Experiences with a CANoe-based Fault Injection Framework for AUTOSAR

Patrick E. Lanigan, Priya Narasimhan
ECE Department
Carnegie Mellon University
Pittsburgh, PA 15213
planigan@ece.cmu.edu, priya@cs.cmu.edu

Thomas E. Fuhrman
General Motors Research & Development
Warren, MI 48090
thomas.e.fuhrman@gm.com

Abstract

Standardized software architectures, such as AUTomotive Open System ARchitecture (AUTOSAR), are being pursued within the automotive industry in order to reduce the cost of developing new vehicle features. Many of these features will need to be highly dependable. Fault injection plays an important role during the dependability analysis of such software. This work evaluates the feasibility of leveraging the CANoe simulation environment to develop software-based methods for injecting faults into AUTOSAR applications. We describe a proof-of-concept fault-injection framework with example fault-injection scenarios, as well as implementation issues faced and addressed, lessons learned, and the suitability of using CANoe as a fault-injection environment.

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 suggested that "up to 90% of all innovations are driven by electronics and software" [5]. It is further estimated [5] that 50-70% of an Electronic Control Unit (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 [5]. Recently, the industry has seen the emergence of standardized software architectures, such as AUTOSAR [4], as a way to promote software reuse and reduce the development costs incurred when introducing new software-based features. AUTOSAR was developed through the partnership of over 100 companies, including the major automobile manufacturers and their suppliers [4].

When problems in software-based systems are uncovered after a vehicle has gone to production, recall costs can rival development costs [14]. Moreover, many of these advanced systems are critical to ensuring the safe operation of the vehicle. Therefore, they must be designed to tolerate faults and provide high levels of dependability.

If AUTOSAR is to be used in these systems, its own fault and error handling capabilities must be characterized. Formal verification methods are typically 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 [8].

The automotive industry is traditionally cost-sensitive, and hardware-based fault injection typically requires expensive, specialized equipment. A low-cost, software-based fault-injection framework could provide a substitute for hardware-based techniques when their full functionality is either not required (e.g., during software prototyping) or not appropriate (e.g., for targeting specific software components). Software-based techniques are also easier to reconfigure and to modify (e.g., by adding a new type of fault or a new point of fault injection) than hardware-based techniques, where new hardware might need to be created for every new modification.

This report describes an experience using off-the-shelf tools to build a low-cost fault-injection framework that supports AUTOSAR. The framework uses software-implemented techniques in a simulated execution environment, namely, Vector CANoe. The framework consists of hooks that are inserted into the AUTOSAR codebase and implemented in a separate Software Implemented Fault-Injection (SWIFI) module. Hooks can manipulate specific data structures or directly force error codes. Faults are manually configured and activated at run-time using a CANoe control-panel. The purpose of this proof-of-concept was to evaluate the feasibility of building such a system, and not to provide a dependability analysis of the AUTOSAR specifi-

cation or of the specific AUTOSAR implementation used.

To the best of our knowledge, this represents the first published attempt to inject faults into an AUTOSAR application running in a simulation environment. It provides a starting point for researchers and practitioners to identify suitable fault-injection locations within the AUTOSAR stack. This work suggests that CANoe is a suitable fault-injection environment for some faults, but that other faults cannot be represented using the level of abstraction that CANoe provides. It provides an important first step in developing a more comprehensive fault-injection framework for AUTOSAR applications that integrates both hardware- and software-based techniques.

The remainder of this report is organized as follows. Section 2 provides some background for our work, while Section 3 outlines our specific goals. The fault-injection framework is presented in Section 4. We evaluate the framework in Section 5 and discuss lessons learned in Section 6. Section 7 covers some related work. Section 8 summarizes this report.

2. Background

AUTOSAR is an emerging standard software-architecture for automotive applications. It is a layered architecture, with each layer containing components that provide services to higher layers [3]. Components in the Basic Software (BSW) layers provide services based on hardware abstractions, while application-layer components implement application functionality. The Runtime Environment (RTE) layer enables information exchange between components in the application and BSW layers. Typically, vendors provide their own proprietary AUTOSAR-compliant implementations of the BSW and RTE, which the vehicle manufacturer then uses to develop application-level software.

Built-in error-handling mechanisms allow components to be informed of errors as they occur, with low-level errors being abstracted as they are passed up the stack [4]. Exercising the AUTOSAR error handling mechanisms at different layers would provide empirical insight into how errors are abstracted between components. To that end, a fault-injection framework should provide the capability to cause errors in specific AUTOSAR components at different layers. This could occur directly by forcing a component to return an error code, or indirectly by corrupting a data structure (i.e., a memory location) within the component.

There are various ways to inject faults into a system (see Section 7). Any adopted fault-injection framework should account for the cost-sensitive nature of the automotive industry. In order to minimize costs, it is desirable to utilize existing resources and off-the-shelf tools as much as possible. The costs incurred in developing the fault-injection framework can be further amortized by encouraging portability between systems and applications. A significant portion of the AUTOSAR codebase is contained in files that are auto-generated from network-, system- and application-specific configuration files. Therefore, any changes required to facilitate fault injection in the AUTOSAR codebase should avoid auto-generated files completely. Modifying configuration files (e.g., CANdb, FIBEX) should be avoided as well.

We decided to explore a proof-of-concept fault-injection framework based on a common and readily available software tool: Vector CANoe. CANoe provides a simulation and evaluation environment for automotive applications. During early stages in development, it allows networks of ECUs to be simulated using behavioral models. Communication Access Programming Language (CAPL) is a scripting language used within CANoe to define the functionality of simulated nodes and to control the simulation environment. An AUTOSAR application can be simulated in CANoe by compiling the AUTOSAR codebase as a Dynamic-Link Library (DLL) and associating the DLL with a single simulated node. As development progresses, simulated nodes can be mixed with physical networks to aid integration testing.

3. Goals

The initial question that we wanted to answer was whether or not faults could be injected in AUTOSAR using only the tools available to us (i.e., CANoe). If so, we wished to gain a practical understanding of such a framework's strengths and weaknesses. Therefore, this proof-of-concept was developed without regard for any single fault model. After all, if the proof-of-concept is not feasible in the first place, then it is irrelevant to question whether or not the framework supports a particular fault model. Once the the general advantages and disadvantages of the framework have been identified, future work can formally assess its applicability to various fault models.

The proof-of-concept will be evaluated with respect to six specific goals, which are outlined below.

- **Functionality.** The framework should be able to exercise the AUTOSAR error-handling mechanisms. The user should be able to control fault injection parameters and view the effects of injected faults at run-time within CANoe.
- **Controllability.** The user should be able to specify and perform experiments with parameters that are repeatable in time (e.g., fault duration) and space (e.g., fault location).
- **Observabilty.** The effect(s) – or lack thereof – of an injected fault should be readily apparent. It should be

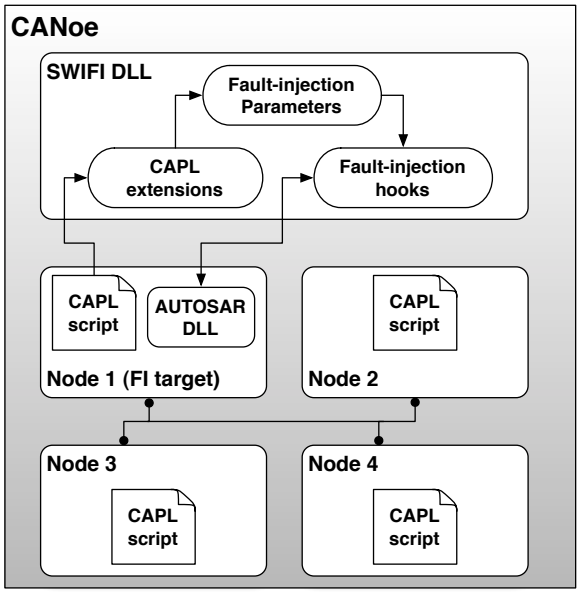

Figure 1. An overview of the fault-injection framework.

possible to distinguish between a fault that has no effect (e.g., injected into "dead code") versus a fault that is masked by some error-handling mechanism.

- **Portability.** The modifications needed to apply the framework to different AUTOSAR-based applications, and to port it between AUTOSAR implementations, should be minimal.

- **Flexibility.** The framework should be flexible enough to support a wide range of fault-injection scenarios and fault types.

- **Probe effects.** It is important for the fault-injection framework to avoid probe effects, which are unintended and undesired alterations to the system caused by the fault injection process itself (as opposed to effects attributed to the faults being injected).

4. A Fault-Injection Framework for CANoe

Our fault-injection framework consists of two major components: an **AUTOSAR DLL** that implements application functionality, and a **SWIFI DLL** that implements fault-injection functionality (see Figure 1).

Fault-injection hooks are defined in the SWIFI DLL and called from the AUTOSAR DLL. Two types of fault-injection hooks can be inserted into the AUTOSAR codebase. *Suppression hooks* cause errors directly. When a suppression hook is active, it signals that an AUTOSAR Application Programming Interface (API) call should abort immediately and, depending on the API, return an error code to the caller. *Manipulation hooks* cause errors indirectly by manipulating specific data structures. The arguments passed to manipulation hooks can be either modifiable or non-modifiable. Modifiable arguments represent the data structures that can be manipulated inside of the hook. Non-modifiable arguments are not manipulated inside of the hook. Rather, they can be used to provide additional context to the hook (e.g., current slot, global time, etc). In general, active suppression hooks take precedence over active manipulation hooks (i.e., the target API call will be aborted before its data structures are manipulated).

The SWIFI DLL also implements six **fault-injection parameters** that define faults, as well as **CAPL extensions** that allow the parameters to be changed by CAPL scripts (see Figure 1). The *location* parameter refers to the AUTOSAR API call where the fault is to be injected. The *argument* parameter represents a specific data structure (e.g., local or global variable) within the scope of that call, which is to be manipulated by applying a *mask* using a bitwise *operation*. For large data structures, an optional *offset* can specify which bytes the mask should be applied to. The sixth parameter is a *flag* that specifies whether a suppression hook is active at the selected location. In this case, the argument, mask, operation and offset parameters are ignored.

A CANoe panel provides a user interface that allows manual control over the fault-injection process. However, this panel cannot access the fault-injection parameters directly. Instead, the panel modifies CANoe system-variables. These system variables are then mapped to fault-injection parameters by a CAPL script, which is associated with the target node (e.g., Node 1 in Figure 1).

4.1. Fault-Injection Hooks

We selected three components at different BSW layers to target for fault injection. We did not add hooks to the RTE because it is mostly auto-generated by the AUTOSAR configuration tools. The Watchdog Manager (WdgM) is part of the System Services layer and provides a service that monitors selected AUTOSAR components for timing violations. Hooks added to the WdgM allow watchdog timeouts to be triggered manually. The AUTOSAR COM (Com) component resides in the Communication Services layer, just below the RTE, and provides a signal-based gateway for routing data between components. Injecting faults in the Com component allows both inter-node (i.e., components on different ECUs) and intra-node (i.e., components on the same ECU) communication to be disrupted. It also provides a basis for comparison with faults injected at lower layers. The FlexRay Driver (Fr) provides an abstraction of the FlexRay controller hardware. It is part of the Commu-

AUTOSAR Layer	AUTOSAR API	Description	Manipulated data
System Services	`WdgM_UpdateAliveCounter`	Notifies the WdgM that a supervised entity is still alive.	entity identifier
Communication Services	`Com_SendSignal`	Sends data to the RTE as an AUTOSAR signal.	signal identifier, data buffer
Communication Services	`Com_ReceiveSignal`	Receives data from the RTE as an AUTOSAR signal.	signal identifier, data buffer
Communication Drivers	`Fr_TransmitTxLPdu`	Transfers data to the FlexRay controller as a PDU.	data buffer, buffer length, channel, slot identifier, controller flags
Communication Drivers	`Fr_ReceiveRxLPdu`	Receives data from the FlexRay controller as a PDU.	data buffer, buffer length, controller flags

Table 1. Hooks were added to manipulate data structures in five AUTOSAR API calls.

nication Drivers layer, which is the lowest-level BSW layer. These hooks are meant to cause low-level FlexRay protocol errors that are then passed to higher layers or onto the FlexRay bus directly.

Hooks were added to a total of five AUTOSAR API calls, listed in Table 1. Complete descriptions of these APIs, including their arguments and error codes, can be found in the AUTOSAR specifications. The functions implementing each call were modified to contain both manipulation hooks and suppression hooks. Suppression hooks were generally placed at the very beginning of the function, before any built-in error checking takes place. Thus, the entire API call was aborted and any side effects of the call (intended or otherwise) were avoided. Manipulation hooks were generally placed after the target argument had been initialized, but before it was first used.

5. Evaluation

For this evaluation, we used a demo version of CANoe (v7.1, running on Windows XP) and a vendor-supplied implementation of the AUTOSAR 3.0 specification. We applied the fault-injection framework to a simple "by-wire" application developed specifically for CANoe demonstrations. We should note that this application was not designed to be fault-tolerant, and does not provide any application-level error handling. However, it is sufficient for demonstrating that faults can be injected and visualized. In fact, application-level error handling could make it more difficult to verify that the fault injection framework itself is functional by masking errors that would otherwise be visible.

The application consists of two FlexRay nodes, a single Controller Area Network (CAN) node and a CAN/FlexRay gateway. The CAN node sends throttle and brake inputs through the gateway to the FlexRay nodes. The FlexRay nodes calculate the wheel speed for the front and rear wheels, respectively. A CANoe control panel provides user-adjustable throttle and brake controls, as well as a graphical display of the calculated front and rear wheel speed. The functionalities of the CAN node, the gateway node and the front FlexRay node are implemented with CAPL scripts. The functionality of the rear FlexRay node is provided by an AUTOSAR DLL that has been instrumented with fault-injection hooks at the locations listed in Table 1. When no faults are active, the speed reported by the front and rear wheels changes smoothly with respect to changes in the throttle value, and identically with respect to each other.

Some manipulations designed to manifest at the FlexRay protocol level worked as expected. For example, modifying the channel argument caused otherwise valid frames to be sent on the wrong channel, while null frames were registered on the correct channel. Applying a mask to set the null-frame indicator bit in `Fr_TransmitTxLPdu` also caused null frames to be sent. In this case, the rear-wheel speed did not change when the throttle increased. Other manipulations designed to cause FlexRay protocol errors were not as effective. For example, the FlexRay protocol is designed to register a *ContentError* when a static frame is received with the startup frame indicator set to 1 while the sync frame indicator is set to 0. However, applying a mask to set these bits in the controller flags prior to sending the frame from `Fr_TransmitTxLPdu` did not result in a *ContentError* being recorded by CANoe. It is likely that the simulated controller ignores these bits in favor of the preconfigured FlexRay parameters.

Suppression hooks in components at different layers caused visible differences in temporal manifestation. Permanently forcing `Com_ReceiveSignal` to return an error code caused the wheel speed at the rear FlexRay node to drop immediately and no longer respond to throttle changes. However, when the same hook was activated in `Fr_ReceiveRxLPdu`, no differences in front vs. rear wheel speed were visible until the throttle input was changed. A possible explanation is that suppressing `Fr_ReceiveRxLPdu` prevents the receive buffer from being updated when a frame is received. However, the previous value contained in this buffer can still be read and passed through the AUTOSAR stack to the application. In contrast, suppressing the higher-level `Com_ReceiveSignal` call seems to prevent any data from reaching the application software.

Targetting the `WdgM_UpdateAliveCounter` API

call exercised two of AUTOSARs built-in error-handling mechanisms. Modifying the entity-identifier parameter caused errors to be reported by the AUTOSAR Development Event Tracer (DET). Suppressing the API call entirely caused the watchdog timer to expire and report an error.

6. Lessons Learned

This proof-of-concept framework showed promise in several ways, but there were also significant drawbacks to using CANoe in this manner.

Functionality. The functionality of the framework was mostly satisfactory. By forcing API calls to return error codes and modifying certain data structures, it is possible to exercise AUTOSAR error-handling mechanisms such as the DET and WdgM. Furthermore, manual control of the simulation allows faults to be visualized as they are injected. This feature was particularly useful while developing the framework because it gave us rapid insight into whether the framework itself was functioning correctly.

Portability. Using hooks in the AUTOSAR code allows fault injection logic to be implemented in a separate module, keeping the required changes to AUTOSAR codebase minimal. No auto-generated code or system-specific configuration files need to be modified, which allows the same instrumented AUTOSAR codebase to be used across projects. However, some of the hooks target arguments that correspond to implementation-specific data-structures. In the future, it might be desirable to target only the API parameters that are defined in the AUTOSAR specification. This would allow greater portability of the SWIFI DLL across AUTOSAR implementations from different vendors.

Controllability. Manipulating data structures, as opposed to arbitrary memory locations, allows specific components and functionality within the AUTOSAR stack to be targeted. Even if the physical memory address associated with a data structure changes (e.g., due to recompilation), the semantic meaning of the fault location does not. Temporal repeatability is currently limited by our initial focus on manual triggering. The activation and duration of faults is only as accurate as the user "flipping the switch". However, this is an implementation issue and not an inherent design issue. As noted in Section 4, non-modifiable arguments are available to provide extended context to fault-injection hooks. Such context can be used to define more specific trigger conditions, which could then be automated using scripts.

Observability. Observability is restricted to application-level failure modes and existing error-handling mechanisms. If a fault is masked by an error-handling mechanism, it will only be observable if the mechanism logs the error or otherwise notifies the user.

Flexibility. It is not possible to inject many protocol-level faults (e.g., frame decoding errors) directly using this framework. In general, fault-injection hooks implemented in software simply have limited access to the internal functions of hardware communication controllers. However, methods using simulated hardware models have been successfully employed to inject faults directly in CAN [10, 12] and FlexRay [9] controllers. We had hoped that the simulated controller in CANoe would allow similar transparency. However, the level of abstraction provided by CANoe is more high-level, and the simulated controller can only be accessed through its external interfaces.

Another limitation is that the SWIFI DLL cannot differentiate between calls to hooks made from multiple AUTOSAR DLLs. Therefore, the framework can only support a single fault-injection node. Removing this limitation would allow the framework to support scenarios involving cascading and correlated faults among multiple ECUs

Probe effects. The principal drawback to this approach is that it is difficult to avoid simulation-wide probe effects when injecting certain types of faults. For example, modifications to certain arguments (such as the length of a data buffer) would normally be expected to cause memory corruption on the faulty node. However, because nodes share memory space with and within the CANoe simulator, injecting such a fault crashes the entire simulation. Similarly, attempting to cause timing violations directly by inserting artificial delays into tasks can potentially block the entire simulation.

7. Related Work

A useful overview of general fault injection techniques is available in [7]. Here, we focus on fault injection techniques that specifically target the automotive domain.

Faults can be injected directly into hardware by exposing it to heavy-ion radiation or Electromagnetic Interference (EMI). Heavy ion fault injection was used to investigate the fail-silence assumption and membership service of the Time-Triggered Protocol/Class-C (TTP/C) protocol [13]. Another study compared error propagation in TTP/C bus and star network topologies [1].

Fault-injection hardware can also work at the bus-level in order to target the communication protocol specifically. These devices can be used to inject physical faults directly (e.g, by shorting the bus lines) or to inject fault manifestations indirectly (e.g., by employing CRC recalculation). One such device, the $^{\text{TTP}}$Disturbance Node, was used along

with EMI injection to evaluate a strategy for diagnosing connector faults in TTP/C [11].

Software-implemented fault injection (SWIFI) techniques have been proposed specifically for use in automotive time-triggered networks. The FITS fault injection environment follows a modularized approach that separates target-specific code from system-independent code [6]. The use of hooks in our framework was inspired by FITS.

Software-based and hardware-based techniques are complementary. Hardware-based fault-injection using heavy ions or EMI can access physical locations (e.g., in the communication controller) that software hooks simply cannot. However, hardware-based techniques cannot accurately target specific software modules due to the random nature of the process. Bus-level devices are very effective at injecting a variety of protocol-level faults, but they cannot inject faults into the internal memory of individual ECUs.

8. Summary

The framework developed here uses software-implemented techniques (e.g., code insertion) in a simulated environment (e.g., CANoe). Manipulation hooks cause errors indirectly, by injecting faults into the memory occupied by specific data structures. Suppression hooks cause errors directly, by forcing AUTOSAR API calls to return error codes.

The errors caused by injected faults were able to be visualized quickly in CANoe. Visible differences in application-level manifestation were observed when similar faults were injected into different components. However, CANoe did not provide an ideal environment for injecting certain faults due to simulation-wide probe effects. Value faults can be injected into specific AUTOSAR BSW components using this framework, but specialized bus-level fault injection hardware would be better suited to injecting protocol-level faults. Overall, faults can be successfully injected and visualized when they fit within the level of abstraction that CANoe provides. Considering that we were dancing on the boundaries of what the tool was designed to do, CANoe performed quite satisfactorily.

While this study cannot, and should not, be construed as a dependability study of AUTOSAR, the described framework could be further developed to provide the basis for such a study. Given that this was a feasibility study, the focus was on determining *how* faults could be injected using the tools available to us, but not necessarily *which* faults could be, or should be, injected. A more in-depth study is required to determine the most appropriate AUTOSAR locations and data structures to target with fault-injection hooks, and how the injected faults correspond to common fault models.

Acknowledgements

The authors wish to acknowledge Sandeep Menon and Larry Peruski at General Motors R&D. Sandeep is the original developer of the by-wire demo that we adapted for this study. Larry Peruski provided a great deal of support in helping us become familiar with the lab equipment.

References

[1] A. Ademaj et al. Evaluation of fault handling of the time-triggered architecture with bus and star topology. In *International Conference on Dependable Systems and Networks (DSN)*, pages 123–132. IEEE Computer Society, June 2003.

[2] J. Arlat et al. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, Feb 1990.

[3] AUTOSAR GbR. *Layered Software Architecture*, 2008. Document ID 053. http://www.autosar.org.

[4] H. Fennel et al. Achievements and exploitation of the AUTOSAR development partnership. SAE Technical Paper Series 2006-21-0019, SAE International, October 2006.

[5] H.-G. Frischkorn. Automotive software - the silent revolution. Automotive Software Workshop, Jan 2004.

[6] R. Hexel. Fits: A fault injection architecture for time-triggered systems. In *Australasian Computer Science Conference (ACSC)*, volume 16, pages 333–338. ACS, Feb 2003.

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

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

[9] V. Lari et al. Assessment of message missing failures in flexray-based networks. In *Pacific Rim Internation Symposium on Dependable Computing (PRDC)*, pages 191–194. IEEE Computer Society, Dec 2007.

[10] J. Pérez, M. S. Reorda, and M. Violante. Accurate dependability analysis of CAN-based networked systems. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 337–342. IEEE Computer Society, Sept 2003.

[11] P. Peti, R. Obermaisser, and H. Paulitsch. Investigating connector faults in the time-triggered architecture. In *Emerging Technologies and Factory Automation (ETFA)*, pages 887–896. IEEE, Sept 2006.

[12] H. Salmani and S. G. Miremadi. Contribution of controller area networks controllers to masquerade failures. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, page 5. IEEE Computer Society, Dec 2005.

[13] 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, Mar 2003.

[14] D. Wilson. Ray of hope for auto industry. *Electronic Business*, Nov 2006.