Preprint: Dependable Systems and Networks 2014

# Monitor Based Oracles for Cyber-Physical System Testing

Practical Experience Report

Aaron Kane
Carnegie Mellon University
Pittsburgh, PA USA
akane@cmu.edu

Thomas Fuhrman
General Motors
Warren, MI USA
thomas.e.fuhrman@gm.com

Philip Koopman
Carnegie Mellon University
Pittsburgh, PA USA
koopman@cmu.edu

***Abstract*—Testing Cyber-Physical Systems is becoming increasingly challenging as they incorporate advanced autonomy features. We investigate using an external runtime monitor as a partial test oracle to detect violations of critical system behavioral requirements on an automotive development platform. Despite limited source code access and using only existing network messages, we were able to monitor a hardware-in-the-loop vehicle simulator and analyze prototype vehicle log data to detect violations of high-level critical properties. Interface robustness testing was useful to further exercise the monitors. Beyond demonstrating feasibility, the experience emphasized a number of remaining research challenges, including: approximating system intent based on limited system state observability, how to best balance the simplicity and expressiveness of the specification language used to define monitored properties, how to warm up monitoring of system variable state after mode change discontinuities, and managing the differences between simulation and real vehicles when conducting such tests.**

## I. Introduction

Modern automobiles are becoming increasingly complex with the addition of advanced connectivity and autonomy features. Many of these new features have substantial control authority over vehicle motion, and are thus safety-critical. Although full formal verification of such systems is still out of reach, formal methods may be of some help in performing more thorough system testing. This work describes our experiences exploring the feasibility of using a "bolt-on" external passive runtime monitor to improve the results of system testing on a prototype vehicle design.

A major challenge when testing complex critical systems, especially distributed Cyber-Physical Systems (CPSs), is creating an automated or semi-automated method to evaluate the results of system testing. This is essentially the testing oracle problem [18].

Runtime verification [13] is a lightweight formal method that attempts to verify that execution traces (whether at runtime or offline from logs) conform to a given specification. A (runtime) monitor is a device that that reads a system trace and yields a verdict about whether the trace satisfies some target property. Although due to time and complexity constraints of the experiments all the monitoring in this work was performed offline (on stored log data), we use the term *runtime monitor* in this work. There is no fundamental reason the monitoring could not be done at runtime, but offline verification was more useful at this stage of work since it is more flexible to changes and system access restrictions. It also permits running multiple experiments on identical system traces, which would be impractical with live vehicle testing.

Runtime verification can be used to provide a partial oracle to ensure that critical system properties hold during testing. While creating a formal specification that exactly describes the runtime behavior of a system as complex as an automobile is impractical, a specification that approximately bounds safe behavior and is amenable to runtime verification might be attainable.

Runtime monitors designed for testing safety-critical systems have additional constraints beyond traditional runtime monitors. A significant consideration is that the monitor must be isolated from the target system to minimize (or, ideally, eliminate) any disruption of the system under test, especially with regard to real time performance. Isolation is important because if the monitor is not isolated, it must be designed to at least the same level of safety integrity [8] as the system it is monitoring, increasing development cost and requiring that the monitor itself be deployed with the system. While that approach might be desirable in some situations, our work considers the common desire to have a "bolt-on" testing box that can be inserted into an existing safety-critical distributed system to improve the ability to analyze system-level testing results and then removed when deploying the final system without invalidating the test results with regard to system safety. Ultimately this will require a formalized argument that the interface between the monitor and target system is truly passive and non-interfering. This could be created using a system model with wormholes [17] or through a rigorous safety case argument [1].

Most runtime monitoring work to date has been in creating integrated monitors that are not designed with isolation in mind [7]. Moreover, most runtime monitoring work has assumed access to source code, which is often not true of components integrated into commercial systems. Thus, we expected that there would be significant practical issues in creating an isolated monitor. For example, there is the question as to whether enough visibility into system operation is likely to be available on an existing CPS embedded network without having to modify the system design to make monitoring viable.

The primary question we address is: given the constraints inherent in such an approach, can a bolt-on testing monitor be useful? To understand what happens when attempting to build an external runtime safety monitor for use as a testing oracle, we implemented a prototype runtime monitor on a hardware-in-the-loop (HIL) vehicle simulator for a prototype system provided by an automobile manufacturer. Besides passively checking system test traces during normal operation, we also performed robustness testing to better exercise the monitor. Additionally, we compared results to passive monitoring on logs from an operational prototype test vehicle. In this paper we discuss our results on the feasibility of and remaining challenges for creating scalable runtime monitors of this type.

## II. Background

Test oracles are functions that identify whether a given test has succeeded or failed. In traditional testing, human users act as test oracles, sometimes aided by automatic tools. Automated oracles can provide more accurate (no mistakes) checking of test results at a faster rate than manual checking, if they can be designed to accurately predict correct system behavior in response to test stimuli. The *oracle problem* is how to create such an automated predictor of system responses, including addressing situations in which such a predictor is impractical [18].

Partial oracles, which are oracles that can sometimes – but not always – correctly decide whether a test has succeeded or failed, can be simpler to identify. For this work we seek to create test oracles that are partial in two respects. First, the oracles only describe critical system properties rather than all system behaviors. In particular, they only attempt to describe properties that correspond to system safety. Second, the oracles only provide approximate bounds to safety rather than attempting to specify exact safety invariants. For this work, such oracles are deemed useful if they discover problems with the system that the designers did not discover using their traditional testing techniques.

While typical runtime monitoring frameworks might be used as partial testing oracles, they are largely targeted at pure software systems rather than CPS applications. Goodloe and Pike present a thorough survey of monitoring for distributed real-time systems in [7]. Those runtime monitors that are intended for CPS applications tend to assume that the system under test must be augmented with instrumentation, compromising isolation. For example, Copilot [15] generates constant-time and constant-space on-chip monitors so that the added overhead is known and bounded. But even so, the system under test must be modified to accommodate monitoring. Moreover, many existing monitors require access to the underlying source code, which is often unavailable in commercial systems.

At least one existing monitor framework, BusMOP [14], generates external bus monitors from high level specifications targeting commercial off-the-shelf peripherals (specifically, the PCI-X bus). BusMOP avoids some observability issues by limiting specification properties to bus-visible accesses (memory, I/O, and interrupts). However, there has been no full-scale experiment on a realistic system demonstrating that safety monitoring of a black-box CPS with no added instrumentation is feasible and useful.

## III. Test System

Automobiles are a straightforward target for a passive network monitor since they are highly distributed systems with a broadcast bus (usually CAN [2]). Automotive networks tend to periodically broadcast system state messages, enabling a monitor to obtain an incomplete, but useful, view of the state of the system without additional instrumentation and without adding new message traffic. We use an external, passive bus monitor which minimizes the intrusiveness of the monitor on the target system. Our monitor checks properties written in a specification language containing a simplified bounded temporal logic loosely based on MTL [10] and state machine descriptions used to encode mode-based state. The logic contains the usual boolean connectives, arithmetic comparisons, and two bounded temporal operators (always and eventually). To ensure non-interference and avoid issues related to performance (which are left to future work), the monitoring in this work was done offline on system logs captured from the target system.

The system under test was a dSPACE hardware-in-the-loop (HIL) simulator testbench for an automobile manufacturer. The simulator uses MATLAB SIMULINK models to generate the code for individual electronic control units (ECUs). CARSIM [3] is used to provide the simulated vehicle and environment which the feature models on the HIL operate within.

The dSPACE HIL uses the dSPACE ControlDesk interface software to manage loading models and running tests (including calibration, logging/measurements, and diagnostic access). ControlDesk includes a library (rtplib) allowing real-time scripting access to the models running on the HIL. We used this library to create some robustness testing scripts, and additionally used ControlDesk's control panel functionality to control manual injection of some individual signals. All logging was performed with ControlDesk's trace capture functionality.

The vehicle tested was a prototype development platform for semi-autonomous driving features, including Full Speed Range Adaptive Cruise Control (FSRACC), automated lane keeping, and emergency collision avoidance. Because feature development was still in progress, the only feature we were able to test was a third-party supplied FSRACC. The FSRACC was not hardened for robustness, and therefore our findings of robustness issues do not reflect upon the quality of production-grade features. However, the FSRACC did provide a prototype-quality realistic automotive feature for our testing purposes.

To facilitate the black box interception and injection of vehicle network messages for robustness testing purposes, we added some instrumentation to the vehicle feature model. (These modifications were solely for input interception/injection to elicit system failures, and did not provide instrumentation for runtime monitoring. Moreover, they did not involve modification of the FSRACC code itself.) Each input signal to the FSRACC module was routed through an added multiplexor with a inject signal value controlled by an enable signal. This allowed us to have each input signal individually passed-through or overwritten by the chosen injection signal. These additional signals (the injection and enable signals) were accessible through ControlDesk's layouts and through Python scripts using the included rtplib as if they were a part of the original feature model. Because this part of the system was a simulation running in a fast HIL computer, the modifications did not affect system timing.

Although we added instrumentation for robustness testing, we note that the monitor itself only requires access to network messages that are already available on the vehicle's CAN broadcast network. This means that the monitor could be used on a real vehicle without requiring any instrumentation (beyond connecting it to the network or otherwise exporting network logs). To validate the simulation, we also ran the monitor on logs from an actual vehicle running similar code, and caught some of the same violations on the real vehicle as found on the simulator during normal (no signal injection) operation.

### A. *Robustness Testing*

In this work the primary goal of robustness testing was to increase the chances of seeing system faults during testing to better exercise the monitor, and not to characterize the quality of prototype vehicle software, which was expected to be non-robust. We used the existing ControlDesk interface to perform network value interception and injection. The injected values were limited by data-type bounds checking performed by the interface. This limited injection target's signal values to floats (including exceptional values e.g., NaN, infinity), booleans (true/false), and enumerations (positive integer values).

We performed three different classes of robustness testing on the HIL: random bit flips (one, two, and four bits), random value injections, and exceptional value injections. For each testing type we injected a particular number of faults each for $20s$ (to allow time for the fault to manifest into a specification violation). Bits to flip were randomly chosen for each individual bit flip fault. For random value injection we injected values from $[-2000, 2000]$ for floats, $[0, 1]$ for booleans, and $[0, maxint]$ for enums. The float range was chosen such that it would go beyond the possible non-faulty values of the target messages while keeping the range small enough that at least some values chosen would land in the value's normal range. We used Ballista [9] style exceptional value injection which targeted float-typed messages with values chosen from the set $\{$NaN, $\infty$, $-\infty$, 0.0, -0.0, 1.0, -1.0, $\pi$, $\frac{\pi}{2}$ $\frac{\pi}{4}$, $2\pi$, $e$, $\frac{e}{2}$, $\frac{e}{4}$, $\sqrt{2}$, $\frac{\sqrt{2}}{2}$, $ln(2)$, $\frac{ln(2)}{2}$, 4294967296.000001, 4294967295.9999995, 4.9406564584124654e-324, -4.9406564584124654e-324$\}$. Random valid value injection values were used for exceptional-input injection targeting non-float data types due to the strong value checking enforced on the HIL testbed.

| I/O | Name | Type |
|---|---|---|
| Input | `Velocity` | float |
| Input | `AccelPedPos` | float |
| Input | `BrakePedPres` | float |
| Input | `ACCSetSpeed` | float |
| Input | `ThrotPos` | float |
| Input | `VehicleAhead` | boolean |
| Input | `TargetRange` | float |
| Input | `TargetRelVel` | float |
| Input | `SelHeadway` | float |
| Output | `ACCEnabled` | boolean |
| Output | `BrakeRequested` | boolean |
| Output | `TorqueRequested` | boolean |
| Output | `RequestedTorque` | float |
| Output | `RequestedDecel` | float |
| Output | `ServiceACC` | boolean |

Fig. 1. FSRACC Module IO Signals

We performed script-based injection both on each target message individually and as well as some injections against multiple messages at once. We also performed manual exploration of identified faults by creating a ControlDesk Layout with numeric input boxes providing manual control of the injection framework.

### B. *The Feature Under Test*

We performed black box testing of the FSRACC feature. Source code was not available for the feature, so all testing and specifications were based on external interfaces and understanding of the high level behavior.

The inputs and outputs of interest to the FSRACC module are listed in Figure 1. The module has other inputs and outputs that were disregarded for testing because they had no observable effect, immediately cancelled cruise control, were interface indicators, or otherwise did not affect vehicle safety.

The `Velocity` input message is the forward speed of the vehicle. `AccelPedPos` gives the position of the accelerator pedal as a percent (0 being fully released, 100 fully depressed). The pressure applied to the brake pedal is given in `BrakePedPres` and the position of the throttle as a percentage (i.e., how open is the throttle) is `ThrotPos`. The commanded cruising speed is sent in the `ACCSetSpeed` message. The `VehicleAhead` message tells the ACC module whether a vehicle is detected ahead of it in the lane. `TargetRange` and `TargetRelVel` are the distance between the vehicle and the vehicle ahead of it (if one exists) and the relative velocity between those two vehicles respectively. The selected headway distance to the preceding car is an enum `SelHeadway`.

The output `ACCEnable` is whether the ACC thinks it is supposed to be in control of the vehicle (i.e., engine and brake controllers should ignore these output values if ACC isn't enabled). The `BrakeRequested` output is true when the ACC feature is requesting a deceleration. If the `BrakeRequested` output is true, then `RequestedDecel` is a requested deceleration in $m/s^2$ for the brake controller to attempt to provide. If instead the message `TorqueRequested` is true then the `RequestedTorque` output is the additional amount of torque the engine controller should attempt to provide. The `ServiceACC` message is an error message used to enable an interface indicator to alert the driver that the feature has detected an error.

### C. Safety Specification

To evaluate the use of these techniques we created partial behavioral specifications that were motivated by ensuring system safety. We used six safety rules that checked a mix of system robustness and functionality.

Since the feature under test is a third party provided code module designed mainly as a placeholder function to support early system integration, there was no available specification. While it would be ideal to have a system specification from which to derive monitoring rules, that was not the case for the available systems we had to test. Instead we created a set of specification rules based on "expert" elicited common sense (i.e., properties a knowledgable engineer could expect to hold based on automotive domain experience) through discussions with the system's engineers and looking over existing system metrics and other potentially related documentation. While we would have preferred to have rules directly derived from system documentation, this is not always possible in industry (as in this example). In cases like this the usefulness of the monitoring results depend heavily on the experts and the quality of the rules they choose. Though expert derived rules may not provide as clear a notion of monitoring coverage, they can be made with the expert's direct needs in mind. For example, while the rules we check in this work are in no way complete, they would be high priority (likely leading to vehicle collisions) for a production quality feature. The six rules we checked against the robustness testing traces were:

**Rule #0** If the `ServiceACC` signal is true, then `ACCEnabled` must be false.
*A simple consistency check to ensure that the feature does not continue to attempt to control the vehicle when it knows something is wrong.*

**Rule #1** If the actual vehicle headway time is below 1.0s, then it must be recovered to greater than 1.0s within 5s elapsed time.
*This rule is derived from an existing headway metric for another similar system*

**Rule #2** If `TargetRange` is less than half the desired headway, then `RequestedTorque` should not be increasing.
*Check for feature trying to increase speed when it is already too close to the target vehicle*

**Rule #3** If `Velocity` is greater than `ACCSetSpeed` and `RequestedTorque` is less than 0, `RequestedTorque` must still be less than 0 in the next timestep.
*Check for vehicle attempting to increase speed when already above the set speed, avoiding tripping on control oscillations by only checking after there are no active requests.*

**Rule #4** If `Velocity` is greater than `ACCSetSpeed` then `RequestedTorque` must not be increasing some point within 400ms.
*Similar to #3: if vehicle velocity is increasing while above set speed, should start slowing down (or at least hold speed) within 400ms*

**Rule #5** If `BrakeRequested` is true then `RequestedDecel` must be less than or equal to 0.
*Checks if the value of a requested deceleration is in fact a deceleration (negative).*

**Rule #6** If `VehicleAhead` is true and `TargetRange` is less than 1, then `TorqueRequest` must be false or `RequestedTorque` must be less than 0.
*Checks for near collisions – assuming a feature should not be requesting a increase in speed when the target vehicle is extremely close.*

Because these rules were picked without any knowledge of the internal control algorithms or design parameters of the system, some of them may be too strict (this turned out to be the case as we shall see). It seems likely that this sort of approach would be common when applying runtime monitoring to real-world systems, which often have incomplete specifications and opaque internal operation. Thus, the approach we took was to adopt these rules and then relax them when false positives and uninteresting violations were found. We think this is a reasonable approach to employing runtime monitors in practice.

The issue of whether the data required to implement the monitor would be observable was simple since we were able to create our rules with knowledge of this restriction (and thus write rules based on system state available on the CAN bus). Observability would be a more difficult issue when deriving rules from system requirements which may include requirements on properties that are not externally observable. We discuss this further in Section V-D.

Coverage of the safety rules is not intended to be complete. Rather, the idea is to express a set of safety rules that are useful and see if runtime monitoring detects rule violations for a black-box system which has not been augmented with additional monitoring information.

## IV. Testing Results

For each of the eight target signals we ran three tests – one Ballista-style injection, one bit flip test in which one, two, and four bit bit-flips were injected, and one random value injection. Random and Ballista testing included eight different injection values per test and bit-flips included four injections for each bit-flip size (with all injections held for $20s$). We also ran eight tests of 20 injection values on multiple target signals at once (e.g., `TargetRange`, `VehAhead`, and `TargetRelVel` at the same time). Testing time was limited by the physical time to run tests on real automotive hardware setups. Statistical analysis of robustness testing techniques was not a goal of this work. The number of tests were sufficient to demonstrate that monitoring detected problems under robustness fault injection and indicated a lack of problems (to the degree possible given available data) in non-faulted operation.

The results of the robustness testing identified many specification violations. The testing results are summarized in Table I. An "S" represents a rule satisfied by the given trace, while a "V" represents a violated rule. The mBallista, mRandom, and mBitFlip entries are tests where more than one message was targeted at once. "Range+" injected `TargetRange`, `TargetRelVel` and `VehAhead`. "Range+Set" also included `SetSpeed`, and "All" was all 9 FSRACC inputs. Six out of the seven rules were detected as violated during testing (all except Rule #0). Many of the violations could be caused, and were detected, by multiple test runs (i.e. different signals being targeted or different types of injections to the same signal).

All three types of robustness testing found similar robustness problems in the system under test. This is not an unreasonable outcome, because all three fault classes easily exercised out-of-range faults that caused most of the identified violations.

A major identified cause of problems was the lack of input checking in the feature. The `Velocity`, `TargetRange`, `TargetRelVel`, and `ACCSetSpeed` messages all have direct and strong effects on the control output, but are neither bounds checked (for exceptional inputs) nor consistency checked against each other or other inputs. This makes them vulnerable to a bad input value causing the control algorithm to command an unsafe output. For example, an exceptional `TargetRange` value when following a target causes the ACC feature to command the vehicle to accelerate into (and through, because the simulator doesn't check collisions) the target vehicle. This is an obviously dangerous situation (the ACC feature causing a crash during target following). The same problem can be caused by an incorrectly negative relative velocity, which is a situation caught by Rule #6. The feature does have enough information to protect against these two signals being inconsistent and causing a failure by checking the consistency between the change of `TargetRange` and `TargetRelVel`. It just doesn't do the checking.

TABLE I
Fault Injection Results

| | | Specification Rule | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Injection** | **Target Signal** | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| Random | Velocity | S | V | S | V | S | S | V |
| Random | TargetRange | S | S | V | S | V | S | V |
| Random | TargetRelVel | S | V | S | S | S | S | V |
| Random | ACCSetSpeed | S | V | S | V | S | S | V |
| Random | ThrotPos | S | S | S | S | S | S | S |
| Random | AccelPedPos | S | S | S | S | S | S | S |
| Random | BrakePedPos | S | S | S | S | S | S | S |
| Random | SelHeadway | S | S | S | S | S | S | S |
| Ballista | Velocity | S | S | V | S | S | V | V |
| Ballista | TargetRange | S | V | S | S | S | V | V |
| Ballista | TargetRelVel | S | V | S | S | S | S | V |
| Ballista | ACCSetSpeed | S | S | V | V | V | S | S |
| Ballista | ThrotPos | S | S | S | S | S | S | S |
| Ballista | AccelPedPos | S | S | S | S | S | S | S |
| Ballista | BrakePedPos | S | S | S | S | S | S | S |
| Ballista | SelHeadway | S | S | S | S | S | S | S |
| Bitflips | Velocity | S | V | V | S | V | V | V |
| Bitflips | TargetRange | S | V | S | S | S | V | V |
| Bitflips | TargetRelVel | S | V | S | S | S | V | V |
| Bitflips | ACCSetSpeed | S | V | S | S | S | V | V |
| Bitflips | ThrotPos | S | S | S | S | S | S | S |
| Bitflips | AccelPedPos | S | S | S | S | S | S | S |
| Bitflips | BrakePedPos | S | S | S | S | S | S | S |
| Bitflips | SelHeadway | S | S | S | S | S | S | S |
| mBallista | Range+ | S | V | S | S | V | V | V |
| mBallista | All | S | V | S | S | S | S | S |
| mRandom | Range+ | S | V | V | S | V | V | S |
| mRandom | All | S | V | S | S | S | V | S |
| mRandom | Range+Set | S | V | S | S | S | V | S |
| mBitflip1 | Range+ | S | V | S | S | S | V | V |
| mBitflip2 | Range+ | S | V | V | V | V | V | V |
| mBitflip4 | Range+ | S | V | S | S | S | V | S |

Some violations turned out to be overly strict rules, but sometimes a violation that is primarily a too-strict rule can also detect a valid transient violation. Most violations of Rule #5 were due to control system overshoot from negative to a single-cycle positive acceleration when brakes were released, which might be considered acceptable. But there were also transient violations of this rule caused by an injected fault turning the ACC feature on that caused a one cycle blip of positive `RequestedDecel`. While one cycle of bad requested deceleration may be tolerated in an operational vehicle, it is worth noting such anomalies in test data, because they can provide a clue to a potential latent bug that will have more severe effects in difficult-to-test corner cases. Such violations can be hard to detect without a tool such as a runtime monitor, since the vehicle may appear to behave properly from a driver's point of view during these fleeting problems.

### A. Real Vehicle Logs

We also analyzed log data from a prototype vehicle that implemented the functions in the HIL simulation. These logs were of normal operation, not robustness testing, covering a couple hours of vehicle operation for representative driving scenarios. Data was limited due to the experiments being of lower priority than vehicle development work on the limited resource of a single available test vehicle at an industry partner facility. No safety problems were detected in the vehicle logs we had available. However, results did correspond to observations made on non-faulty HIL test results.

The same rules checked on the simulator were checked against the real vehicle logs, and similar system dynamics were found. Rules #0, 1, 5 and 6 were not violated in the vehicle logs. Rules #2, 3, and 4 had some violations, but upon further examination they were determined to be reasonable violations (i.e., overly strict rules). Rule #2 does not gracefully handle small headway gaps and acceleration that can occur during overtaking (passing) or a vehicle cutting in, and Rules #3 and #4 are not fair to the real dynamics of the system where torque request increases do not necessarily imply system intent (e.g., starting up a hill torque must increase to maintain constant vehicle speed). The identified violations included negligibly sized increases as well as extremely short transient increases. These results lead to our identifying intent approximation as a challenge, as discussed below.

## V. Discussion

In the process of building the monitor, writing the specification, and performing testing we ran into issues that on further inspection lead to deeper research questions. We identified three major research challenges:

**Intent Approximation:** How do we approximate or represent a desired system intent based on the available observable system properties?

**Specification Languages:** What language features are necessary for efficiently specifying the desired system properties?

**System Mapping:** How do we map the target system onto the monitor's assumed model of the system?

We also discuss how generalizable we believe these insights are to other systems and some issues regarding the passive observability of a target system.

### A. Intent Approximation

The viability of a black box, external runtime monitor rests upon the assumption that some portion of the desired specification can be directly checked from the observable state of the system. The obvious concern is that not enough data such as vehicle speed or steering commands will be available for monitoring. As it turns out, automotive networks have plenty of such data available due to their use of a distributed system architecture that broadcasts such data on a CAN bus. However, a more subtle problem with observability emerged in the course of doing this work in the form of the *intent estimation* problem. This is the question of how to represent a feature's intent to perform some high-level action based on some set of lower level properties [6]. This problem has been explored in many areas including defense aerospace [11], unmanned undersea vehicles [5] and automobile driver intent [12].

We would expect the intent estimation problem to be easier when performing white-box rather than black box testing, since being able to directly see how a system's input affects its output would help understand system intent. But since source code doesn't necessarily explicitly encode intent it is unclear how much system access affects intent estimation.

In these experiments we used an increase in FSRACC requested torque as an estimation for the FSRACC intending to accelerate the vehicle. While this is a somewhat causal relationship (increasing engine torque should generally increase vehicle speed), torque requests depend upon a host of factors such as road conditions and grade, and can be differentiated by factors such as duration and amplitude of the increase. Based on this example, we expect that designers who wish to employ an external monitor will face a tradeoff between carefully architecting selected internal system information that reveals intent vs. building somewhat more complicated monitors that decipher intent based on observable information. (It is easy to say that intent should always be broadcast, but that may not be feasible for system integrators purchasing off-the-shelf components from multiple outside vendors.)

In order to be able to tune these approximations, designers must be able to evaluate a given violation and decide whether the violation was real or not. This may be non-trivial on some systems, especially if a part of the reason for the use of a monitor is to help developers understand the test traces. In our case, we took into account the intensity and duration of the violations, as well as the apparent cause to make a decision on whether a violation was a safety problem or not.

A monitor used as a test oracle for safety rules can provide evidence for a system safety case [1] that the system did in fact pass the executed tests. For this type of use we want to have no false negatives (i.e. the estimation catches every violation of the intended high level rule) to allow the testing results to be used as strong evidence. If having no false negatives is impossible or it results in an unmanageable amount of false positives, using an intent estimation that reports some false negatives is still more useful as part of a comprehensive safety approach than not checking at all. (Detecting even a single safety violation provides useful evidence that the system is unsafe.)

### B. Specification Languages

We have designed our monitor to use a simplified temporal logic combined with state machines. Most existing monitor technologies use some form of logic [4] or a domain specific language that matches the implementation [16]. Logic-like languages are promising, but some researchers use a form of state machine to encode modal system state or to reduce the complexity of temporal operators in logic. For example, we avoid nesting of temporal operators by using state machines when needed, and that proved useful in this work.

The trade-off between simplicity and expressiveness in monitor logic is important because it affects the efficiency of the monitor, and the ultimate goal for this type of system is to operate with the system being monitored in real time. It is not yet clear what degree of expressiveness is required to monitor typical safety properties on real systems. The specification rules that we used in this work are relatively simple, yet they did identify system faults under robustness testing. So it may be that relatively simple logic languages which only provide a subset of the usual temporal logic functions suffice for runtime CPS monitoring. There are additional complexity trade-offs between the specification notation and in the system-to-monitor mapping (discussed below).

### C. Monitor Rule to System Mapping

A major challenge is that of mapping the real system to an abstracted model at run-time that provides the system state information for the monitor to check. There are many different existing monitoring techniques that each have their own unique model of a system. Here we generally discuss issues related to external monitors such as ours. Inline monitors (i.e. monitors that exist within the system code, which are beyond the scope of this work) may have fewer mapping issues because they are more directly integrated into the system, but it would be no surprise if they have similar tradeoffs.

Our monitor is designed around the use of a set of network messages representing system state.

*1) Multiple Sampling Periods:* In the vehicle we tested there are two relevant message periods, with some messages being updated four times slower than most others. At first we simply assumed that these slower values stayed constant between updates. But, dealing with values across multiple timesteps required more care, because a slowly sampled value that is in fact increasing would appear to be unchanging for several cycles while the faster samples were being checked.

As an example, to see if the FSRACC feature was requesting increasing torque, we would calculate the difference of the previous and current `RequestedTorque` value. However, if the held value is used in a monitor that updates four times between every `RequestedTorque` update, the torque would appear to be constant for three samples out of four due to the repetition of the most recent sampled value being held. Additionally, jitter would sometimes cause slower-period messages to be delayed, resulting in five faster frequency message updates occuring between the slower message updates. Once recognized, it was relatively simple to work around these problems in an ad hoc manner. But, the observation remains that runtime monitoring that involves data sampled at different periods can be tricky, and a runtime monitoring architecture should have a uniformly applied mechanism to deal with that issue.

*2) Discrete Value Jumps:* Another network value issue that we came across was ensuring that rules could handle message transitions from non-active to active. Some messages in a system, such as `TargetRange` can perform large discrete jumps when they are activated even though they represent continuous physical properties. As an example, `TargetRange` is 0 when there is no target being tracked, but once a target is found this value will immediately jump to the actual range. This was noticed for rules that checked if the ACC would command control when the change in `TargetRange` did not agree with sign of `TargetRelVel`. So while these values should always agree in a non-fault condition, there is one situation where they may not: when a vehicle comes into sensor view the relative velocity may be correctly reported as negative, but the first change in range seen is necessarily positive (change from zero to the actual positive range). Delaying the check of such a rule until after the activation (allowing the "change" variable to initialize before testing) avoids this problem.

Other message or rule types may also have initialization issues, such as rules that rely on an integrator or running average of a value. A general observation is that run-time monitors should have a uniform way of "warming up" monitors for data that changes state abruptly, especially when changing from invalid to valid, to avoid false alarms.

*3) System vs. Model:* Even though a HIL simulation is supposed to be of high fidelity, we found a significant difference in that the HIL platform performed strong type checking of fault-injected values, prohibiting things such as out-of-range enumerated values. This limited the amount of fault injection possible compared to what might be done on a complete vehicle (which we were not permitted to do robustness testing on). As a result, robustness testing of the HIL platform likely missed problems that would be expected to be present in the real system, which does not have such type checking. For this reason it can be important to do runtime monitoring on the actual vehicle even if HIL testing finds no problems, especially if robustness testing results are desired on the real vehicle.

### D. Observability and Generalizability

The most limiting factor for a passive monitor targeting a specific system is how much system state is observable passively (without invasive instrumentation). For our target application (modern automobiles) and other similar systems (autonomous ground vehicles) there is a useful though not complete set of observable state available to be passively monitored due to the prevailing architecture where system state is broadcast between distributed system nodes on a single or small number of bus networks. While there are likely some systems which require instrumentation to reveal system state, we expect that there is a non-negligible class of systems that have similar architectures to automobiles. This is likely since high criticality systems tend to be distributed or have visible communication between replicated components. If a system is distributed, then some amount of useful state will be observable as the distributed nodes must communicate their state to each other. Whether this communicated state is enough to allow monitoring useful system properties is left to be seen in other systems, but for automobiles it does appear to be the case.

Whether the insights and results we have seen here are generalizable to other systems hinges essentially on whether other systems can be monitored in a similar manner. While it is unclear how many other classes of systems lend themselves to passive monitoring, at worst the explained methods and insights are applicable to other autonomous ground systems which are becoming increasingly important. We expect that other types of systems are similar, though some may require different levels of instrumentation. Intent approximation and mapping systems onto their abstract models are problems that will exist in some degree for any practical formal verification techniques for the foreseeable future.

## VI. Conclusion

Automated testing of complex safety-critical systems is an important method to increase the number of tests that can be completed, but an automated analysis of such test results is not necessarily easy. Runtime monitors can be used to create partial test oracles for critical system properties, such as system-level safety rules. Isolated, external runtime modules are especially attractive, but pose some potential challenges.

We showed the feasibility of a bolt-on monitor by implementing a passive network monitor (using log analysis) to check a HIL simulated vehicle provided by an automobile manufacturer. Despite not having source code access and working only with existing network signals, a practical monitor was developed with a half-dozen rules that identified system faults under robustness testing of a HIL system. This illustrates the possibility of expanding the use of this approach in testing key properties of CPS designs.

The experience revealed a number of challenges remaining to further advance this approach, including: approximating system intent based on limited system state observability, how to best balance complexity vs. expressiveness and scope of the specification language used to define the monitored properties, how to warm up monitoring of system variable state after mode change discontinuities, and managing the differences between simulation and real vehicles when conducting such tests.

## Acknowledgment

This research was funded in part by General Motors through the GM-Carnegie Mellon Vehicular Information Technology Collaborative Research Lab.

## References

[1] Bishop, P., Bloomfield, R.: A methodology for safety case development. In: SAFETY-CRITICAL SYSTEMS SYMPOSIUM, BIRMINGHAM, UK, FEB 1998. Springer-Verlag, ISBN 3-540-76189-6 (1998),

[2] Bosch, R.: CAN specification version 2.0 (Sep 1991)

[3] Corporation, M.S.: Carsim overview (November 2013)

[4] Delgado, N., Gates, A., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. Software Engineering, IEEE Transactions on 30(12), 859 – 872 (dec 2004)

[5] Fong, E.H.L.: Maritime intent estimation and the detection of unknown obstacles. Master's thesis, MIT (2004), http://hdl.handle.net/1721.1/30279

[6] Foo, P.H., Ng, G.W., Ng, K.H., Yang, R.: Application of intent inference for surveillance and conformance monitoring to aid human cognition. In: Information Fusion, 2007 10th International Conference on. pp. 1–8 (2007)

[7] Goodloe, A., Pike, L.: Monitoring distributed real-time systems: a survey and future directions (NASA/CR-2010-216724) (July 2010), http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.4769\&amp;rep=rep1\&amp;type=pdf,

[8] ISO: ISO/DIS 26262 - Road vehicles – Functional safety. Tech. rep., Geneva, Switzerland (November 2011)

[9] Koopman, P., Devale, K., Devale, J.: Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project, pp. 201–226. John Wiley & Sons, Inc. (2008)

[10] Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2, 255–299 (October 1990),

[11] Lee, K., Lunas, J.: Hybrid model for intent estimation. In: Information Fusion, 2003. Proceedings of the Sixth International Conference of. vol. 2, pp. 1215–1222 (2003)

[12] Lefevre, S., Ibanez-Guzman, J., Laugier, C.: Context-based estimation of driver intent at road intersections. In: Computational Intelligence in Vehicles and Transportation Systems (CIVTS), 2011 IEEE Symposium on. pp. 67–72 (2011)

[13] Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming 78(5), 293 – 303 (2009), the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07)

[14] Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems. 2008 Real-Time Systems Symposium pp. 481–491 (Nov 2008),

[15] Pike, L.: Copilot: Monitoring embedded systems. Tech. Rep. NASA/CR-2012-217329, NASA Langley Research Center (January 2012), available at http://ntrs.nasa.gov/search.jsp?R=20120001989&hterms=pike+goodloe&qs=Ntx%3Dmode%2520matchallpartial%2520%26Ntk%3DAll%26N%3D0%26Ntt%3Dpike%2520goodloe

[16] Pike, L., Goodloe, A., Morisset, R.: Copilot: A Hard Real-Time Runtime Monitor. In: 1st International Conference on Runtime Verification. No. Rv (2010), http://www.cs.indiana.edu/\~{}lepike/pubs/pike-rv2010.pdf

[17] Veríssimo, P.E.: Travelling through wormholes: A new look at distributed systems models. SIGACT News 37(1), 66–81 (Mar 2006),

[18] Weyuker, E.J.: On testing non-testable programs. The Computer Journal 25(4), 465–470 (1982),