement of one contract by another). In particular, the separation of specification into assumptions and guarantees is aligned with the two distinct use cases of runtime monitors described in Figure 4; that is, monitoring for the violation of an assumption by the environment or a guarantee by a component.

3.2 Property Specification Language

Two main factors were taken into consideration for the choice of a language for specifying contracts: (1) *expressiveness* of the language in being able to capture a wide range of requirements in automotive systems, and (2) the availability of *tool support* for simulation and verification against a property specified in the language. In particular, requirements related to ADAS features such as CPMS or FSP are typically time-sensitive (e.g., "the brake must be enabled within a certain number of seconds"), and so it was critical that the language to be considered would allow the engineer to readily specify such constraints.

Signal Temporal Logic (STL) is a specification logic designed for modeling and reasoning about the continuous behavior of a system over time [12]. This logic itself is an extension of linear temporal logic (LTL) [17] with an ability to specify properties over real values and real time. In the past, STL has been successfully applied to formally specify and analyze continuous and hybrid systems [4, 10, 15].

Let us briefly summarize the syntax and semantics of STL [12]. A *signal* s over domain D is a function $s : T \longrightarrow D$, where T represents the *time domain* in $\mathbb{R}_{\geq 0}$; the notation $x[t]$ for $t \in T$ denotes the value of x at time t. An STL formula takes the following form:

$$\varphi := u \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathbf{U}_{[a,b]} \varphi_2$$

where $a < b$ for $a, b \in \mathbb{Q}_{\geq 0}$, and $u(t)$ is an atomic predicate of the form $f(s_1[t], \ldots, s_k[t]) > 0$ for a set of k signals $\mathbf{s} = (s_1, \ldots, s_n)$ at some time t.

The satisfaction of STL formula φ by signal $\mathbf{s}$ at time t is defined as follows:

$$\begin{aligned}
(\mathbf{s}, t) \models u &\quad\Leftrightarrow\quad f(s_1[t], \ldots, s_k[t]) > 0 \\
(\mathbf{s}, t) \models \neg\varphi &\quad\Leftrightarrow\quad (\mathbf{s}, t) \not\models \varphi \\
(\mathbf{s}, t) \models \varphi_1 \wedge \varphi_2 &\quad\Leftrightarrow\quad (\mathbf{s}, t) \models \varphi_1 \wedge (\mathbf{s}, t) \models \varphi_2 \\
(\mathbf{s}, t) \models \varphi_1 \mathbf{U}_{[a,b]} \varphi_2 &\quad\Leftrightarrow\quad \exists t' \in [t+a, t+b] \cdot (\mathbf{s}, t') \models \varphi_2 \\
&\qquad\quad \wedge \forall t'' \in [t, t'], (s, t'') \models \varphi_1
\end{aligned}$$

The until operator **U** alone is sufficient to express two other types of temporal operators that are often useful in system specification—eventually (**F**) and always (**G**):

$$\mathbf{F}_{[a,b]}\varphi = \top \mathbf{U}_{[a,b]}\varphi \qquad \mathbf{G}_{[a,b]}\varphi = \neg \mathbf{F}_{[a,b]} \neg\varphi$$

A partial list of the STL formulas used to specify the system requirements and contracts for CPMS and FPS is shown in Figure 5. The formulas **R.1** and **R.2** describe the safety requirements that the vehicle must be designed to satisfy. In particular, **R.2** may be decomposed into (1) a guarantee to be provided by the CPMS feature and (2) an assumption about the environment of CPMS (which includes the input pedals and controllers that it communicates to) such that they together satisfy the overall requirement; i.e., **G.C** $\wedge$ **A.C** $\Rightarrow$ **R.2**. Note that the ADAS applications (CPMS and FPS) are capable of manipulating the engine and the brake only via requests to the controllers, and thus the formulas that describe their guarantees do not reference the actual torque values (i.e., tor_{brk} and tor_{eng}).

System Requirements

R.1: *If the vehicle is stationary and a collision is about to occur with the following vehicle, the brake must be enabled and the engine must be turned off until the collision occurs.*

$$v_{ego} = 0 \wedge t_{\mathtt{TTC}} \leq T_{\mathtt{safe}} \Rightarrow (tor_{brk} = TB_{\mathsf{CPMS}} \wedge tor_{eng} = TE_{\mathtt{min}})\, \mathbf{U}_{[t, t+t_{\mathtt{TTC}}]}\, (\mathtt{collision})$$

R.2: *If the gas pedal is pressed before the collision occurs, the brake must be disabled and the engine must be engaged depending on the angle of the gas pedal.*

$$v_{ego} \leq V_{\mathtt{slow}} \wedge ang_{gas} \geq \mathtt{GAS_{on}} \wedge \neg(\mathtt{collision}) \Rightarrow \mathbf{F}_{[t, t+T_{\mathtt{runaway}}]}(tor_{brk} = 0 \wedge tor_{eng} = f(ang_{gas}))$$

CPMS Assumption

A.C: *If the vehicle is stationary or rolling, and no brake request exists, the engine is engaged depending on the angle of the gas pedal.*

$$v_{ego} \leq V_{\mathtt{slow}} \wedge req_{brk} \leq TB_{\mathtt{off}} \Rightarrow \mathbf{F}_{[t, t+T_{\mathsf{CPMS}}]}(tor_{brk} = 0 \wedge tor_{eng} = f(ang_{gas}))$$

CPMS Guarantee

G.C: *If the gas pedal is pressed before the collision occurs, a request is generated to reduce the brake torque to zero.*

$$v_{ego} \leq V_{\mathtt{slow}} \wedge ang_{gas} \geq \mathtt{GAS_{on}} \wedge ang_{brk} \leq \mathtt{BRAKE_{off}} \wedge \neg(\mathtt{collision}) \Rightarrow \mathbf{F}_{[t, t+T_{\mathsf{CPMS}}]}(req_{brk} = 0)$$

FPS Guarantee

G.F: *If the gas pedal is pressed abruptly while the vehicle is stationary or rolling, a request is generated to reduce the engine torque to some minimal value.*

$$v_{ego} \leq V_{\mathtt{slow}} \wedge \Delta ang_{gas} \geq \Delta_{\mathtt{high}} \Rightarrow (req_{eng} \leq TE_{\mathtt{low}})\mathbf{U}_{[t, t+T_{\mathsf{FPS}}]}(ang_{gas} \leq \mathtt{GAS_{release}})$$

(v_{ego}: vehicle speed, $V_{\mathtt{slow}}$: rolling speed, $t_{\mathtt{TTC}}$: time to collision, $T_{\mathtt{safe}}$: min. time to avoid collision, T_{CPMS}: CPMS activation duration, T_{FPS}: FPS activation duration, $T_{\mathtt{runaway}}$: max. time before runaway, TB_{CPMS}: target brake torque by CPMS, $TE_{\mathtt{min}}$: min. engine torque, $\Delta_{\mathtt{high}}$: abrupt angle change, f: angle to torque translation.)

Figure 5: A partial list of system requirements and contract specifications for CPMS and FPS.

3.3 Monitoring Tool

Two practical considerations were taken into account during the selection of a monitoring tool: (1) the generated monitor must be compatible with the modeling environment and hardware platform used by automotive engineers at an OEM (in our case, Simulink/MATLAB) and (2) the tool must support *online* monitoring (i.e., the monitor must be able to compute the satisfaction of an STL formula *dynamically* as it observes its target, instead of requiring complete traces to be generated beforehand).

Breach [6] is a framework designed to enable formal analysis and monitoring of continuous and hybrid systems. Given a system property as a STL formula, the tool is capable of synthesizing an online monitor that detects when the property is falsified by a signal generated by the system. The monitor may be generated as a C++ program or a MATLAB S-function (which can be realized as a Simulink block), which enables seamless integration with other automotive controller models that we have constructed for our own simulation purposes.

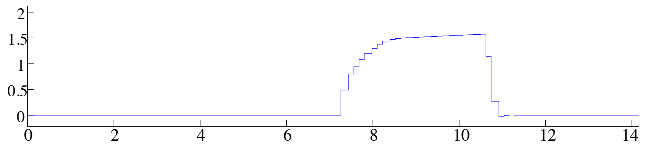
Another major feature of Breach is its ability to check not only the Boolean satisfaction of a formula, but also compute the *robustness* of the satisfaction. In particular, Breach extends the original semantics of STL (described above in Section 3.2) with the notion of *distance* between the actual signal and the value that determines the satisfaction or violation of a property. In continuous domains such as automotive where signals are subject to noise and errors, such quantitative notion of satisfaction is useful as a way of distinguishing marginal satisfactions from definite ones (even though this feature was not critical for our case study, as we were mainly concerned with detecting a violation, the robustness output was nevertheless useful for debugging our controller models).

4 CASE STUDY

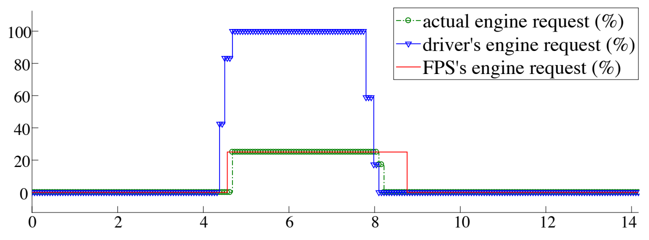
We present a case study on applying our runtime monitoring framework to detect possible interactions between two ADAS features (CPMS and FPS) in a vehicle simulation environment.

4.1 Implementation and Setup

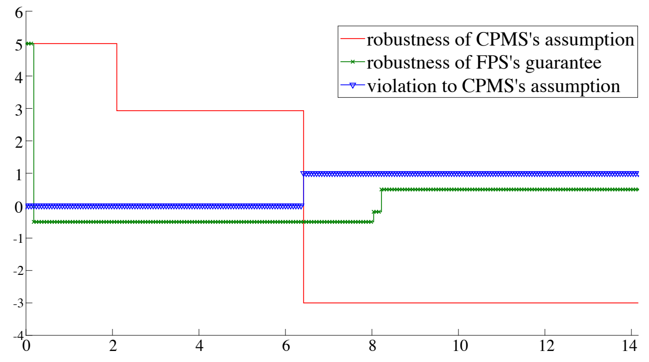
Vehicle controllers and monitors. To simulate the dynamics and behaviors of a vehicle, we constructed models of controllers that manipulate different actuators in the vehicle (e.g., engine, brake) as well as models of the ADAS applications (FPS and CPMS) and monitors. In particular, the main vehicle controllers were implemented and integrated inside Unity (https://unity3d.com), a popular game development environment that can be extended to perform simulation with advanced vehicle physics. The FPS and CPMS applications were implemented as Simulink models, and additional MATLAB functions were used to transmit the outputs of these applications to the Unity controllers (the communication method will be described shortly). Furthermore, Breach is capable of generating online monitors as Simulink blocks from STL formulas, and so the integration the monitors with the CPMS and FPS models was straightforward.
Communication framework. Runtime monitors and their targets (i.e., controllers whose behavior is being monitored) may not always reside on the same device. For instance, during design and testing phases, a monitor may be implemented as a program in a high-level language (e.g., C++), a Simulink/MATLAB block, or as a low-level FPGA component. To achieve flexibility to deploy and reuse the monitor on multiple platforms, we specifically developed a system architecture with a middleware called MQTT [16], a messaging protocol that allows components (possibly running on different devices) to communicate using a publish-subscribe pattern.



(a) Vehicle speed (km/h)



(b) Requests to the engine controller (%)



(c) Robustness for property satisfaction

Figure 6: Results from simulation of CPMS and FPS. The *x*-axis on each graph is measured in seconds.

In this architecture, the developer of a monitor or a controller only needs to adhere to an MQTT API for publishing and subscribing to messages, and does not need to be aware of the platform details of other components. In particular, Mosquitto (https://mosquitto.org), an open source implementation of MQTT, was used.

Simulation scenarios. For our simulation runs, our goal was to construct scenarios similar to those shown in Figure 2. In the initial system state, the ego vehicle (Vehicle B in the figure) is stationary at an intersection, with another vehicle (A) approaching the former and unable to stop in time to avoid a collision. In particular, Unity allows the construction of various scenarios by spawning multiple vehicles with an initial velocity and distances between them. At the time of the collision, we scripted the ego vehicle driver to press the gas pedal at the maximum angle to model the runaway action. We tested two different scenarions, depending on whether FPS is enabled or disabled, and measured the outputs of the ADAS controllers as well as their monitors.

4.2 Experimental Results

Figure 6 shows measurements from a simulation run with both the CPMS and FPS controllers enabled. In the corresponding scenario, the driver of the ego vehicle is notified of a potential collision from rear by CPMS and decides to run away by fully pressing on the gas pedal, which then results in the maximum engine request as shown in Figure 6b. At the same time, FPS detects that the gas pedal has been pressed abruptly (through the sudden change of its angle) and subsequently generates requests to reduce the engine torque to a minimal value ($\leq$ 20%). As a result, the vehicle fails to accelerate and run away from the collision despite the design intent of CPMS.

Figure 6c shows the robustness output from the Breach monitors for the satisfaction of the CPMS assumption **A.C** and FPS guarantee **G.F** (stated as STL formulas in Figure 5). An additional monitor was allocated to observe the violation of the CPMS assumption (i.e., $\neg$**A.C**). In this scenario, as FPS begins to override the driver's intent to run away, the assumption that the engine is engaged according to the angle of the gas pedal no longer holds, resulting in the violation of **A.C** (and thus, its robustness value dropping below 0). Note the FPS guarantee **G.F** becomes satisfied only when the driver finally releases the gas pedal (around time 8)—thus the delay between the violation of **A.C** and the satisfaction of **G.F**.

In the other scenario where FPS is disabled, the CPMS monitor did not detect any violation, as expected. Throughout our experiments, however, our monitor also detected violations of the guarantee for CPMS, due to implementation errors in our Simulink model of CPMS. Our experience demonstrates the usefulness of runtime monitoring for both detecting unexpected interferences (Type 2 in Section 1) and debugging implementations (Type 1 in Section 1).

Performance. A runtime monitor can potentially introduce timing delays between the application being monitored and the rest of the system; in safety-critical systems like vehicles, such delays may not be acceptable. To evaluate the level of overhead incurred by our monitoring approach, we (1) measured the additional simulation time introduced by the use of the monitor for our scenarios and (2) performed a stress test of the MQTT-based communication architecture. For (1), we used a built-in MATLAB function to measure the overall simulation time consumed by Simulink with and without the runtime monitor activated. For scenarios of length 30 seconds (in virtual time), Simulink took approximately 1.871 CPU seconds to complete the simulation with the monitor, while it consumed 0.420 seconds without it, resulting in around 4.5 times overhead introduced by the monitor. While this overhead may appear significant, we believe that this is in part due to the performance characteristics of the Simulink environment. In practice, for integration into an actual vehicle, we envision using a dedicated Electronic Control Unit (ECU) or hardware device that runs in parallel with the target component and periodically monitors the input and output signals. Nevertheless, further research is warranted to optimize the performance of online STL monitoring and generate efficient monitoring code that can be deployed real-time.

For (2), we setup a local-area network (LAN) with 200 nodes (100 subscribers and 100 publishers), and measured the throughput and transmission delay. In total, each node was scripted to send 10,000 messages using the Mosquitto publish-subscribe API, with each message being 1024 bytes in size. On average, the overall throughput was approximately 5220 messages per second (delivered), with the average transmission delay of 0.5 seconds per message. Our evaluation suggests that an MQTT-based architecture is suitable for testing and simulation, as it provides a similar (if not better) level of performance as the Controller Area Network (CAN) bus and thus, can be used to emulate an in-vehicle network with high fidelity. However, a further investigation is needed to evaluate whether the same architecture can be deployed inside a vehicle, as hardware capabilities and constraints of in-vehicle components (e.g., ECUs) may be different from those that were used in our simulation environment.

5 RELATED WORK

Runtime verification [8] is an active area of research on techniques and tools (including monitoring) for reasoning about behaviors dynamically during the execution of a system. Runtime monitoring has been applied to automotive systems [9, 11, 20], but has not been used to detect violations of an environmental assumption due to an unexpected interference from other components.

The problem of *feature interaction*—which arises when two or more applications interact in subtle ways that lead to a violation of a desired system property—has been studied in the context of the automotive domain [2, 5, 13]. As far as we know, our approach is the first to leverage STL formulas in order to detect interactions among features that result in assumption violations.

6 CHALLENGES AND FUTURE DIRECTIONS

Based on our experience applying the state-of-the-art monitoring tool to an automotive case study, we believe that runtime monitoring is a promising approach for providing an additional layer of assurance in connected vehicles. Here, we summarize the challenges of building and applying the monitoring framework from the perspective of an OEM, and suggest potential research directions that we believe are crucial for making runtime monitoring more accessible to automotive engineers.

One of the authors is an automotive engineer with experiences in controller design but without a prior exposure to formal specification and verification. It took the author approximately 1.5 months to study and become comfortable with STL. While we believe that the potential benefits from runtime monitoring are worth the effort of acquiring this knowledge, better language support could be developed to facilitate the encoding and reuse of specifications for automotive engineers (for example, by providing a collection of common specification patterns or templates in automotive use cases, similar to LTL property patterns in [7]).

Design is often an iterative process, beginning with an initial abstract design that focuses on a discrete high-level protocol or logic, and eventually arriving at a more concrete design that contains details about signals and timing. A tool or framework that supports this iterative process by allowing the engineer to specify design models using both LTL (for abstract design) and STL (detailed design), and perform an analysis that the concrete design conforms to the abstract one would be highly beneficial.

The decomposition of an end-to-end system requirement into individual component specifications is another challenging activity that could greatly benefit from automation. While various theories for specification decomposition have been studied [3, 18], there is a need for mature tools that seamlessly integrate and carry out the decomposition task as part of a design process.

During the system deployment, once a violation of an assumption is detected, further actions may be performed to minimize the possible consequences of the violation, either by disabling the problematic application or overriding the existing requests with one that will ensure the safety of the vehicle. Mechanisms for the *enforcement* of safety in the presence of possible assumption violations are another important problem that we believe will play an important role in safe automotive design.

Finally, seemingly auxiliary aspects of a tool, such as ease of installation, tutorials, well-designed UI, and documentation, are crucial for making these types of system verification techniques more approachable to engineers.

ACKNOWLEDGMENTS

The authors would like to thank Omar Hussien, Santhana Raghavan, and Zhihao Zhang for their valuable inputs.

REFERENCES

[1] Alberto Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, and Kim Guldstrand Larsen. 2015. *Contracts for Systems Design: Theory.* Research Report RR-8759. INRIA.
[2] Cecylia Bocovich and Joanne M. Atlee. 2014. Variable-specific resolutions for feature interactions. In *International Symposium on Foundations of Software Engineering (FSE)*. 553–563.
[3] Chris Chilton, Bengt Jonsson, and Marta Kwiatkowska. 2014. Compositional assume-guarantee reasoning for input/output component theories. *Science of Computer Programming* 91 (2014), 115–137.
[4] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. 2017. Robust online monitoring of signal temporal logic. *Formal Methods in System Design* 51, 1 (2017), 5–30.
[5] Alma L. Juarez Dominguez, Nancy A. Day, and Jeffrey J. Joyce. 2008. Modelling feature interactions in the automotive domain. In *International Workshop on Modeling in Software Engineering (MiSE)*. 45–50.
[6] Alexandre Donzé. 2010. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *Computer Aided Verification (CAV '10)*. Springer Berlin Heidelberg, 167–170.
[7] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *International Conference on Software Engineering (ICSE)*. 411–420.
[8] Klaus Havelund and Grigore Rosu. 2002. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 342–356.
[9] Donal Heffernan and Ciaran MacNamee. 2016. Runtime observation of functional safety properties in an automotive control network. *Journal of Systems Architecture - Embedded Systems Design* 68 (2016), 38–50.
[10] Austin Jones, Zhaodan Kong, and Calin Belta. 2014. Anomaly detection in cyber-physical systems: A formal methods approach. In *IEEE Conference on Decision and Control (CDC)*. 848–853.
[11] Aaron Kane, Omar Chowdhury, Anupam Datta, and Philip Koopman. 2015. A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System. In *Runtime Verification (RV)*. 102–117.
[12] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems.* Springer Berlin Heidelberg, 152–166.
[13] Andreas Metzger. 2004. Feature interactions in embedded control systems. *Computer Networks* 45, 5 (2004), 625–644.
[14] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (1992), 40–51.
[15] Pierluigi Nuzzo, John B. Finn, Antonio Iannopollo, and Alberto L. Sangiovanni-Vincentelli. 2014. Contract-based design of control protocols for safety-critical cyber-physical systems. In *Design, Automation & Test in Europe (DATE)*. 1–4.
[16] OASIS. 2015. MQTT Version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. (2015). Accessed: 2018-03-11.
[17] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Symposium on Foundations of Computer Science (SFCS '77)*. 46–57.
[18] Jean-Baptiste Raclet. 2008. Residual for Component Specifications. *Electr. Notes Theor. Comput. Sci.* 215 (2008), 93–110.
[19] Rakesh Rana, Miroslaw Staron, Christian Berger, Jörgen Hansson, Martin Nilsson, and Fredrik Törner. 2013. Increasing Efficiency of ISO 26262 Verification and Validation by Combining Fault Injection and Mutation Testing with Model Based Development. In *International Conference on Software Technologies*. 251–257.
[20] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, and Radu Grosu. 2016. Applying Runtime Monitoring for Automotive Electronic Development. In *Runtime Verification (RV)*. 462–469.