Supporting Synthetic Data-Driven Diagnosis through Automated Fault-Injection

Patrick E. Lanigan, Priya Narasimhan
Carnegie Mellon University
Electrical & Computer Engineering
planigan@ece.cmu.edu, priya@cs.cmu.edu

Thomas E. Fuhrman
General Motors Research & Development
Electrical & Controls Integration Lab
thomas.e.fuhrman@gm.com

Abstract

Given the lack of empirical data available from automotive serial-communication networks, an automated fault-injection environment can be used to create synthetic datasets for training and testing data-driven diagnosis algorithms. We use commercial fault-injection hardware with custom software to implement such an environment. A small pilot study using injected physical-layer faults shows promise in producing identifiable error-patterns.

1 Introduction

The automotive industry has become steadily more reliant on software-intensive distributed systems to implement advanced vehicle features. In fact, it has been estimated [7] that "up to 90% of all innovations are driven by electronics and software". It is further estimated [7] that 50-70% of an ECU's development costs come from software, with some vehicles having up to 70 ECUs. Overall, electronics and software can account for up to 40% of a vehicle's cost [7]. When problems in software-based systems are uncovered after a vehicle has gone to production, recall costs can rival development costs. For example, 2004 saw the recall of 680,000 Mercedes-Benz E-Class vehicles due to issues with the electronic brake-by-wire system [23].

A growing trend is toward features that assist the driver in maintaining safe control of the vehicle under a variety of conditions. Previously, such assistance has been provided *passively* in the form of information or warnings. These features are now being given increasing amounts of authority to control the vehicle's motion by *actively* supplementing the driver's inputs. The long term trend is towards fully autonomous operation [16, 21].

Because these systems are critical to ensuring the safe operation of the vehicle, they must be designed to tolerate faults and provide high levels of dependability. Typically, a systematic safety analysis is conducted during the design phase, to evaluate both the severity and likelihood of the consequences of possible faults. Formal verification methods are used to analyze system dependability. Fault-injection can play a complementary role in this analysis by providing an empirical way to study the system's dependability in the presence of faults and to analyze the system's fault-handling capabilities with respect to a particular fault model. This can aid in fault-removal and fault-forecasting [2]. The upcoming ISO 26262 standard for functional safety in automotive electronics highly recommends that fault-injection be included as part of the dependability analysis of critical systems [10].

Despite extensive design processes, emergent behavior will still appear at runtime in dependable automotive systems. Such behavior occurs due to unforeseen interactions and complexity between independently designed components. These interactions are not readily apparent to the system designers, and might not be captured by system models. Therefore, diagnostic approaches that rely solely on system models are unlikely to provide a satisfactory diagnosis when presented with emergent behavior. A data-driven diagnostic approach that analyzes system metrics as well as system models has the potential to provide a more accurate diagnostic output [12].

In order to support the development of data-driven diagnostic approaches, we built an automated fault-injection environment for FlexRay [6]. This environment combines off-the-shelf fault-injection hardware with custom software to allow a large number of *highly repeatable* experiments to be coordinated from a centralized host. It also enables data-logging from *each* node in the cluster, as opposed to relying on a single monitoring point.

We performed a pilot study to validate this fault-injection environment and to determine whether it is feasible to distinguish faults based on their manifestations. The results of this study show that faults do no necessarily manifest symmetrically in a linear bus topology. Furthermore, those manifestations produce identifiable error patterns.

2 Synthetic Data-Driven Diagnosis

The analytic techniques used for data-driven diagnosis vary [3, 4, 8, 11], but commonly require a large dataset for algorithm training and testing. This data typically comes in the form of metrics that are derived from various instrumentation points. These instrumentation points exist at the system-level as well as the component-level. Examples of potential instrumentation points are the status indicators exposed by the FlexRay Controller-Host Interface (CHI) (see section 3.2), hooks inserted into software components, and Operating System (OS) metrics such as context-switch rates. The data can then be analyzed to infer a correlation between the metrics and the system health. Ideally, this correlation leads to some actionable diagnostic output.

Usually, such metrics are gathered from deployed systems. This is problematic in automotive systems, because real-world failure data is scarce. The most advanced dependable automotive systems exist only as research prototypes, and therefore have not seen wide enough deployment to generate useful failure metrics. For the few systems that have been deployed, Original Equipment Manufacturers (OEMs) are understandably reluctant to release failure data for public scrutiny.

Therefore, we propose leveraging fault-injection to build a synthetic data-driven approach. Fault-injection is already recommended to be used in the dependability analysis of critical systems [10], so the impact on existing development processes should be minimal. Even so, there are many research challenges involved in developing such an approach. We discuss these research challenges elsewhere [12], but the entire endeavor rests on a few key propositions.

Proposition 1 *The errors induced by faults form identifiable patterns.*

Proposition 2 *The error patterns corresponding to faults are evident in, and can be derived from, system metrics.*

Proposition 3 *The error patterns derived from system metrics allow faults to be distinguished by type, persistence, etc.*

The remainder of this paper describes a study with dual purposes. The primary purpose is to determine whether our fault-injection environment is suitable for studying Propositions 1–3. The secondary purpose is to determine whether the propositions themselves have any credibility. The experimental apparatus (i.e., fault-injection environment) and process are described in Section 3. We specify the parameters of the pilot study in Section 4. Section 5 discusses the results of the study. A brief overview of related work is contained in Section 7. Section 8 concludes this paper.

3 An Automated Fault-Injection Environment for FlexRay

In order to study Propositions 1–3 — and, indeed, *any* propositions — the fault injection environment should provide *repeatability*, *controllability* and *observability*. We also require *automation* in order to allow unattended operation for extended periods of time.

Controllability. The experimenter should be able to define experimental parameters accurately, with respect to time (e.g., fault activation-trigger and duration), space (e.g., fault location) and value (e.g., fault type).

Repeatability. Experiments run under similar conditions with similar parameters should produce similar results.

Observability. The effect(s) — or lack thereof — of a fault should be readily apparent. *We do not assume that faults manifest symmetrically across nodes*. Therefore, *observations* (i.e., snapshots of instrumented data) must be made at each node and compared with respect to the time, space and value domains. In FlexRay, the *space* domain corresponds to the node identifier (ID); the *time* domain corresponds to the local view of global time (denoted by the current macrotick and cycle counter); and the *value* domain corresponds to the measured data itself.

Automation. While a manual fault-injection process [13] can be useful for rapid-prototyping applications, such a process becomes unwieldy when a large number of experiments are required. In order to develop a robust dataset for training and testing different diagnosis algorithms, we need to experiment with a wide range of faults. Experiments also need to be repeated many times in order to allow for statistically significant analysis.

These goals are achieved through a rigorous process that is enabled by off-the-shelf hardware (see Section 3.1) and implemented by custom software (see Section 3.2).

3.1 Hardware Architecture

The fault-injection environment is based on a cluster of (6) Elektrobit EB 6120[1] prototyping nodes that communicate with each other over a linear FlexRay communication bus. The prototyping nodes feature MFR4310 FlexRay controllers. The $^{\text{TTX}}$Disturbance node, by TTTech, provides fault-injection capabilities at the FlexRay physical-layer and protocol-layer. An Amrel ePower PDS8202 programmable power-supply provides (8) independent DC out-

[1] Formerly known as DECOMSYS NODE MPC5200

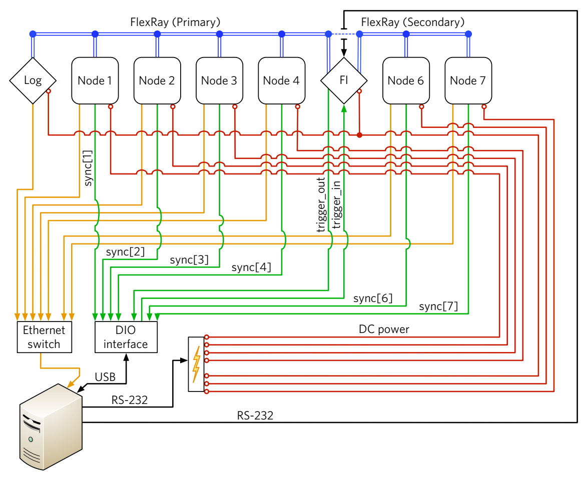

Figure 1. The hardware architecture of the fault-injection environment.

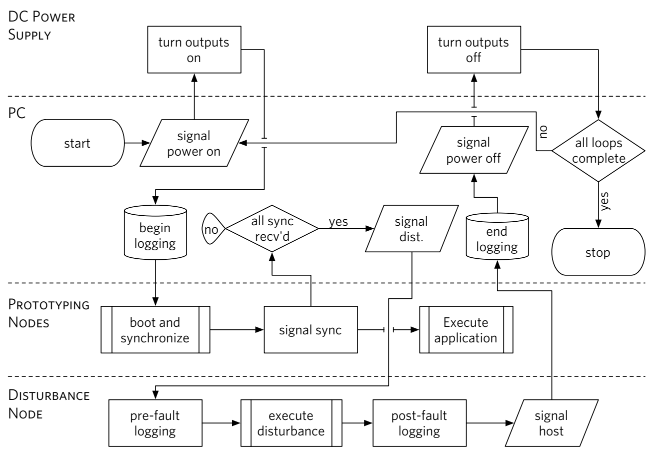

Figure 2. The fault-injection process requires coordinating actions across disparate components.

puts that allow each node to be power-cycled programmatically.

The placement of the ^{TTX}Disturbance node corresponds to the location of injected physical-layer faults. For example, in the current topology, the bus lines can be short circuited or broken between the fourth and fifth prototyping nodes (see Figure 1).

A Windows-based personal computer (PC) communicates directly with the prototyping nodes using an Ethernet backchannel. The PC also controls the power supply and ^{TTX}Disturbance node via RS-232 connections. Finally, a National Instruments USB-6008 Data Acquisition (DAQ) unit connected to the PC provides digital and analog I/O interfaces for triggers and signals. Figure 1 shows the hardware architecture in detail.

3.2 Experimental Process

The high-level experimental process is fairly straightforward.

Step 1 *Power-cycle each prototyping node to reset its internal state.*

Step 2 *As the the prototyping nodes boot and synchronize, begin logging observations.*

Step 3 *Once all of the prototyping nodes have synchronized, delay for a specified time to allow for steady-state observations (i.e., the* pre-fault delay*).*

Step 4 *Wait for a repeatable* trigger *before activating a fault of some* duration *and* type.

Step 5 *When the fault has passed, allow additional time to log observations as any lingering fault-effects pass (i.e., the* post-fault delay*).*

Step 6 *Return to Step 1 and repeat the process, if required.*

A custom control-application that runs on the PC implements the configuration, coordination, and data-collection functionality that this process requires.

3.2.1 Configuration

The *pre-fault delay* (from Step 3); fault *duration*, *type* and *trigger* (from Step 4); and *post-fault delay* (from Step 5) are set in a configuration file that is uploaded to the ^{TTX}Disturbance node by the PC over RS-232. This configuration file is known as a *disturbance scenario*, and specifies the faulty behavior that is applied during the fault-injection process.

The number of times that a particular disturbance scenario is repeated is defined by the *reps* parameter, which is provided as an input to the control application.

3.2.2 Coordination

Implementing this high-level process involves many steps taken by disparate components without any common communication channel. The PC provides the "glue" required to coordinate these steps (see Figure 2).

The PC commands the power supply to turn on its outputs, which causes the prototyping nodes to boot and synchronize. The fault should not be injected until all of the prototyping nodes have synchronized (i.e., reached a steady-state). The ^{TTX}Disturbance node does not provide a way to wait until *all* nodes achieve synchronization, so the prototyping nodes each send a `sync` signal through the DAQ to the PC. Once the PC has received all of the `sync` signals, it sends a signal through the DAQ to trigger the ^{TTX}Disturbance node, which begins running the configured disturbance scenario. The PC then waits until it receives signal from the ^{TTX}Disturbance node that the disturbance scenario has terminated. After receiving the signal, the PC commands the power supply to turn off its outputs. If the configured number of iterations has been completed, then the PC ends the process. Otherwise, the process is repeated.

3.2.3 Data Collection

The PC provides centralized data-collection functionality for observations made by the prototyping nodes. Each prototyping node is assigned to a dedicated port on the PC that listens for incoming data using User Datagram Protocol (UDP). All of the data that the PC receives is tagged with the source node ID and experiment ID and then logged for offline analysis.

4 Pilot Study Specification

For this pilot study, the prototyping nodes were loaded with a simple application. Note that the purpose of this application was not to provide realistic application-level behavior. Rather, it was only used to stimulate bus traffic. Each node transmitted a message counter and a node IDs using two frames in the static segment and a single (arbitrated) frame in the dynamic segment. These frames were received by all controllers but not read by the application. The OS task-schedule and FlexRay communication-schedule were configured to execute synchronously with a 2ms period. The nodes were connected in a linear topology, as shown in Figure 1. Each node in the network was configured as a sync-node with respect to the FlexRay protocol.

4.1 Instrumentation

The FlexRay specification defines various data structures that indicate the status of the communication protocol. Such data structures provide metrics that can be used to detect error patterns. They are accessed though the FlexRay CHI, which is accomplished on the MFR4310 controller by reading and writing 16-bit memory-mapped registers.

A custom instrumentation-component runs on each EB 6120 node and makes observations by reading a subset of the CHI registers at the beginning of each task period (e.g., once every 2ms). Observations are buffered in volatile memory on the node. The instrumentation component periodically empties the buffer by sending all of the stored observations to the PC using the Ethernet backchannel.

For this study, we observed 4 discrete error-indicators during each experiment: Boundary Violations (BVs), Syntax Errors (SERRs), Content Errors (CERRs) and Valid Frames (VFs). The indicators themselves were aggregated over the entire communication cycle. Observations were made at the beginning of each communication cycle by reading error indicators from the FlexRay CHI.

4.2 Disturbance Scenarios

The consisted of 5 disturbance scenarios, each of which was repeated 100 times. For this study, we choose to focus on a small set of physical-layer faults. Each scenario was associated with a different fault-type, which was activated for 500ms. No fault was activated during the **`none`** scenario, which provided a *baseline case*. The **`break`** scenario caused a physical separation of the Bus Plus (BP) and Bus Minus (BM) lines. The **`noise`** scenario injected differential white noise onto the bus. The **`short_vcc`** and **`short_gnd`** scenarios short-circuited the BP line to ground and supply voltage, respectively. Observations from each node were logged for 1s prior to activation and 5s following deactivation of the disturbance.

5 Results

Recall that the instrumentation component records an observation once every 2ms. Therefore, 500 observations are expected during the pre-fault period (1000 ms); 250 observations are expected while the fault is active (500 ms); and 2500 observations are expected during the post-fault period (5000 ms). For this study, we are not interested in observations made during the synchronization phase. In total, each experiment is expected to produce 3250 total observations. Note that because each observation accumulates indicators over an entire cycle, you can have multiple indicators set in a single observation (i.e., the sum of the observed indicators may be greater than the total number of observations).

`none` **(baseline)** As expected, no error indicators were observed during the baseline scenario.

`break` The `break` scenario resulted in SERRs, CERRs and BVs at each node, with some variation across nodes (see fig. 3). The `break` scenario was further distinguished by being the only scenario to result in CERRs (see fig. 4).

`noise` A roughly equal number of BVs and SERRs were observed during the `noise` scenario, along with a corresponding drop in VFs (see fig. 5).

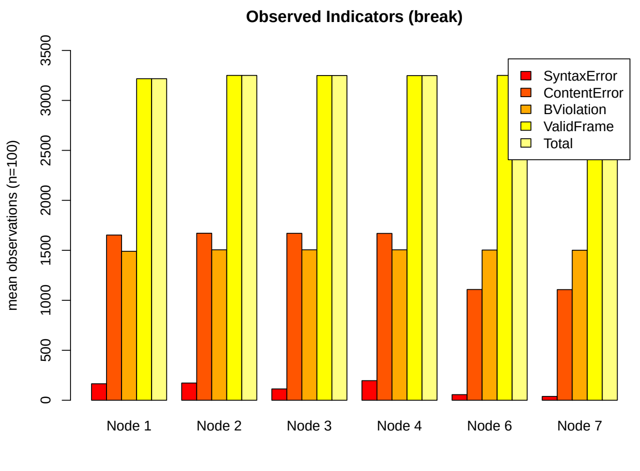

Figure 3. The `break` scenario resulted in SERRs, CERRs and BVs at each node, with some variation across nodes.

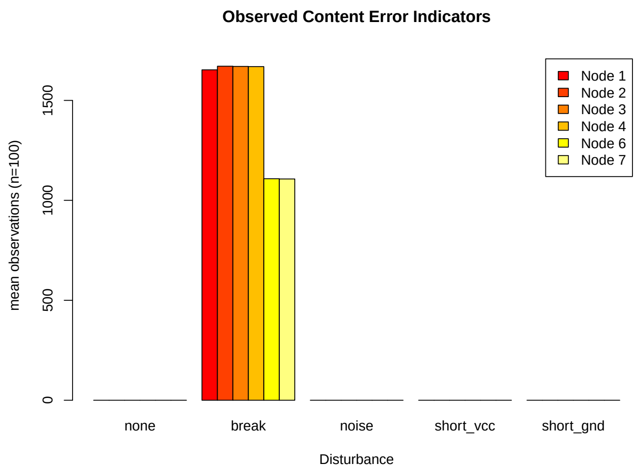

Figure 4. Content Errors (CERRs) were only observed during the `break` scenario.

`short` Short circuits resulted in a drop in valid frames, compared to the total number of observations (see fig. 6). Between three and four syntax errors were also observed consistently during each short circuit. There was no measurable difference in the effects of shorting to ground vs. shorting to supply voltage.

6 Discussion

In order to look for potential error patterns, we simply compared the average number of times a particular indicator is observed for each scenario. Thus, we can deduce the following preliminary error patterns:

`break`: many content errors and boundary violations with comparatively few syntax errors

`noise`: equal number of syntax errors and boundary violations with a corresponding drop in valid frames

`short_vcc`: lack of valid frames with comparatively few syntax errors

`short_gnd`: lack of valid frames with comparatively few syntax errors

The fault-injection process itself showed good controllability and repeatability. It was trivial to specify the experimental parameters accurately. Given consistent parameters, the process produced consistent results.

7 Related Work

A useful overview of general fault injection techniques is available in [9]. Here, we focus on fault injection techniques that specifically target the automotive domain, with an emphasis on diagnosis.

Fault-injection experiments using heavy-ion fault injection have shown fail-silence violations [20] and error propagation [1] in Time-Triggered Protocol/Class-C (TTP/C). Fault-injection experiments using hardware models have show that transient faults in the CAN Communication Controller (CC) and Communication Network Interface (CNI) can result in masquerade failures, where a faulty node "impersonates" a non-faulty node [17]. Within the context of diagnosis, this result can be viewed as a false-positive on the non-faulty node as well as a false-negative on the true faulty node. Experiments performed using modeled FlexRay controllers have highlighted instances of error propagation in FlexRay bus and star topologies [5].

The online diagnosis algorithms developed by [22] were extended to discriminate healthy nodes from unhealthy nodes in time-triggered automotive systems [18]. A prototype implementation of the protocol was built using TTP/C controllers and analyzed via physical fault injection [19].

Out-of-Norm Assertions (ONAs) are introduced as a way to correlate fault effects in the three dimensions of value, time and space . The ONA mechanism underlies a framework for diagnosing failures in time-triggered networks [14]. A prototype implementation of the framework using TTP/C controllers instruments the frame status field of the controller, which is similar to the information available from the FlexRay CHI. The $^{\mathrm{TTP}}$Disturbance node was used

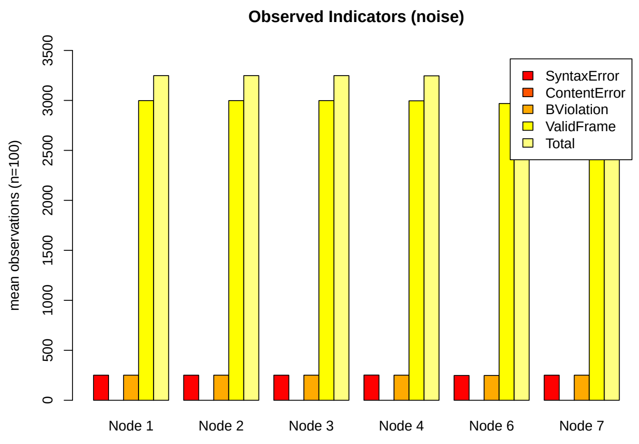

Figure 5. The `noise` scenario produced a roughly equal number of BVs and SERRs with a corresponding drop in VFs.

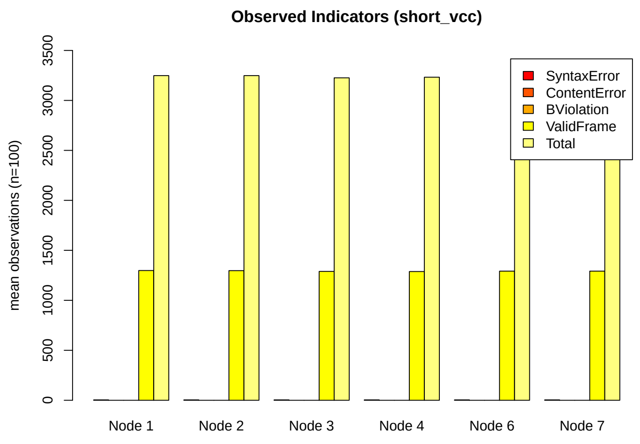

Figure 6. Both short-circuit scenarios produced similar drops in valid frames.

along with Electromagnetic Interference (EMI) injection to evaluate the effectiveness of this diagnostic framework as applied to connector faults TTP/C [15].

8 Conclusion

These preliminary results suggest that (1) *errors do not manifest uniformly across nodes* and (2) *identifiable error patterns may exist*. However, these results cannot be generalized beyond this small study. These patterns are likely to change, perhaps drastically, depending on many factors such as network topology, communication schedule, node configuration, etc. Furthermore, other disturbances that we did not include in this pilot might show similar patterns, making them difficult to distinguish from each other. Clearly, more advanced analysis will be required for more robust fault models.

References

[1] A. Ademaj, H. Sivencrona, G. Bauer, and J. Torin. Evaluation of fault handling of the time-triggered architecture with bus and star topology. In *Proceedings, 2003 International Conference on Dependable Systems and Networks*, DSN '03, pages 123–132, Los Alamitos, CA, USA, June 2003. IEEE Computer Society.

[2] J. Arlat, M. Aquera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990.

[3] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings, 8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 103–116, Berkeley, CA, USA, December 2008. USENIX Association.

[4] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. *ACM SIGOPS Operating Systems Review*, 39(5):105–118, December 2005.

[5] M. Dehbashi, V. Lari, S. G. Miremadi, and M. Shokrollah-Shirazi. Fault effects in flexray-based networks with hybrid topology. In *Proceedings, 3rd International Conference on Availability, Reliability and Security*, ARES '08, pages 491–496, Los Alamitos, CA, USA, March 2008. IEEE Computer Society.

[6] FlexRay Consortium. *FlexRay Communications System Protocol Specification*, December 2005.

[7] H.-G. Frischkorn. Automotive software – the silent revolution. Automotive Software Workshop, San Diego, CA, Jan 2004.

[8] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings, 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOSPLA '04, pages 251–269, New York, NY, USA, October 2004. ACM.

[9] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[10] *ISO/DIS 26262: Road vehicles – Functional safety*, volume 4–6. International Organization for Standardization, Geneva, Switzerland, 2009.

[11] S. Kavulya, R. Gandhi, and P. Narasimhan. Gumshoe: Diagnosing performance problems in replicated file-systems. In *Proceedings, 2008 IEEE Symposium on Reliable Dis-*

tributed Systems, SRDS '08, pages 137–146, Los Alamitos, CA, USA, October 2008. IEEE Computer Society.

[12] P. E. Lanigan and P. Narasimhan. Holistic data-driven diagnosis for dependable automotive systems. Appeared in *NIST/NSF/USCAR Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components*, March 2011. Available at http://varma.ece.cmu.edu/Auto-CPS-2011/Papers/index.html.

[13] P. E. Lanigan, P. Narasimhan, and T. E. Fuhrman. Experiences with a CANoe-based fault injection framework for AUTOSAR. In *Proceedings, 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, pages 569—574, Los Alamitos, CA, USA, June 2010. IEEE Computer Society.

[14] P. Peti and R. Obermaisser. A diagnostic framework for integrated time-triggered architectures. In *Proceedings, 9th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '06, page 11pp, Los Alamitos, CA, USA, April 2006. IEEE Computer Society.

[15] P. Peti, R. Obermaisser, and H. Paulitsch. Investigating connector faults in the time-triggered architecture. In *Proceedings, 11th IEEE Conference on Emerging Technologies and Factory Automation*, ETFA '06, pages 887–896, Piscataway, NJ, USA, September 2006. IEEE.

[16] J. D. Rupp and A. G. King. Autonomous driving – a practical roadmap. SAE Technical Paper Series 2010-01-2335, SAE International, Warrendale, PA, USA, October 2010.

[17] H. Salmani and S. G. Miremadi. Contribution of controller area networks controllers to masquerade failures. In *Proceedings, 11th Pacific Rim International Symposium on Dependable Computing*, PRDC '05, page 5, Los Alamitos, CA, USA, December 2005. IEEE Computer Society.

[18] M. Serafini, A. Bondavalli, and N. Suri. Online diagnosis and recovery: On the choice and impact of tuning parameters. *IEEE Transactions on Dependable and Secure Computing*, 4(4):295–312, October-November 2007.

[19] M. Serafini, N. Suri, J. Vinter, A. Ademaj, W. Brandstäter, F. Tagliabò, and J. Koch. A tunable add-on diagnostic protocol for time-triggered systems. In *Proceedings, 2007 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 164–174, Los Alamitos, CA, USA, June 2007. IEEE Computer Society.

[20] H. Sivencrona, P. Johannessen, and J. Torin. Protocol membership in dependable distributed communication systems – a question of brittleness. SAE Technical Paper Series 2993-01-0108, SAE International, Warrendale, PA, USA, March 2003.

[21] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. M. Howard, S. Kolski, A. Kelly, M. Likhachev, M. McNaughton, N. Miller, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, B. Salesky, Y.-W. Seo, S. Singh, J. Snider, A. Stentz, W. R. Whittaker, Z. Wolkowicki, J. Ziglar, H. Bae, T. Brown, D. Demitrish, B. Litkouhi, J. Nickolaou, V. Sadekar, W. Zhang, J. Struble, M. Taylor, M. Darms, and D. Ferguson. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, July 2008.

[22] C. J. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *IEEE Transactions on Software Engineering*, 23(11):684–721, November 1997.

[23] D. Wilson. Ray of hope for auto industry. *Electronic Business*, Nov 2006. Available at http://www.edn.com/article/CA6385672.html.

Acronyms

BM	Bus Minus	4
BP	Bus Plus	4
BV	Boundary Violation	4
CC	Communication Controller	5
CERR	Content Error	4
CHI	Controller-Host Interface	2
CNI	Communication Network Interface	5
DAQ	Data Acquisition	3
EMI	Electromagnetic Interference	6
ID	identifier	2
OEM	Original Equipment Manufacturer	2
ONA	Out-of-Norm Assertion	5
OS	Operating System	2
PC	personal computer	3
SERR	Syntax Error	4
TTP/C	Time-Triggered Protocol/Class-C	5
UDP	User Datagram Protocol	4
VF	Valid Frame	4