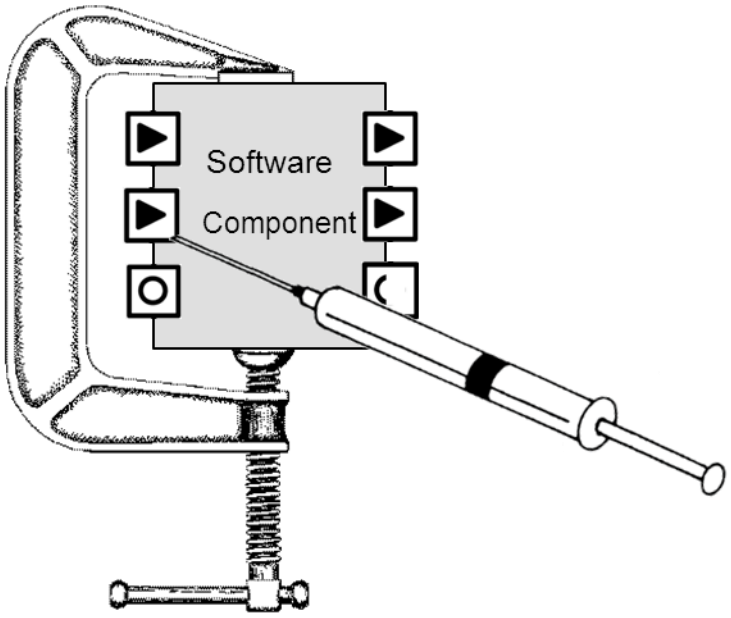
CHALMERS



Software implemented fault injection for AUTOSAR based systems

Master of Science Thesis
Software Engineering and Technology Programme

JOHAN HARALDSSON
SIGURJÓN ÞORVALDSSON

Chalmers University of Technology
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2012

Software implemented fault injection for AUTOSAR based systems

JOHAN HARALDSSON
SIGURJÓN ÞORVALDSSON

Supervisor: JOHAN KARLSSON
Examiner: MATTHIAS TICHY

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
Syringe image in courtesy of www.aperfectworld.org. Clamp image in courtesy of www.CKSinfo.com. The software component image is drawn according to AUTOSAR modelling standard (www.autosar.org).

Department of Computer Science and Engineering
Gothenburg, Sweden June 2012

ABSTRACT

This master's thesis describes the design and implementation of a software implemented fault injection tool, which can be used to perform robustness testing on application software components in embedded systems based on the AUTOSAR standard architecture. The thesis analyses the AUTOSAR standard in order to identify mechanisms, which can be used at run-time in order to inject faults. Three techniques are identified: the use of wrappers, the use of trace hooks and modification of the run-time environment. The wrapper technique was found to be most suitable and therefore implemented in a prototype fault injection tool. The fault injection tool is evaluated on two applications. The first application is a calculator application residing on a single electronic control unit, and the second application is a brake-by-wire system distributed over several electronic control units. The validation shows that the tool is successful in injecting faults into the interfaces of application software components, and that it can emulate hardware faults by causing the same reactions in the brake-by-wire system as an open circuit fault. Furthermore, it shows that it is possible to automate the wrapper generation by processing AUTOSAR XML configuration files and that an AUTOSAR complex device driver component can be efficiently utilised as an embedded fault injection controller in order to achieve performance, low intrusion and portability.

Acknowledgements

We would like to thank our supervisor Johan Karlsson for his enthusiasm and sharing of expert knowledge which has inspired and helped us a lot during the thesis. We would also like to thank Svante Möller and express our appreciation for all his support.

Furthermore, we would like to thank, Michael Jones, Mafijul Islam, Mats Olsson, and Fredrik Bernin from the DEDISafe team for all their support.

Finally, we would like to thank Sara and Victoria for their understanding, support and encouragement that have made it possible for us to conduct our studies.

CONTENTS

1. INTRODUCTION

A trend in automotive systems is the increasing use of electrical and electronic (E/E) systems with increased integration and interaction. In 1980, electronics made up for less than 1% of the total cost of a vehicle. In 2010, electronics accounted for roughly 20%, and the cost is expected to rise up to 40% by 2015 [1].

Quality and Safety are two important values in the automotive industry [2], which are also applicable for automotive electronics including embedded software. The use of increasing amounts of electronics and software can have a negative effect on safety and quality, by producing higher overall failure rates, if the increased complexity is not handled properly. One of the reasons for the increasing amounts of electronics is to replace features and components that traditionally have been implemented by mechanics or hydraulics with electronics. This is made in order to reduce component cost, improve fuel consumption and increase controllability.

As a response to the increasing complexity of automotive E/E systems, the major Original Equipment Manufacturers (OEMs) and suppliers within Europe have developed the AUTomotive Open System Architecture (AUTOSAR) standard [3], which includes a methodology for making application software independent of hardware for automotive applications [4].

The increasing amount of electronics has also increased the need for standardising requirements for functional safety. Functional safety is defined as "absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems" [5], and is standardised for road vehicles in the ISO 26262 standard. The first version of ISO 26262 is explicitly limited to vehicles of a gross weight of up to 3.5 tonnes. It is anticipated that the exclusion of heavy vehicles will be removed in the second revision to be initiated around 2015, with publication expected in 2018.

ISO 26262 recommends using fault injection for testing and verification [5]. Fault injection is a testing technique used to validate the dependability of systems [6], [7], [8], [9]. During fault injections, faults are deliberately introduced in a controlled manner into a system in order to observe how the system responds when the error resulting from the fault propagates through the system [7]. The two common uses of fault injection are either to test and evaluate fault handling mechanisms, or to get a measure of the system's dependability. Dependability is specifically important for computer systems where a failure in the system can cause people to die, cause severe injuries or result in a loss of large sums of money.

No tools are available on the commercial market for doing fault injections into AUTOSAR systems. Lanigan and Fuhrman [10] discuss a technique that they used to inject faults into an AUTOSAR system running in a CANoe simulation environment. No other articles were found that look into ways to inject faults into AUTOSAR based systems. Furthermore, as the automotive industry is taking on the AUTOSAR standard and ISO 26262 recommends using fault injections, the need for fault injection tools with support for AUTOSAR based systems is increasing.

1.1. Research question

The objective of this master's thesis is to identify interception points in the AUTOSAR architecture, which could be suitable for injecting faults by using software implemented fault injection (SWIFI). The purpose is to create a tool capable of testing robustness of software components by the use of SWIFI. Software components are building blocks for applications in the AUTOSAR architecture and robustness is defined by IEEE as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [11].

The objective has been broken down into the following research questions:

- Can existing fault injection techniques be used or modified for injecting faults into AUTOSAR application components and/or basic software?
- What are the suitable interfaces and mechanisms in the AUTOSAR architecture that can be used for fault triggering, fault injection and observation of a target system?

In the automotive industry, software components are often delivered by suppliers and the source code may not be available. Hence, in this work we assume that software components are delivered as object code and therefore regarded as black boxes. The focus of the analysis is on the software layers surrounding the run-time environment in the AUTOSAR architecture, in order to find suitable fault injection locations.

The result is validated by a proof-of-concept implementation of a prototype fault injection tool.

1.2. Stakeholders

This thesis is aligned with two major research projects performed at Advanced Technology and Research at Volvo Group Trucks Technology, which act as stakeholders for the thesis outcome. The primary stakeholder is the DEDICATE project, which focuses on techniques for improving fault management. Both in-vehicle solutions and external services are considered, to find new techniques in order to increase the commercial vehicles' reliability. In the DEDICATE project the thesis outcome is intended to be used to test a hierarchal error management concept for the E/E-system of heavy duty trucks. The second stakeholder is the BeSafe project, which is investigating benchmarking of functional safety. The prototype tool should be extendable to be able to perform dependability benchmark measures on AUTOSAR basic software components in the future.

1.3. Structure of this document

This report is structured as follows:

- Chapter 1 gives an introduction to the thesis, the research question and the thesis project stakeholders.
- Chapter 2 describes how the master's thesis has been carried out.
- Chapter 3 gives a short introduction to the dependability terminology that the thesis uses.
- Chapter 4 gives an overview of the AUTOSAR standardised architecture and the error model used by AUTOSAR.
- Chapter 5 contains the result from the literature review and gives a background on different fault injection techniques.

- Chapter 6 contains the results from the analysis on which fault injection techniques are suitable to use in order to inject faults into AUTOSAR based systems.
- Chapter 7 contains a description of the implemented SWIFI tool prototype.
- Chapter 8 contains the evaluation of the implemented SWIFI tool prototype.
- Chapter 9 summarises the current limitations of the SWIFI tool prototype and discuss possible extensions and future work.
- Chapter 10 contains the conclusions drawn from the thesis.

2. RESEARCH METHOD

The research method for this master's thesis was derived from design science research papers. March & Smith [12] describe the main activities in design science to be to 'build' and 'evaluate', and compare the need of justification in natural science to primarily be the explanation to why an artefact works within its environment. The research method, as illustrated in Figure 1, was tailored from Peffers et al. [13] into five phases: problem analysis, literature review, architecture & design, implementation and evaluation.

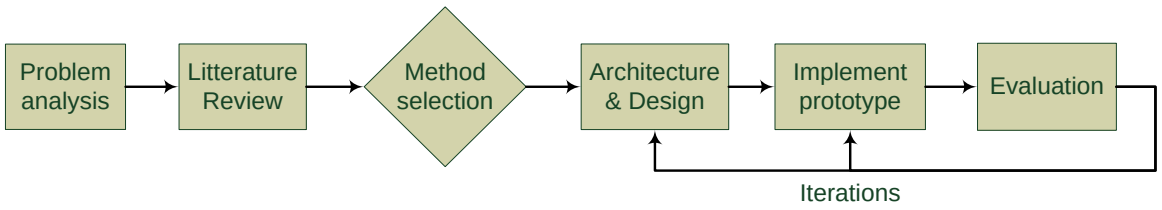


Figure 1: The different phases and main decision point of the research method

First, the problem was defined and analysed in collaboration with stakeholders and supervisors. A literature review was carried out in the form of a background study of fault injection, by studying books and research articles about the subject. The main emphasis was on identification of different techniques already used to perform software implemented fault injections. After that, relevant specifications on the AUTOSAR application layer and interfaces used by application layer software components were reviewed with the goal of finding suitable interception points for fault injection. In the literature review, searching for relevant literature was carried out in Chalmers Library Summon, IEEE Xplore, ACM Digital Library and Google Scholar as well as using references in articles already read (generally known as the snowball technique).

Having identified interfaces which can be used for Software Implemented Fault Injection (SWIFI), we created a list of fault injection techniques that could be implemented in the AUTOSAR context. A prototype was designed and implemented that uses one of the identified fault injection techniques. Finally, the prototype was evaluated for meeting its purpose. The architecture & design, implementation and evaluation phases were iterated in order to elaborate the concept and the prototype.

According to Cleven, Bubler & Hüner [14] the evaluation process has a number of variables for a design science research artefact evaluation. In this thesis, the main artefacts are tools for generation of fault injection mechanisms and for running injection experiments. The choices made when designing the evaluation study are shown (blue markings) in Table 1 below.

Table 1: Method configuration in accordance with [14]

Variable	Value				
Approach	**Qualitative**			Quantitative	
Artefact focus	**Technical**		Organisational		Strategic
Artefact type	Construct	Model	Method	**Instantiation**	Theory
Epistemology	**Positivism**			Interpretivism	
Function	**Knowledge**		Control	Development	Legitimisation
Method	Action research		Case study	Field experiment	Formal proofs
	Controlled experiment		**Prototype**		Survey
Object	**Artefact**			Artefact construction	
Ontology	**Realism**			Nominalism	
Perspective	Economic		Deployment	**Engineering**	Epistemological
Position	Externally			**Internally**	
Reference point	Artefact against research gap		**Artefact against real world**		Research gap against real world
Time	Ex ante			**Ex post**	

The purpose of Table 1 is mainly to make this thesis comparable with other design science research projects. Some of the choices may be obvious, such as having a technical approach and an engineering perspective, since this is a thesis for a master's degree in software engineering. The reasoning behind other choices may need some further description.

We chose to do a qualitative study, as most of the evaluation criteria will not be numerical. Parts of the evaluation have numerical results though, such as the measurement of the time overhead of the chosen fault injection technique. However, the main part of the evaluation was based on the authors' understanding of the suitability of the evaluated fault injection technique, based on a number of defined criteria. We chose the knowledge function since it has been a part of our course and the primary objective is learning.

As part of our collaboration with the company where the thesis was conducted, a prototype tool was requested, and the choices of artefact, instantiation, prototype, and artefact against the real world, were taken together with the supervisor at the company.

It was classified as internal, as we did both the design and evaluation ourselves and ex post, since we designed the prototype first and then evaluated the implementation.

3. DEPENDABILITY TERMINOLOGY

This chapter gives an introduction to the dependability terminology used in this thesis. We use definitions and terminology for dependability and its attributes given by Avižienis et al. [15]. The use of the term system may need some explanation before diving deeper into dependability. The word system is used quite extensively in this text in a broad sense, and can mean anything from a single software component to a vehicle containing several networks of computers with software. This is important in the pathology of failure described later on in this chapter, where the failure of one component in a system becomes the fault at the input of another component.

A definition for dependability given by Avižienis et al. is the "ability of a system to avoid service failures that are more frequent and more severe than is acceptable" [15]. This means that a dependable system is a system where the user (either human or machine) can trust the services provided by the system. The dependability of a system is characterised by a set of attributes, and the most common attributes are:

- *Availability;* readiness for service
- *Reliability;* continuity of correct service
- *Safety;* absence of catastrophic consequences on the user(s) and the environment
- *Integrity;* absence of improper system alterations
- *Maintainability;* ability to undergo modifications and repairs
- *Confidentiality;* absence of unauthorised disclosure of information

During the life-cycle of a system (see [16] for information of life-cycle models), from concept generation to decommissioning, events may occur that introduce faults into the system.

An error is defined by Avižienis et al. as "a deviation of one or more states from its correct value(s)" [15]. A fault is "the cause, either adjudged or hypothesized, behind an error" [15]. If errors propagate so that an external state of the system deviates from its correct service state, it leads to a system service failure. A failure, also called a service failure, is defined as "an event that occurs when the delivered service deviates from correct service" [15]. The relationship between faults, errors and failures is called the "pathology of failure" and its causality chain can be described as being Figure 2, where the failure of one component is the fault of another part in the system.

In the operation phase of the life-cycle, faults may prevent the system from delivering its intended service, if not dealt with properly. However, faults do not necessarily lead to system failure, first the faults need to be activated and result in an error in the software system. Faults in the system that are not (yet) activated are said to be dormant. An error is what happens when, for example, the system executes a software instruction containing a bug. Errors are usually what can be detected during testing and can be seen as symptoms of faults. An error in one part of the program can lead to errors in other parts of the program. This process is called error propagation (Figure 2).

If an error propagates to the system boundary and becomes visible to the environment of the system, this is then called a failure of the system. If the system goes into a degraded mode as a reaction to a fault, it is not necessarily considered a failure, if that is the specified behaviour in that situation.

This leads to a pathology of failure, where a chain of threats with causality relationship acts recursively in a way that errors propagate through the system, which means that several faults must usually be present in order for the system to end up in a service failure.

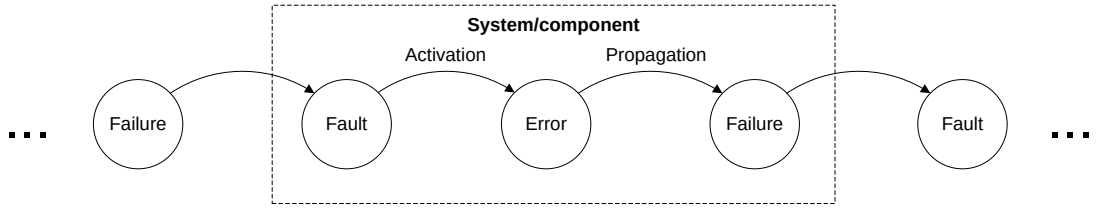


Figure 2: Fault-Error-Failure causality chain [15]

There are methods to achieve and analyse the dependability of a system which are referred to as the "means of dependability". The means are used at different stages in the life-cycle and are usually: Fault Prevention, Fault Tolerance, Fault Removal, and Fault Forecasting.

Fault prevention activities are carried out in the development life-cycle with the aim of preventing the occurrence or introduction of faults. Fault tolerance is a capability built into the product or system to avoid service failures in the presence of faults. Fault removal reduces the number and severity of faults, e.g. by testing and correcting bugs. Fault forecasting involves activities to estimate the present number, the future incidence, and the likely consequences of faults.

According to Avižienis et al. the different means of dependability have slightly different aims. Fault prevention and fault tolerance have "the aim to provide the ability to deliver a service that can be trusted" [15], while fault removal and fault forecasting aim to "reach confidence in that ability by justifying that the functional and the dependability and security specifications are adequate and that the system is likely to meet them" [15].

4. OVERVIEW OF AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is an open industry standard for automotive Electrical/Electronic architectures [3].

4.1. The Virtual Function Bus

One of the key concepts of AUTOSAR is the introduction of a Virtual Functional Bus (VFB). The VFB separates the application software from lower layers. The purpose of the VFB is to make software components (SW-C) independent of the underlying hardware and therefore make it possible to relocate software components to other electronic control units (ECUs) when configuring the system. This implies that a software component does not know, and does not need to know, whether the component it is trying to communicate with resides on the same ECU, or on another ECU on the network.

Figure 3 provides an overview of the concept of the Virtual Function Bus. Software components are building blocks for applications. An application is built up by one or several interconnected software components. Software components are referred to as atomic, since a single software component cannot be distributed over several ECUs. Hence, an application can be distributed by having several interconnected atomic software components where the individual components are located on different electronic control units. In addition to application software components, AUTOSAR also specifies specific sensor and actuator software components. They are independent of the ECU, but will be dependent on the sensor or actuator hardware they are connected to. For performance issues the sensor and actuator components will in most cases be located on the same ECU as the hardware they are dependent on.

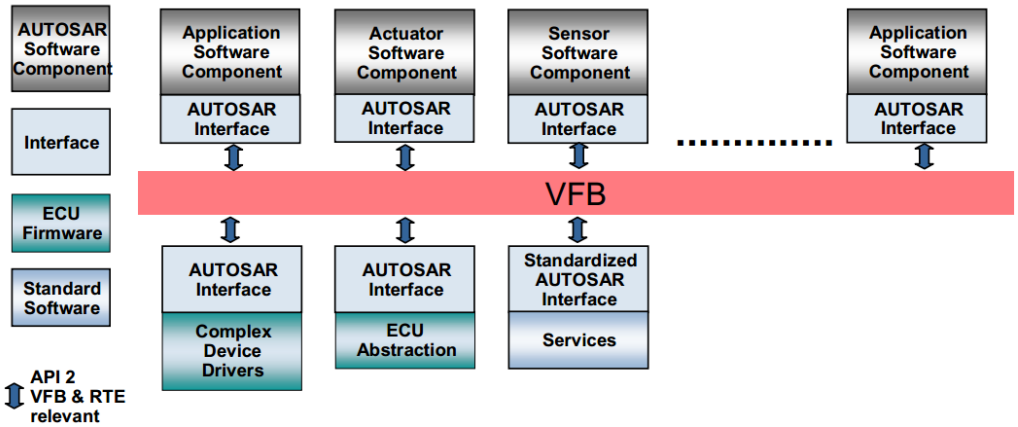


Figure 3: Software Components interconnected via the Virtual Function Bus [3]

Software components have required (input) and provided (output) ports that they use to communicate with each other [17]. All communication between application software components has to take place using ports according to the AUTOSAR standard [17]. Commonly used port type interfaces supported by the VFB are client-server ports and sender-receiver ports. In client-server communication, a server provider port on a SW-C

provides operations to one or more SW-C that have a corresponding client required port. The call is initiated by the client. In sender-receiver communication, the sender initiates the operation and can provide information to several other components. A required receiver port can also be configured to get information from several provided ports.

Figure 4 shows an ECU view of an AUTOSAR system. The implementation of the virtual function bus is called the Run-Time Environment (RTE), which is the layer in the AUTOSAR layered architecture that separates the application layer, from the basic software. The basic software itself also consists of different layers. From the bottom there is a Microcontroller Abstraction Layer (MCAL) which implements specific drivers for the microcontroller and its internal peripherals. Above the MCAL is an ECU abstraction layer which encapsulates the specifics of the ECU with a standardised interface. A services layer offers real-time operating system (RTOS) functionality and services for memory, diagnostics and mode management. The service layer is also the layer that is responsible for the routing of communication signals to other electronic control units. There can also be Complex Device Drivers (CDDs) to handle things not standardised by AUTOSAR, such as legacy software. Complex device drivers can communicate directly with the basic software [17].

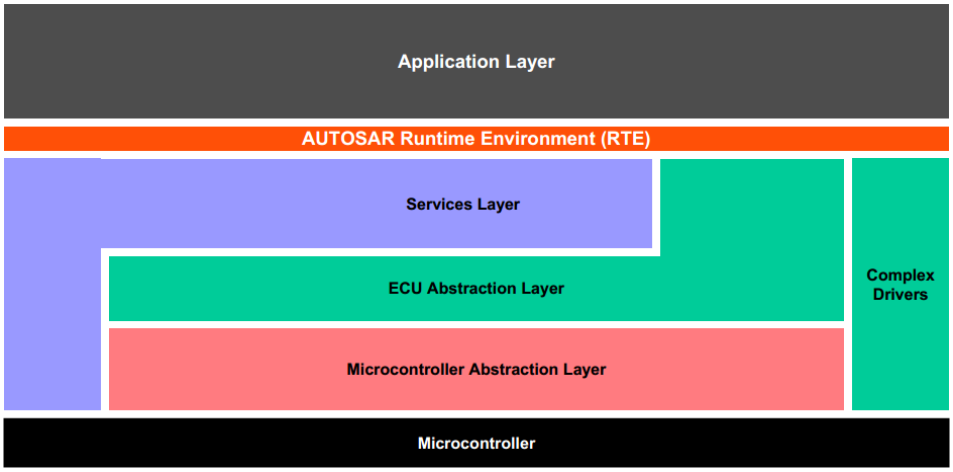


Figure 4: Overview of software layers in AUTOSAR [18]

4.2. RTE Generation

The AUTOSAR specification defines a meta-model using the unified modelling language that is used to describe the platform [19]. A model describing an AUTOSAR system can be mapped to and stored in an XML file, whose schema is defined by the AUTOSAR standard [19]. The AUTOSAR standard specifies that the data can be stored as a single XML file or broken down into several smaller files [19]. The standard also defines how a section in one XML file can reference a different section in another file and how the documents should be merged together when parsed. Different tools used to configure each ECU or generate source code files (such as the RTE) will then read and write the XML files.

The RTE is according to the AUTOSAR specification generated in two phases [20]. In the first phase, the so called contract phase, header files for software components are generated from an ECU XML configuration file and the components' XML internal description files. The RTE generator creates the API needed to send and receive data between different components. In the second generation phase, the remaining RTE source code files are generated.

The naming conventions for the generated functions are standardised by the AUTOSAR standard [20]. Depending on the type of call, the name will consist of the type of call being made, port name, data type and the component's name.

The RTE is also responsible for running the so called "runnables" [20], which are functions implemented in the software components. A runnable can be scheduled to be run periodically or when a specific event occurs in the system.

4.3. Error Handling in AUTOSAR

Error handling in AUTOSAR is described in several dimensions. There are some errors that are handled directly by the AUTOSAR basic software, e.g. errors on the communication link. There are other errors that are described on a conceptual level in order to advise the application developer on how common error handling strategies can be implemented in a software component and be supported by the mechanisms of the AUTOSAR framework. An example of this is voting mechanisms, which need to be implemented by software components by the application developer.

On a detailed level, as described in the "Description of the AUTOSAR standard errors" [21], the error handling support that is built into the basic software in a standardised way, is limited to errors on the CAN network and different types of non-volatile memory faults. This error handling follows a process for fault detection, fault isolation, and fault recovery. This is often referred to as the FDIR process. The errors detected in the application or the basic software are reported to a Diagnostic Event Manager. For recovery the software components can be notified of detected errors in order to initiate their own recovery procedure and there is also a defined healing cycle for the basic software. The standard lists explicitly failure modes that shall be supported and the detection, reaction, report and recovery mechanisms that are expected if applicable.

The application error management [22] defines an error model, which includes the errors described in Table 2. It is called an error model instead of fault model, due to the constraint that only software mechanisms are considered and hence it is errors that are detected. The error model focuses on errors resulting from random external faults. Both transient and permanent faults are considered. The error handling considers mainly operational faults, even though design faults that slip through verification and manifest themselves as operational faults are also handled to some extent.

Table 2: AUTOSAR Error Types on application level, excerpt from [22]

Error Type	Description
Data	A data error is characterised by an erroneous value of a parameter, variable or message. The source of the error can be either internal (e.g., SW defect) or external (e.g., malfunctioning sensor, other faulty SW-Component). Handling of data errors can break a causality chain that would lead to subsequent errors that are more complicated to handle, such as program flow or access violations.
Program Flow	Program flow errors (also "control flow errors") manifest as actual program flows differently than expected, possibly leading to missed, wrong or superfluous operations being carried out. The source of the program flow error can be both internal (SW defects) and external (data errors).

Access	For increased separation between executing components the system designer can partition the SW and restrict access to resources from the partition, e.g., memory access. When a component tries to access a resource in another partition without the proper access rights an access violation occurs. Access errors can be the result of a data or program flow error, e.g., an invalid program counter or pointer.
Timing	A communication (message, function invocation, etc.) is time critical when the delivery time has an effect on the correctness/usefulness of the communication. A timing error can be a message being delivered early, late or missing completely (omission). The last type of timing error, omission, is of special interest and is sometimes referred to as crash or fail-silent behaviour (note that it may be impossible to distinguish between crash, which is an uncontrolled state, and fail-silence, which is a controlled state). Timing errors also refer to execution time, where strict deadlines can be defined on how long a component is allowed access to the CPU.
Asymmetric	When errors propagate from one SW-Component to another using some means of communication one differentiates between symmetric and asymmetric errors. In the symmetric case all receivers receive the same (erroneous) value. When the component can fail by sending different values (all of which may be valid) the error is said to be asymmetric. This error model is sometimes also referred to as the Byzantine model, which implies that no assumptions whatsoever are made on the behaviour of a malfunctioning component. Byzantine errors can only be detected by use of redundant components exchanging values to reach a common result.

AUTOSAR based systems are the target environment studied in this thesis and the error types defined by AUTOSAR is used as reference when selecting a fault injection technique.

5. OVERVIEW OF SWIFI

This chapter gives an overview of Software Implemented Fault Injection (SWIFI) techniques and aims to give a picture of the state-of-the-art within the field.

5.1. Introduction to Fault Injection

Fault injection is a widely researched technique, both Yangyang & Johnson [7] and Voas & McGraw [23] claim that the first published work about fault injection should be credited to Harlam Mill's work at IBM on statistical validation of computer programs done in 1972. According to Voas & McGraw, fault injection can be viewed as a testing technique, although not in a traditional sense. So, in a way, while testing focuses on finding defects that are already there according to a specification of the correct behaviour, fault injection focuses on how the system will behave in a scenario with errors, sometimes under different, currently unknown circumstances [23].

However, given the initial differentiation between testing and fault injection, the main objectives of fault injection is usually either fault removal or fault forecasting [7], [6], [24] [9], [8]. Fault removal aims at observing how a particular fault tolerant design behaves under errors propagating from the injected fault. In this sense, fault injection is used as a testing technique in order to verify if the system meets its specifications and to identify issues. The designer of the experiment needs to have detailed knowledge about the system [7]. The results can then be used to improve the existing design [8]. Fault forecasting, on the other hand, is used to estimate the dependability of a particular component by introducing various faults that the component should tolerate in order to establish a measure of the component's dependability [7] and [8]. In this sense, it is a way to rate the efficiency of the operational behaviour of the dependable system, which aims to quantify the confidence that can be attributed to a system by estimating the number and the consequences of possible faults in the system. Fault forecasting can be either qualitative or quantitative [7]; qualitative fault forecasting involves the activities of identifying, classifying and ordering of failure modes, or to identify the event combinations that may lead to undesired events. Quantitative fault forecasting is about evaluating in probabilistic terms some of the measures of dependability [7]. The two major approaches for doing quantitative fault forecasting are modelling and testing [23].

According to Yangyang & Johnson [7], fault injections can be classified by whether they are implemented as hardware, software, simulation or hybrid fault injections. Hardware fault injection techniques introduce faults through the physical hardware, for example using heavy ion rays to modify the memory of the system under target [6]. Simulation based methods focus on doing a fault injection on a simulation of the system [7]. It has the advantage of being able to test how a design behaves under specific faults early on in the development life-cycle, before the system is implemented [6]. The system model might, however, not properly capture all system properties [7]. In a hybrid approach a combination of hardware and software is used to introduce a fault into the system [7]. The remaining part of this chapter focuses on software implemented fault injections (SWIFI).

SWIFI techniques use software to inject faults into the system. SWIFI techniques are attractive as they can access states in the system that might not be accessible with other hardware based testing techniques, no special hardware is needed, experiments can often be

run in almost real-time, and as the experiments are often being run on the same hardware that will be used in production it will take into account any design problems that might be in either the software or hardware [7].

5.2. Fault Injection Framework

Hsueh et al. [6] describe the typical fault injection environment as consisting of a controller, a fault injector, a workload generator, a monitor, a data collector and a data analyser (see Figure 5). The target is the system into which faults will be injected. It can be a stand-alone component, a fully implemented system or a simulation of a system during pre-design. The injection framework might be distributed and include parts on the actual target system.

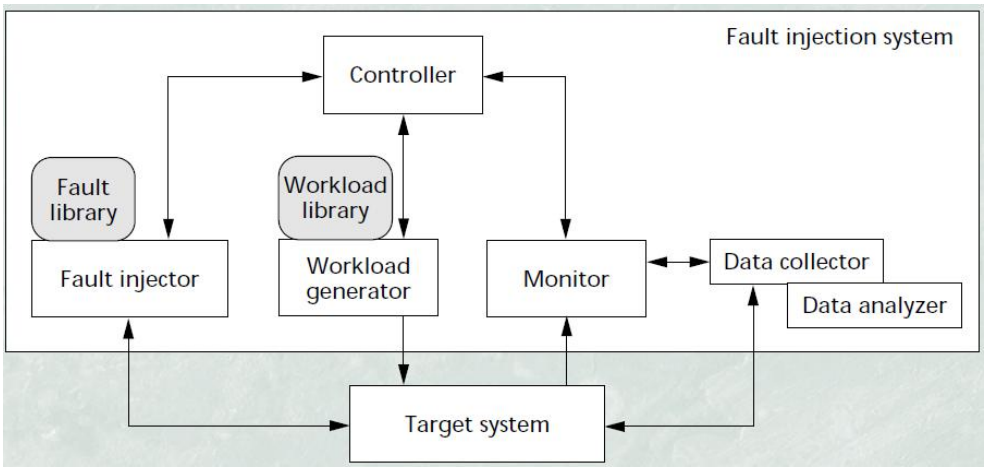


Figure 5: Basic components of a fault injection system according to [6]

The controller is responsible for setting up and controlling an experiment. An experiment is the setup of which fault to inject into the system, including mechanisms to identify when and where to do the injection. Several experiments can be run in a batch, called a campaign.

The fault injector injects fault(s) into the target system. Often the fault injector is split into separate trigger and injection modules (e.g., NFTAPE [24]). After a certain state or event occurs within the targeted system (e.g., a certain memory location is read by the CPU), the trigger instructs the injector to inject the fault. The workload generator creates a workload on the system, which is intended to simulate the load that the system will experience during operation. The monitor and data collector observe and record different events and states on the target during the fault injection experiment [6].

Before carrying out experiments where faults are injected into the system a so called golden run is usually made, where no faults are introduced into the system [7]. This information can then be used to decide where and when to introduce the faults and it can also be used as a comparison to runs where fault injections are done. The data analyser is used to determine the outcome of the experiment. According to Yangyang & Johnson [7] there are three possible outcomes:

1. The fault is covered. This means that the fault is activated, but the error is correctly handled by the system's fault handling mechanism.
2. The fault is not covered. The fault is activated, but the error is not correctly handled by the system.

3. The fault does not lead to a failure in the system. The fault was not activated or the error was masked by the system (e.g. was overwritten by a correct value) before it caused a failure.

5.3. Fault Injection Techniques

Software implemented fault injections can be classified in different ways. Hsueh et al. [6] classifies fault injections depending on whether they are done at (or before) compile-time or during run-time. Compile-time techniques modify source code or assembly code whereas run-time techniques inject the fault during run-time after some fault triggering event occurs. Compile-time techniques are usually used for emulating permanent faults, whereas run-time fault injections are usually used to emulate temporary or transient hardware faults [6].

Another classification made by Stott [24] classifies software implemented fault injection techniques depending on the type of technique used to introduce the fault. The different techniques mentioned by Stott are debugger based, driver based, performance based and target specific fault injections. All these techniques inject faults at run-time and are further described in Section 5.3.2.

5.3.1. Fault Injection Triggers

The fault injection trigger identifies when to inject a fault (depending on time, state or event) and activates the fault injection mechanism. Several different triggering methods can be used to execute a fault injection mechanism during run-time.

Exception based triggers use hardware interrupts to instruct fault injection handlers to inject a fault [6]. One way is to use the debugging features of the CPU to cause an interrupt when a specific event occurs, such as when a specific memory location is read. The Xception fault injection tool can, for example, use break-point registers to inject a fault when fetching an opcode, loading an operand, or storing an operand from a specific memory addresses into the CPU, inject the fault after a certain time has elapsed since start-up or a combination of all [25]. Some processors can also be configured to cause an interrupt when the system load gets too high [24].

Trap instructions inserted into application source code can be used to cause an interrupt to occur when they are executed [6]. An interrupt handler will then inject the fault, as in the case of exception based hardware triggers.

Time based triggers introduce a fault after a specific time-out. One time-out method is to generate an interrupt once a hardware or software timer expires, and a dedicated interrupt handler will then inject a selected fault into the system. The method has the benefit of being simple, but it is only suited for emulating transient or intermittent faults and the injection experiments might not be reproducible [6].

Hsueh et al. also describes code insertion triggers that use instructions added to the application code before compile-time to call fault injectors directly, instead of using interrupts and handlers.

Hexel [9] uses what he calls "hooks" (essentially triggers) to inject faults into time triggered real-time systems. The hooks are inserted into the target system and when executed, do a call-back to the fault injector that introduces the fault. The experiments can be configured before or during run-time. Different hooks that are to be triggered during the experiment can be selected and configured.

The Bond fault injection tool, designed to inject faults into applications running on the Windows NT operating system, uses "interposition agents" to monitor the target application and trigger an injection when certain events occur [26]. The idea is to add a new layer between the target application and the operating system that wraps and intercept systems calls between the two. The benefits are that no modification to the target application is needed [26]. Different events, such as the n^{th} count of before or after a certain API call, or an access to a particular memory location can be used as the triggering event.

5.3.2. Fault Injectors

The injection involves the actual modification of code, signal or hardware element, such as the memory or a register in order to create a fault or error in the system. The notion of fault injection is used even though it is often errors that are actually injected. However, due to the pathology of faults described in Chapter 3, where a failure of one component becomes the fault of another and that the intention is to emulate the effects of faults, it is usually more general to refer to it as fault injections.

5.3.2.1. Compile-time Fault Injection

Compile-time fault injections (or code modification) are good for emulating permanent faults, such as software bugs or permanent hardware faults. One method described by Durães and Madeira [27] to emulate software bugs is to modify the executable binary by first converting it into assembly code and then use pattern matching to modify and insert new code depending on the type of fault that is to be introduced.

Code modification at compile time can also be used to emulate permanent hardware faults by adding or replacing assembly instructions [28]. For example by overwriting register content in the middle of a program execution to emulate a register fault [28].

5.3.2.2. Debugger-based Fault Injection

Debugger based methods use debugging features to write into parts of memory [24]. The method is used to access arbitrary parts of the memory (heap, stack, code segment) and there is no need to modify the fault injection target or the operating system. The method has been used to inject faults into random or specific locations, for example to simulate ion radiation in space or to target specific parts of a system [24].

The injection handling routine can be written as a system interrupt handler that gets executed after an interrupt is triggered. The source of the interrupt can either originate from a software trap or from a hardware event. This was, for example, one of the methods used to inject faults into HARTS, a distributed real-time system [28]. The interrupt handling fault injection process was given the highest scheduling priority to quickly inject the fault and then give the control back to the application that executed the software trap.

5.3.2.3. Target Specific Fault Injection

Target specific fault injection is when extra source code is inserted into the application that then gets executed at some point during the fault injection run [24]. This method requires knowledge and access to the source code, but it is good for manipulating specific data structures where random memory modifications are not adequate [24].

Lanigan and Fuhrman [10] use a method which they claim is inspired by Hexel [9], to inject faults using hooks into an AUTOSAR application running in a CANoe simulation environment. In [10], no clear distinction is made between fault triggers and injectors, but

two types of hooks are defined that are inserted into several AUTOSAR API system calls to inject the faults or cause an erroneous behaviour:

- Suppression hooks are used to abort specific AUTOSAR API calls and depending on the method called, return an error code to the caller. Note that this method does not directly inject a fault, but emulates an erroneous behaviour, as if a fault had been activated in another component by the call.
- Manipulation hooks are used to manipulate specific data structures such as different signal messages communication fields.

Communication protocols usually consist of several layers, where each layer is responsible for taking care of some aspect of the communication and deliver services to the layer above [29]. The DOCTOR fault injection tool uses a fault injection protocol layer to inject communication faults [28]. The fault injection layer can be placed anywhere in the protocol stack (including directly under the target application) and is completely transparent to other protocol layers. The layer receives commands from an external module. The fault injection layer can pass messages without modification. It can also target specific messages and discard them, delay them or modify specific fields within the intercepted message.

5.3.2.4. Performance based Fault Injection

Performance based fault injection is based on exhausting available resources on the system [24]. This could for example be to open multiple files or network sockets without closing them, creating a deadlock, or taking up excessive memory on the system.

5.3.2.5. Driver based Fault Injection

For some operating environments, such as Linux or Windows, consideration needs to be taken to the privileges of the current process or task. For instance, fault injection code that runs in user space on a Linux machine might not have the privileges needed to access or take up resources required to carry out the injection. There are solutions to this problem though, and one way around this restriction is to write the module as a device driver that will run in system mode and thus have more access rights [24]. An interface is made to the driver so that user space programs can trigger the injection.

Even if the fault injection code runs in user space and therefore does not have direct access to the hardware, there might still be an indirect way to inject faults into locations that are not directly accessible. The Bond fault injection tool developed for use on the Microsoft NT operating system can, for example, inject faults into what is called the thread context [26]. The thread context contains the processor's state for a particular thread, including a copy of all registers [26]. A similar approach is used by Tsai and Jewett [30], where a copy of the registers saved in memory (e.g., saved while doing a method call) are corrupted and then the corrupted register copies are loaded back on the processor. Another way of overcoming hardware access limitations is to emulate the behaviour of the error instead of injecting the fault directly into the hardware [28]. Tsai and Jewett used a test portion of a SCSI driver to emulate disk I/O errors on Tandem machines [30]. The test interface made it possible to set a flag that would activate a specific error handler at the next driver request.

5.3.2.6. Robustness Testing

Robustness is defined in IEEE Std. 610.12.1990 as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [11]. Miller, Fredriksen & So [31] and Forrester & Miller [32] have described the Fuzz tool that can test specific operating system elements and interfaces by injecting random data.

The Ballista robustness testing method [33] tests combinations of valid and invalid input parameters. The test cases, often referred to as "dirty tests", use parameters which are taken from a pool of normal and exceptional values based on the value type of each input parameter.

5.3.3. Monitoring the Fault Injection Experiment

In order to determine the outcome of the fault injection run (covered, uncovered or no impact from the fault), it is necessary to monitor the system, and collect and analyse how the target system behaved during the experiment. One option is to use monitoring facilities already in use by the system, if available. This could be events logged by the operating system, such as kernel messages and general error logs, or events monitored and logged by the application. The benefit is that monitoring is already integrated into the system. They might however not always give the granularity of details needed to properly analyse and determine the outcome of the experiment.

Often the fault injection environment contains its own mechanisms to monitor the target. One method is to run the experiment in debug mode in order to get a full system trace containing a detailed history of all system calls, memory addresses read, and other state information [7]. This might however be impractical for real-time systems, as running the system in trace mode can introduce high overheads [24].

The same methods used for triggering a fault injection are sometimes used to monitor the target system. For example, the interposition agent used in the Bond fault injection tool is also used for monitoring the application. The agent can monitor debug events from the kernel, API calls and memory accesses [26].

The MAFALDA fault injector tool, developed for use on real-time microkernel based systems uses a set of interceptors to observe the target system [34]. Events intercepted include scheduler events, results from tasks, signals, termination and return code of system calls.

5.3.4. Avoiding Intrusiveness

Intrusiveness refers to how much undesirable effects the fault injection has on the target system [35]. An example of intrusiveness is when excessive memory is used as parts of the fault injection environment resides and executes on the target system.

Systems used in the automotive industry often have hard real-time constraints, so it is important that no timing constraints on the target system get violated by the fault injection mechanisms. A method used by the MAFALDA tool, while injecting faults into a real-time interrupt driven system, is to disconnect all interrupts (both internal and external devices) [34]. As the notion of time on such systems is built from internal interrupts, it has the effect of "freezing the progression of time" [34]. Once the fault injection tool has finished executing, the interrupts are allowed to resume.

Another method used by Hexel [9] on a time-triggered real-time system is to try to minimise the fault injection interference, using a combination of different strategies. Parts of the fault injection environment are situated on the target system for better performance. The fault injection is configured before or during system run-time so that the only parts that are executed during an experiment are the hook (trigger) and the fault injector. The fault injector is implemented as a call-back from the hook. Finally, the hooks are placed and designed in such a way that the external timing of the target system remains unaffected.

Sometimes, the monitoring of the target system while conducting the fault injection can cause undesirable effects; for example, a fault injection free "golden run" of the system is sometimes done in trace mode to record all system activity, in order to find a suitable location to do fault injection [7]. Running the system in trace mode will, however, greatly reduce system performance as discussed by Stott et al. [24]. One compromise could be to do a number of experiments where the faults are randomly injected into some memory address range and the experiments where no faults are activated are discarded [8].

5.4. Evaluating Different Fault Injection Techniques

According to Arlat et al. [35], if two different techniques produce the same sets of behaviours, then the methods can be considered to be equivalent and other properties should be taken into consideration when selecting between them. Methods that produce different behaviours are considered to be complementary. Arlat et al. provides a (non-exhaustive) list of properties that might be taken into consideration when selecting between equivalent methods:

- Reachability is defined as the ability to reach possible fault locations on the target system. One method might, for example, only reach parts of the memory, while another method might reach both the memory and parts of the processor.
- Controllability is defined both with regards to time and space. The space dimension regards how much control the method has over injecting faults into specific reachable locations on the target. One method could for example only corrupt random parts of a memory region while another method might be able to specify exactly what parts of memory to corrupt. The time dimension regards controlling the instance of when the fault is injected.
- Repeatability refers to being able to repeat an experiment exactly or at least very similarly as before. Using, for example, a time-out to trigger a fault injection on a non-deterministic operating system might have less repeatability than triggering a fault injection when a certain system call is made by the target application. Events on the target system might occur in a different order between two test runs and some events might not even occur in one of the runs.
- Reproducibility means that when an experiment is run more than once, the same or very similar results are obtained.
- Intrusiveness refers to how much undesirable effect the fault injection has on the target system. Further discussion of this property can be found in Section 5.3.4.
- Time measurements relate to obtaining detailed timing information for different monitored events, while the experiment is being run.
- Efficacy refers to the technique's ability to produce significant results from the fault injection experiment. That means that the fault injection produces an observable behaviour that is either covered or not by the target system.

These properties can be used to evaluate different fault injection techniques and are used in Chapter 8 to evaluate the implemented fault injection tool.

6. SWIFI FOR AUTOSAR

We analysed the AUTOSAR specifications with the goal of finding ways to intercept and inject faults into calls between application software components (SW-C) and the Run-Time Environment (RTE). The objective is to test robustness of applications consisting of one or more SW-C. Most of the focus was on the RTE specification [20] and parts of other AUTOSAR standards. Additionally, we reviewed the automatically generated RTE source code from one vendor.

The remaining parts of this chapter will describe techniques applicable to AUTOSAR and specifics required to target function calls consistently. Finally, we discuss which technique was chosen to be implemented in the prototype tool.

6.1. Intercepting RTE Calls

Compile time techniques where the software component binary or source code is modified could be used to emulate permanent faults in the component. The drawback of this technique is that a time consuming build will have to be made between each fault injection experiment.

Using debugger or hardware based fault injection could be done with low intrusion on the target system. It is an interesting technique but it was estimated that implementation could not be done within the parameters of a master's thesis and therefore it was ruled out.

Techniques where additional code or a new layer is added to the target system seem most applicable for capturing calls between the SW-C and the RTE. The concept of adding a new layer in order to do a fault injection has been used both in the Bond and the DOCTOR fault injection tools. The MAFALDA tool uses "interceptors" to capture calls to and from the target which is the same or a similar concept. The AUTOSAR architecture uses a layered structure with well-defined interfaces between the layers. It should therefore be possible to add a new layer into the architecture in order to read and write into data that is passed between the layers.

In the automotive industry, SW-Cs might be delivered as object code from a vendor. Adding extra source code into the SW-C might therefore not be possible. However, all data going to and from a SW-C must pass through the RTE and the RTE layer needs to be custom generated by an RTE generation tool for each ECU. The entire RTE is therefore available as source code before the complete system gets compiled, linked and downloaded to an ECU. Extra source code instructions can therefore be added to the RTE with the purpose of monitoring, triggering or injecting a fault into the system.

We found three main approaches for intercepting calls between the RTE and software components:

- One approach is to create a new layer, a wrapper, which is situated between a software component and the RTE.
- Another approach is to use trace hooks, which are already in place and specified as part of the AUTOSAR standard specification [20].
- As a third approach, the RTE can be modified using code-insertion before compilation to include fault triggering, injection and monitoring capabilities.

6.1.1. Application of Wrappers

A wrapper is an additional layer that is situated between a component or a part of a component (e.g. a function) and its environment. The same or a similar concept is used in the Bond [26], MAFALDA [34] and the DOCTOR [28] fault injection tools as discussed in Chapter 5.

Adding a wrapper around software components has the benefits that both the function parameters and return data can be modified for all calls made between the software components and the RTE. A wrapper can also force API calls to return immediately with an arbitrary value and thus emulate an erroneous behaviour.

Fault injection layers for basic software components residing beneath the RTE layer can be manually created as their interfaces are standardised. Wrappers for application software components will however have to be created individually for each component, as they have a unique interface to and from the RTE.

Since all selected interface data would pass through a wrapper, they are suitable for use in triggering, injecting as well as monitoring purposes.

Before an application component or parts of it can be wrapped, its interface will have to be extracted. Three possible approaches were identified for finding interfaces on a SW-C, in order to generate an application software component wrapper:

- Manually inspect and add code to wrap a software component. This technique is good for testing the concept and developing the method, as a prototype can be built fast. This approach does, however, not scale well due to the work involved in adding the wrapper code manually.
- Automating the generation of SW-C wrappers using available RTE header files. The component's header file(s) can be parsed to find both API prototypes used by the component and its runnables that are run by the RTE layer. The AUTOSAR standard defines exactly how different RTE function calls should be named and the API names should, therefore, be vendor independent. Some information can be extracted from the standardised function names as it includes the type of call being made, the SW-C name, the port name, and data types.
- Automate the wrapper generation using standard AUTOSAR XML configuration files. As all the needed interface information, such as SW-C port names, data types and port operation types is stored in the XML, it can be used to generate a wrapper.

6.1.2. Application of Trace Hooks

Another way to inject faults into AUTOSAR SWC's is to use trace hooks. Trace hooks can be inserted into all communication ports defined in the RTE and at various other locations in the RTE and the basic software. A trace hook needs to be enabled before it can be used. Trace hooks are called in the code that defines SW-C ports as shown in the code snippet in Figure 6. However, the definition of the trace hook function should be user implemented according to the standard and is thus suitable for triggering, monitoring, and in some cases injecting faults.

For each explicit API call to and from the RTE layer, a trace hook is placed at the beginning of the call and then another hook is placed at the end of the call [20]. The hooks will take the same parameter as the API call, but the parameters are not always passed as reference, which makes it harder to use them for fault injections. However, for calls made for external ECU communication, a pointer to the signal data (and in some cases the signal's length) is passed to the trace hook function, and the signal can thus be modified inside the trace hook.

As hooks get called inside an RTE API function, they cannot be used to directly force a return or modify the API return value.

```
Std_ReturnType Rte_Write_Component_Port_Element(SInt32 data)
{
   Std_ReturnType ret = OK;
   WriteHook_Component_Port_Element_Start(data);
   ComHook_Signal(&data);
   ret |= Com_SendSignal(&data);
   WriteHook_Component_Port_Element_Return(data);
   return ret;
}
```

Figure 6: Example of an implementation of a SW-C port send operation in the RTE layer, including Trace hooks

6.1.3. Application of RTE Modification

It would be possible to modify the generated RTE source code before compilation to include fault injection mechanisms. Lanigan and Fuhrman [10] used code modification and placed fault injection call-back hooks into an AUTOSAR based system running in a CANoe simulation environment. However, no emphasis was made in [10] on how to automate the process of adding fault injection instructions into the target system source code. The purpose of this study is to build a tool, and it is therefore important to automate the process. One way to automate the code insertions would be to use the fact that the standard defines exactly the trace-hook symbol name placed at the beginning and end of each RTE API call (one hook for start and one hook for end), to insert extra source code as shown in the code snippet in Figure 7.

```
# define WriteHook_Component_Port_Element_Start(data) \
WriteHook_Component_Port_Element_Start(data);         \
if(inject_fault_now == 1)                             \
{                                                     \
    data = fault_value;                               \
}
```

Figure 7: Example of a macro that can be used to target trace hooks in order to add fault injection instructions

If the macro in Figure 7 is placed in any header file included by the Rte.c source file, then the compiler pre-processor will modify the example from Figure 6 to the one shown in Figure 8.

```
Std_ReturnType Rte_Write_Component_Port_Element(SInt32 data)
{
   Std_ReturnType ret = OK;
   WriteHook_Component_Port_Element_Start(data);
   if(inject_fault_now == 1)
   {
    data = fault_value;
   }
   ComHook_Signal(&data);
   ret |= Com_SendSignal(&data);
   WriteHook_Component_Port_Element_Return(data);
   return ret;
}
```

Figure 8: Implementation of a SW-C port with fault injection instructions added with the macro in Figure 7

6.2. Compatibility Mode Prerequisite

The generation of the RTE software layer is according to the AUTOSAR standard performed in two steps [20]. In the first step, the so called "contract phase", the RTE generator creates an application header file for each individual software component. In the second RTE generation step, the generator creates the rest of the RTE source code.

If the RTE generator has access to the application component source code it can optimise the code by using macros instead of function calls for inter-component communication in the cases where components are located on the same ECU. The first problem with this, from a fault injection perspective, is that the optimisation is vendor specific and thus not standardised. This will make auto-generation of fault injection mechanisms tailored to the RTE more difficult and vendor specific. The second problem is that targeting function calls is easier than targeting code that has been in-lined using macros because the functions can be wrapped and they will contain trace-hooks that can be used for fault injection purposes.

If the configuration uses application components that have been compiled into object code (or thinks that it will be using object code), then no optimisation can be done and the generator will run in "compatibility mode", so that all RTE API calls will be implemented as functions, and not in-lined by using macros.

The downside of using "compatibility mode" is that the application will have an extra overhead by doing additional function calls, in addition to whatever overheads the fault injection mechanisms will introduce to the target.

6.3. Technique Selection

Wrappers were chosen to be implemented and evaluated based on a discussion of the findings with experts on the subject. Wrappers were chosen because they have been successfully used to do fault injections in other environments (e.g. Bond, MAFALDA, DOCTOR tools) and they have the benefits that both the function's parameters and return data can be modified for all calls made between the software components and the RTE. A wrapper can also force API calls to return immediately with an arbitrary value and thus emulate an erroneous behaviour. The first priority is to wrap application software components since their interfaces are configurable and therefore more complex to generate

wrappers for. The technique selected is considered portable to basic software components as well, but wrapping basic software components was scoped out to do in addition to wrapping SW-C.

Trace hooks are an interesting finding and in a way they are natural inception points already present in the AUTOSAR standard, but they are scoped out to implement in addition to wrappers due to resource constraints in the master's thesis project. RTE modification using macros was also considered, but the technique was also ruled out for the same reasons.

We have also chosen to call the fault injection mechanisms directly inside the wrappers, instead of using trap instructions that would indirectly cause an interrupt handler to inject the fault. The reason is that an interrupt will cause a context switch to occur which might add some additional overhead, over calling the fault injection mechanisms directly.

7. PROTOTYPE TOOL

The fault injection tool prototype is described in this chapter. The purpose of the tool is to facilitate robustness testing of application software components. The tool generates wrappers, as discussed in Section 6.1.1 and 6.3, which intercept all calls between the component under test and the Run-Time Environment (RTE).

The prototype is developed to run on the Windows XP platform and uses the Microsoft .NET framework version 4. It consists of several components which can be divided into two major parts: the configurator and the campaign runner. This is illustrated on the left hand side of Figure 9. The configurator and code generator provide the user with a graphical user interface to configure an experiment with regards to setting triggers, connecting faults to triggers and configuring monitors prior to performing an experiment or campaign.
The campaign runner controls the fault injection experiment and consists of a PC based application for running campaigns and experiments and also a controller located on the target in order to get the best possible real-time performance. During a campaign the campaign runner communicates with the embedded controller via a dedicated CAN channel.

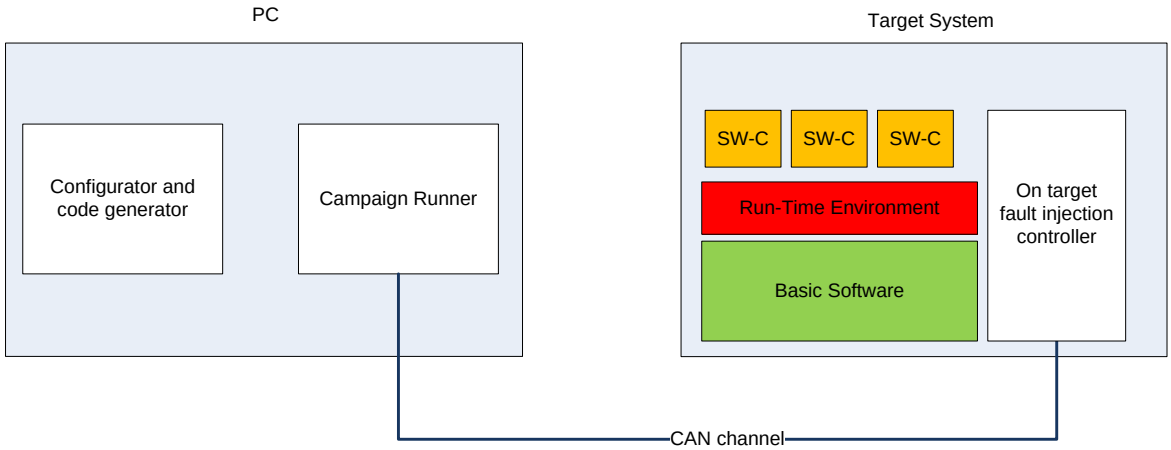


Figure 9: Fault injection tool overview

7.1. Fault Injection Support

The purpose of the tool is to support the error model used by AUTOSAR as described in Section 4.3. In the scope of this thesis, support for injection of data errors has been implemented. The main reason for focusing on data errors is that robustness testing is testing components with invalid inputs which imply the injection of data errors. The concept is not limited to data errors though, so the tool can be extended with support for other error types in the future.

Calls made between an application software component and the RTE were targeted. The purpose of this is to test error handling mechanisms in the software component (or a composition of software components) and to observe how the software component behaves upon erroneous inputs. In the calls made between the component and the RTE, the fault injection tool can access and modify function parameters and return values. The fault

injector has configurable values to overwrite a parameter or a return value upon injection. It can also force a function call to return immediately, with a return value selected by the fault injection experiment designer. When forcing a return, it can be configured whether the actual call shall be executed or not. In some cases the experiment designer wants to test what happens if the call is dropped without notifying the caller, and in some cases the call is cancelled with a user specified return status code.

An error can be injected once (emulation of a transient fault) or every time the targeted function is called (emulation of a permanent fault). The concept can easily be extended to include more logic to simulate intermittent faults or randomised fault injection.

7.2. Trigger Support

Triggers are used in the fault injection experiment to determine when to inject a fault. Triggers are described generally in Section 5.3.1. In the tool a trigger must first be selected in the configurator in order to generate code for the triggers in the wrapper. The following types of triggers are implemented in the prototype:

- Trigger when an RTE API function or runnable for a component has been called a configurable number of times.
- Trigger when a parameter or a return value in an RTE API function is higher than, lower than, or equal to a configurable user value.

Triggers selected for an experiment must be enabled before the experiment is started. Each enabled trigger will also have to have one or more faults connected to it that will be activated after an event in the system activates the trigger. Any trigger can be used to activate any fault configured in the system, regardless of where they are located.

7.3. Monitor Support

The prototype tool supports different data and events to be monitored, including all interface parameters to software components. Monitors can be configured in the configurator but need to be enabled prior to running an experiment. In this way parameters of RTE API function calls can be monitored, i.e. required and provided interfaces of software components can be monitored synchronised with the fault injection. The monitor data is passed over the CAN channel to the campaign runner where it is logged.

The fault injection controller also supports the CAN Calibration Protocol (CCP) [36] which means that a logger can be setup to monitor any address in memory. However, CCP support is currently not built into the campaign runner.

7.4. Using an Embedded Controller

One part of the fault injection controller is located on the target in order to get better accuracy for triggers, injectors and monitors with regard to real-time requirements. Having a fault injection controller residing on an embedded real-time system has been used before by for example Hexel [9]. In our tool, the embedded controller is implemented as an AUTOSAR complex device driver [3] in order to have access to hardware resources on the target, cause minimal intrusion on the software architecture for the system under test, and also for it to be easily portable to other test targets. We consider it to cause low intrusion from a design perspective as the tool is implemented using its own CAN channel for communication with the fault injection tool. This means that the signal mapping for the original software in the ECU does not have to be changed and all that is needed is an invocation of the complex device driver. It requires that a CAN channel is free on the

microprocessor and that it is connected and accessible on the circuit board. Figure 10 describes the internal structure of the on-target fault injection controller and how it is located in the AUTOSAR system.

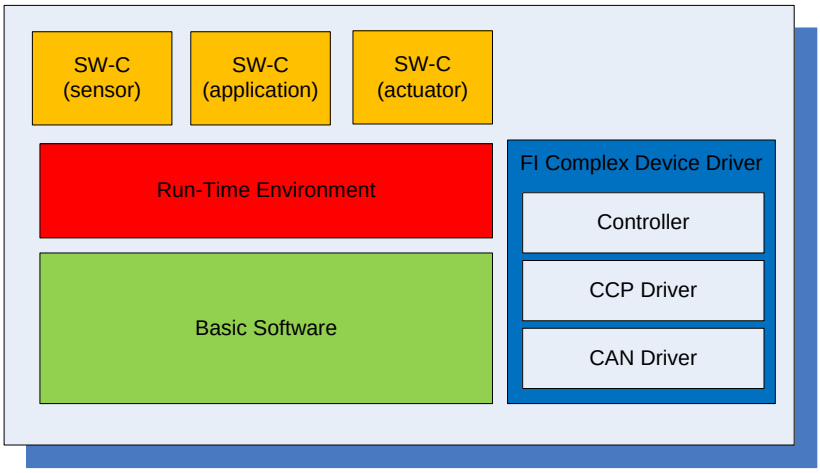


Figure 10: In-target fault injection Controller, implemented as an AUTOSAR Complex Device Driver

The internal structure is built up by a layered structure in order to be easily portable to other microprocessor architectures. The configuration uses a proprietary extension of the ASAM MCD Can Calibration Protocol (CCP) [36]. The CCP is used since it already provides a standardised way to measure internal variables in an embedded system. The standard has been extended with commands to setup the fault injection controller to perform fault injection experiments.

7.5. Process Overview

Figure 11 shows a diagram of the complete process for configuring and running fault injection experiments. The rectangular boxes represent automated steps implemented in the fault injection tool and rounded shapes represent steps currently performed manually. The first step involves the code generator, done on a PC before the RTE is compiled, in which the experiment designer selects which functions and fault mechanisms (monitors, triggers and faults) should be supported. In the next step, after the target has been built with fault injection features selected in the code generator, a campaign consisting of one or more experiments is generated. During a campaign run, each experiment configuration is sent from the PC over a dedicated CAN channel to the target system, where the experiment is configured by a fault injection controller that resides on the target before the experiment is started. Finally, a message to start the experiment is sent to the fault injection controller and information configured to be collected during the experiment is sent by the controller over the CAN channel to the PC host.

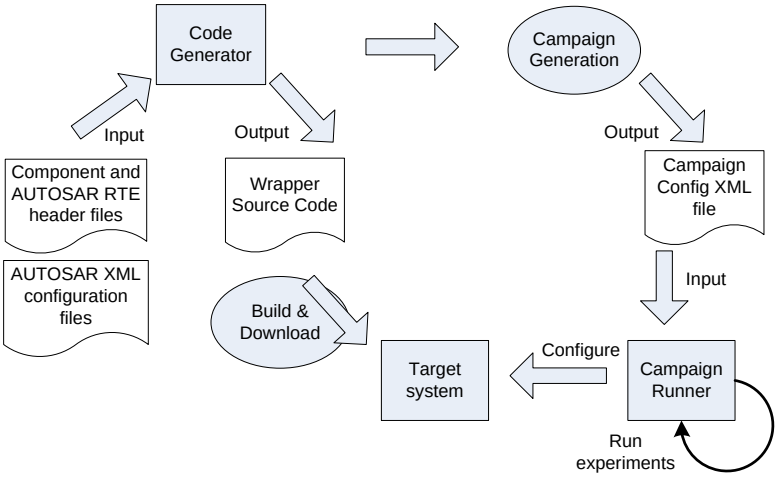


Figure 11: Complete fault injection tool chain

7.5.1. Code Generation Configuration

Instead of wrapping all functions and providing support for monitoring all events, triggering conditions and supported fault types, the experiment designer can select what supported features should be built into the target system. This is to optionally make the intrusion footprint as small as possible.

One extreme would be to select every feature supported and generate fault injection mechanisms for all components. The benefit would be that the experiment designer only has to build the target system once, as all supported fault injection mechanisms are already included. The downside is that selecting all supported fault injection mechanisms will have considerable overheads on the real-time system.

The other extreme would be to select only what to target for each experiment (i.e. target one particular function parameter). This will have far less impact on the target, but instead a time consuming build will have to be made between each experiment. This trade off will have to be decided by the experiment designer depending on factors such as the systems real-time constraints and features being tested.

Figure 12 shows the code generator's graphical user interface (GUI). In the settings, the experiment designer can select an ECU to target by selecting the appropriate AUTOSAR XML configuration files and RTE headers. Once the relevant XML and header files have been parsed, the experiment designer can browse the ECU composition and select different types of monitors, triggers and faults to be generated.

Once all required fault injection features have been selected, the wrapper source code for the selected features is automatically generated and will be built into the target when it is compiled.

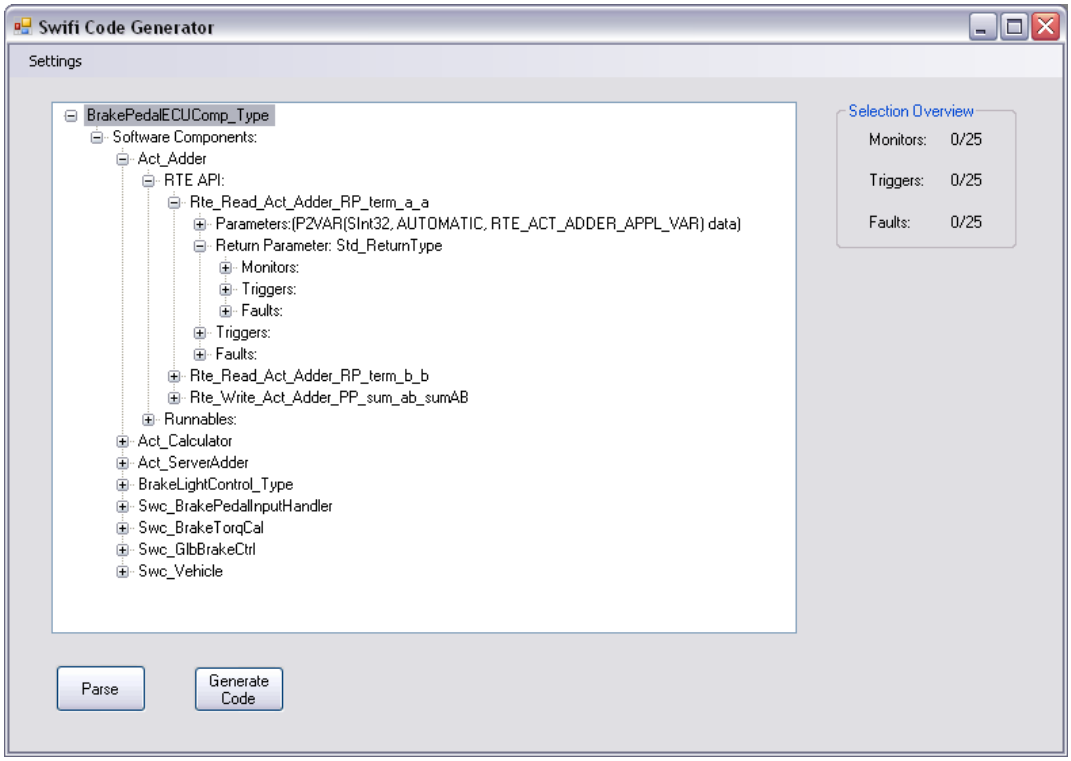


Figure 12: Code Generator GUI

7.5.2. Wrapper Generation

The RTE is generated in two phases as discussed in Section 4.2. The wrapper for the selected application software component is generated after the second RTE generation phase and before the system is compiled. How the wrapper is inserted was designed using the AUTOSAR standards, so the concept should be vendor independent. The targeted software components can be delivered as compiled object code or source code.

The source code includes-dependency diagram in Figure 13 shows how the generated wrapper source files connect to standard AUTOSAR RTE generated files and the targeted SW-C code files.

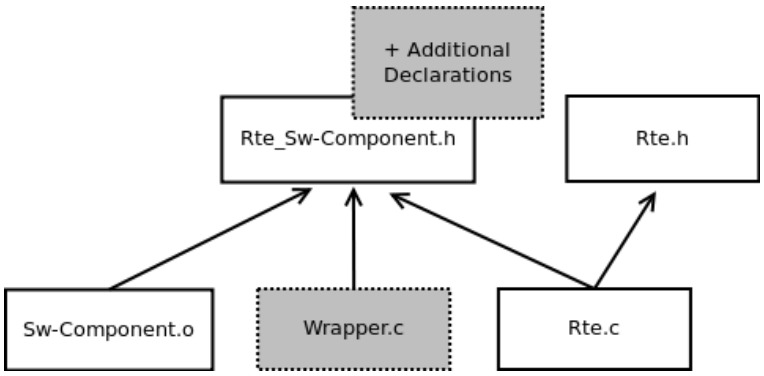


Figure 13: Compile-time modification to insert a wrapper around a software component

The RTE SW-C header file for each software component needs to be modified to include definitions for wrapper functions that encapsulate the original functions and some pre-processing instructions. Additionally, a source code file is generated for each SW-C that was selected to have fault injection capabilities. The automatically generated source code implements a new wrapper function for each RTE API call or runnable that was selected

during configuration of the code generator. Each wrapper function can contain; monitor, trigger and fault injection capabilities before or after the function gets called. Figure 14 shows how a wrapper is situated between a SW-C and the RTE.

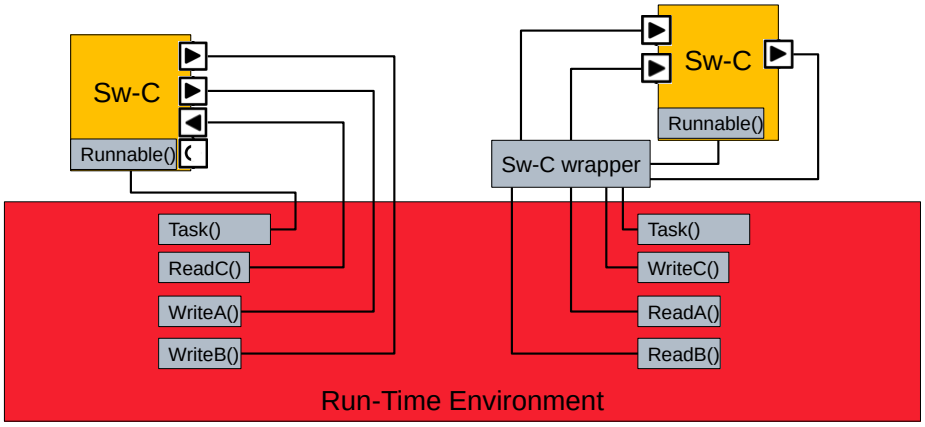


Figure 14: Introducing wrappers to intercept calls between software components and the RTE

7.5.3. Running a Campaign

The campaign runner tool was implemented using the DFEAT fault injection framework. The framework was developed by the DEDICATE project. The campaign runner uses a configuration file that contains sections for configuring monitors, triggers and faults used for each experiment in the campaign.

Figure 15 shows a screenshot of the campaign runner GUI. For each experiment in the campaign, the campaign runner sends configuration messages to the fault injection controller that resides on the target.

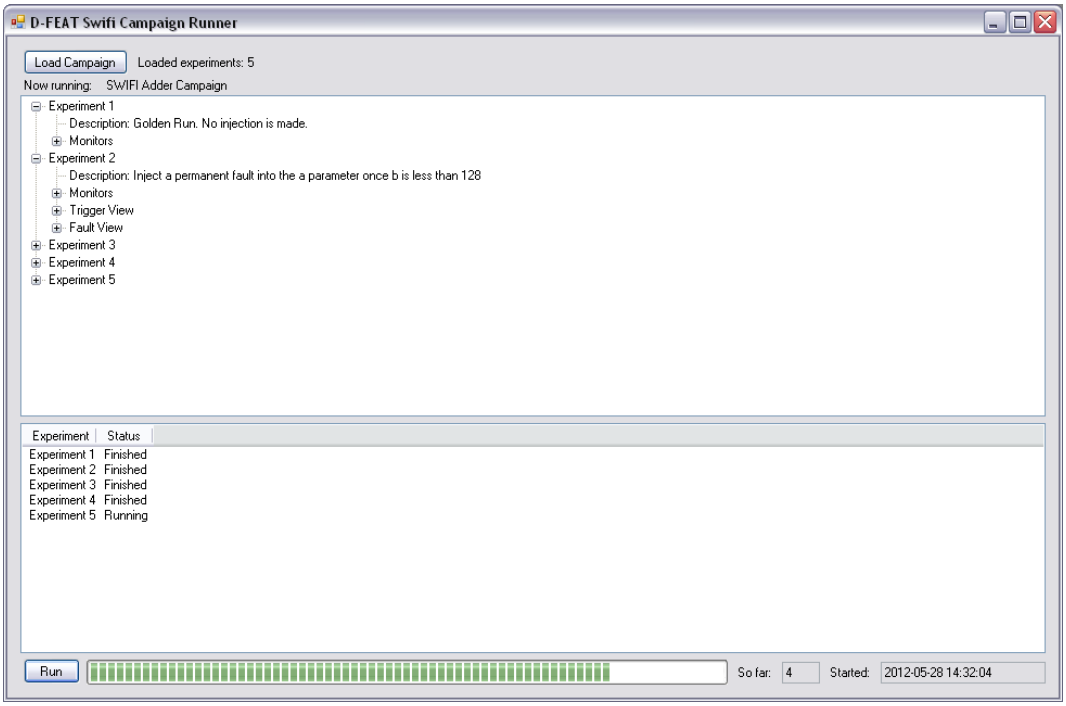


Figure 15: Campaign Runner GUI

The configuration of the fault injection experiment requires that a specific sequence of configuration commands is performed (see Figure 16), in order to ensure consistency in the relationship between triggers and faults. Configuring the experiment prior to running it is

an architectural trade-off that was made in order to have less intrusion on the target. The wrapper is designed to have the fewest amount of statements possible in-line with the execution of the software component under test. Instead a little freedom is taken to setup data structures in a configuration state prior to running the experiment.

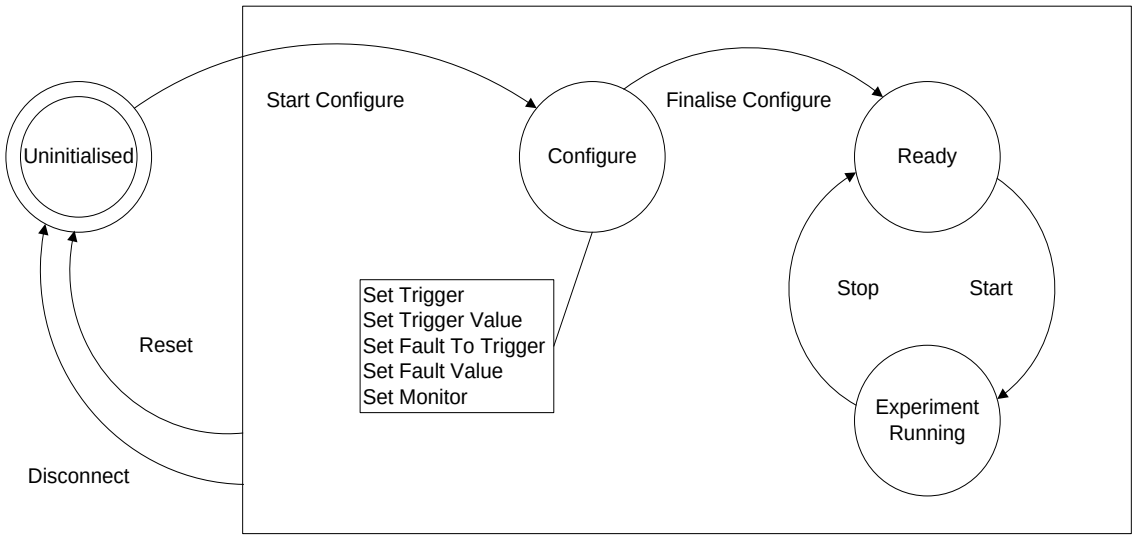


Figure 16: State-machine for the embedded fault injection controller

An experiment always starts with a reset request sent from the campaign runner. This is to ensure that no residues from previous fault injections remain in the system and that the target system always has the correct initial state. The state of the fault injection controller is always 'Uninitialised' after target power-up, illustrated in Figure 16 with the double circles. The target controller requires a command to be put into configuration mode, where all configuration commands are accepted by the embedded controller. After finalising the configuration the experiment can be started and stopped. At any time the fault injection tool can send a command to reset the target, which will force the target into a soft reset and all configuration must be done once again.

Monitors need to be configured in the configuration state but can then be enabled prior to starting an experiment, and as soon as a monitor is enabled its monitoring events are sent over the CAN bus to the campaign runner. Triggers and fault injections cannot be activated unless the experiment is started. Events are logged relative to the real-time clock on the target. The following log events are currently supported:

- Fault injection event; logs the real-time clock counter value for when a fault was injected
- Trigger event; logs the real-time clock counter value for when a trigger was activated
- Monitor event; real-time clock and value for a specific monitor
- Start experiment; the target time for receiving start experiment command
- Stop experiment; the target time for receiving stop experiment command

The purpose of storing events based on the target real-time clock is to be able to analyse events in the experiment using the same time base.

8. PROTOTYPE EVALUATION

The prototype was evaluated by doing fault injections on two different AUTOSAR applications. Both target systems were developed by members of the DEDICATE project team.

The first application is a calculator residing on a single ECU. The second application is a brake-by-wire (BBW) system residing on several ECUs, with fault handling capabilities. “By-wire” is an industry term for when traditional components such as brake, steering or throttle control, which has been implemented as mechanically or pneumatically controlled systems are replaced with electronics. In a brake-by-wire the driver intent is sent electronically from the brake pedal (sensor) to the brake actuator, possibly also via a control unit for algorithms for anti-lock braking and stability control.

The following section describes the experimental setup and the fault injection campaigns run for the two systems. Then we describe the performance measurements performed in order to evaluate the overhead that the fault injection mechanisms has on the target. Finally, the prototype is evaluated against the criteria described in Section 5.4.

8.1. Test Environment

A test environment was setup with the goal of getting the best possible visibility on the entire flow of a fault injection experiment. The same setup, as illustrated in Figure 17, was used for both applications. The target system was monitored using an in-circuit debugger making it possible to stop the target system at breakpoints and view the content of variables and registers. CANoe from Vector was used for monitoring the communication between the campaign runner and the embedded fault injection controller on the dedicated CAN channel. CANoe was also used for setting the system into correct mode for the experiments and to unit test the embedded controller. The campaign runner was run in debug mode in Microsoft Visual Studio making it possible to set breakpoints and view the content of variables there as well.

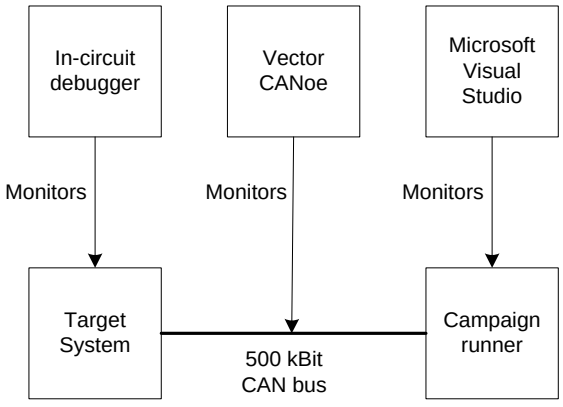


Figure 17: Overview of the test setup environment

8.2. Fault Injection Validation

This section describes fault injection campaigns run for the calculator application and the brake-by-wire application.

8.2.1. Calculator Application

Figure 18 describes the calculator application used to evaluate the fault injection tool. The calculator is a simple AUTOSAR application consisting of two software components, a calculator component and an adder component. The purpose of designing the calculator was to have a simple application with the complete behaviour easily understandable in order to facilitate seeing all effects the fault injection caused.

The calculator implements the logic: $sum = a + b$.

A and b is sent to an adder function which returns the sum. The adder software component is then wrapped and faults are injected into a and b in order to validate the method of intercepting RTE API calls.

As seen in Figure 18, the calculator has two provided ports using the sender-receiver communication pattern and one required sender-receiver port for getting the sum back. The calculator also implements a client-server pattern for the addition. Which communication pattern to use is controlled via the required port 'mode'.

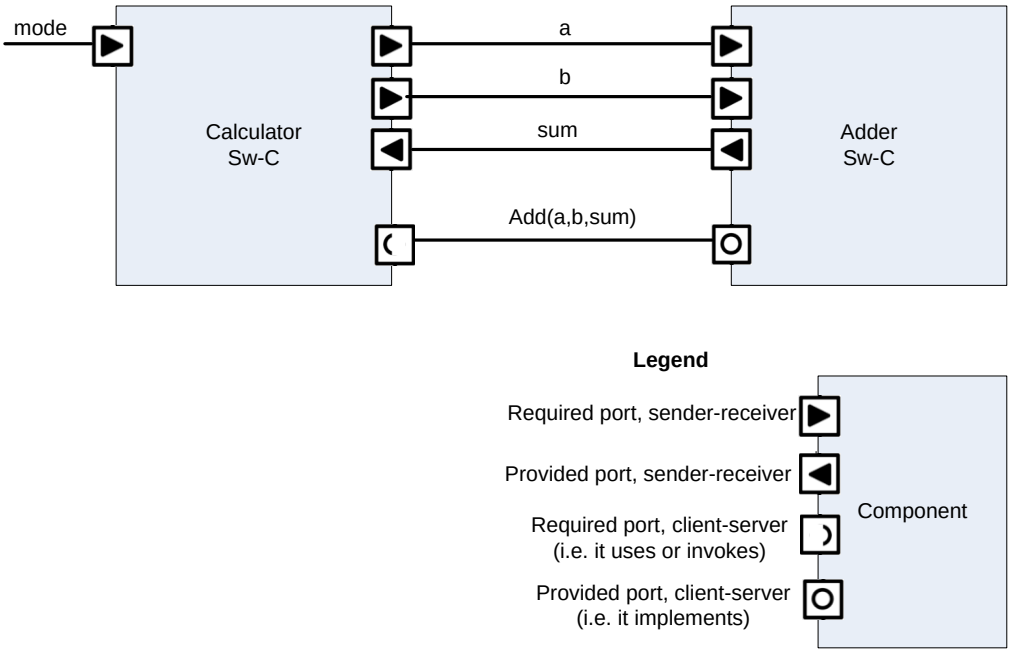


Figure 18: Overview of the Calculator application.

The calculator application was designed to do the same operations periodically. The runnable function that implements the calculator component functionality is configured to be run every 100 milliseconds by the RTE layer. The runnable that implements the adder component functionality is also configured to be run at an interval of 100 milliseconds. The components were configured like they had been delivered as object code in order to force the RTE generator to implement all port operations as standardised AUTOSAR RTE function calls, as described in Section 6.2.

An environment configuration, that simulates the other functions in the vehicle and the necessary environment, was developed in order to set the ECU in normal operation mode. The same environment mode was also used for the brake-by-wire system.

We created a fault injection campaign consisting of eight experiments for the calculator application, in order to test different implemented features in the tool. The first experiment in the campaign is a golden run where some parameters were monitored, but no fault injections were made. All other experiments were configured to inject an error into the target when their triggering condition was met.

Features validated in the calculator experiments were:

- The monitoring functionality
- Trigger when a function has been called a number of times
- Trigger when a parameter is lower than a certain value
- Permanent and a transient fault where a parameter or a return value is over written by a new value
- Connecting a single fault to a trigger
- Connecting two faults to a single trigger at the same time
- Function parameters were read and written both before and after the target function got called inside the wrapper
- The function return value was read and written to after the target function got called inside the wrapper (before the value was returned to the original caller)
- Both sender and receiver ports were targeted
- A runnable was targeted
- Function calls were forced to return immediately

Additionally, we ran a campaign consisting of 1000 identical experiments in order to test the stability of the tool.

The monitored output was sent from the fault injection controller residing on the embedded target system to the PC host where they were logged and manually analysed.

8.2.2. Brake-by-wire Application

In order to validate that the fault injection concept works in a complex environment the tool prototype was also validated on a brake-by-wire (BBW) system. The brake-by-wire system is a research framework developed by the DEDICATE project that implements a brake-by-wire function distributed over five ECUs. The purpose of the BBW-system is to provide a real-world-like example of a distributed safety-critical system for validating research projects. Figure 19 gives an overview of the five nodes in the BBW-system, including the distribution of software components. The BBW-system also incorporates an environment model of the vehicle in order to simulate the behaviour of the entire vehicle with regards to acceleration and braking. In the BBW-system, the driver's intent for braking is read by the BrakePedalECU and then transformed to a brake force request sent to each individual wheel node.

In the BrakePedalECU, the sensor software component reading the brake pedal (BrakePedal sensor SW-C) was wrapped in order to emulate an open circuit fault in the brake pedal's electrical wiring. The detection of an open circuit fault activates the error handling mechanism in the BBW and causes an error reaction in the system. The error reaction was monitored in order to validate the fault injection tool's capability of emulating the open circuit fault.

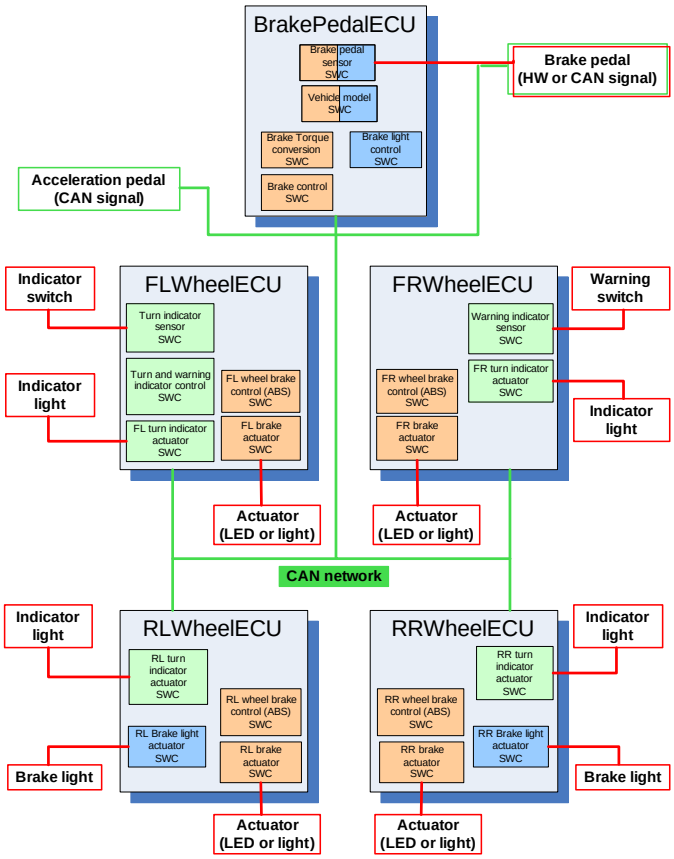


Figure 19: Overview of BBW-system including distribution of application software components [37]

Features validated in the BBW experiments were:

- That the wrapping generation concept scales to handle code generation with many software components in the configuration files
- That the tool and SWIFI technique can be used to emulate electrical faults. The validation showed that injecting a fault in the in the BrakePedal sensor SW-C could cause the exact behaviour of physically disconnecting the brake pedal connector which causes an open circuit fault in the BrakePedalECU
- Wrapping ports of client-server communication pattern as complementary to sender-receiver ports
- That we could trigger the hierarchal error handling mechanism developed in the DEDICATE project and by using the monitors of the fault injection tool, we could follow error conditions and reactions in the BBW system

8.3. Intrusion on Target System

In order to get an understanding on how much intrusion on the target the wrapper generates we performed execution time measurements for a specific RTE call in the adder software component. We measured on a sender-receiver call, and performed measurements in vendor mode, compatibility mode, with an empty wrapper and finally a wrapper with one trigger, one permanent fault and one monitor.

All measurements were made inside the software component using the same measurement code. We used the real-time clock for this measurement. The accuracy of the real-time clock is ±2 µs, which may be too course for this type of measurement. More accurate measures can be done by using one of the special purpose registers, and adding the measurements as in-line assembler code. However, measurements using the real time clock

gave adequate measures to estimate the overhead. Each measurement is an average of 100 samples. The result from the measurements is shown in Table 3.

Table 3: Measurement of time overhead caused by wrapper

No.	Measurement description	Exec. time [µs]		
		Min	Average	Max
1	Without wrapper, original version, with optimisations in the RTE	31	33	33
2	Without wrapper, RTE generated in compatibility mode	35	35	37
3	With wrapper, no content in wrapper	37	37	39
4	With wrapper, 1 trigger, 1 permanent fault and 1 monitor	47	47	49

The measured time to do an RTE API call went from a maximum time of 33 µs for the original vendor optimised call, to a maximum of 49 µs for a wrapped call containing one trigger, one fault injector and one monitor. Several factors contribute to the overhead. First, the API call will have to be made using a function instead of a macro. This will result in an additional function call for each call. Second, for each call that will be wrapped an additional function call is made. Finally, the monitor, trigger and fault injector cause an overhead to the RTE API call. The consequences of this overhead will vary depending on the task that executes the RTE call. According to Hexel [9], care should be taken that the "external timing" of the system is not affected by the fault injection. AUTOSAR SW-C runnables are manually configured to run in tasks that are run by the RTE periodically or when some events occur in the system. A task runs one or more runnables. If there is enough available time in the task that runs the runnable where the RTE API call is made, so that no deadlines are broken, then the overhead should not matter. If there is not enough time in the task, then some real-time constraints might get violated. This could alter the outcome of the fault injection experiment, compared to the same experimental setup where no constraints are broken.

The embedded fault injection controller introduces an overhead into the system each time it is run. The overall overhead will depend on how long it takes to execute the fault injection controller during an experiment and the frequency of the task that runs the controller. The number of enabled monitors will also affect the runtime of the controller, as every time the controller is run, it will send two messages on the CAN network for each enabled monitor. There is also a limit on the number of monitors that can be enabled at a time. There is a static circular buffer that stores monitoring messages and it has a limited size. If there are too many enabled monitors writing into the buffer or if the frequency of the task running the wrapped component is much higher than the frequency of the task running the controller, then older values might get overwritten in the buffer before they are sent out on the CAN network. The reason for having a circular buffer is that it will not crash the target system if it becomes overloaded by too many monitor events.

8.4. Portability

When it comes to portability, we looked at how easy it would be to add support for a different AUTOSAR vendor. We also evaluated how portable the fault injection controller is to different types of ECU. The wrapper code generator reads AUTOSAR XML files and

RTE header files. The AUTOSAR standard does not specify how or if the XML files should be broken down, but standardises how an element in one file can reference elements in other files [19].

The current prototype implementation does not combine different AUTOSAR XML files together. It reads different information from different files depending on how the current supported vendor chooses to split information into different files. We looked at XML files delivered from a different vendor to see if we could parse the files with the wrapper code generator. Unfortunately, the information was split differently by the two vendors. One vendor, for example, delivered software component descriptions as different XML files for each software component, while the other vendor bundled all software component descriptions together in a single XML file.

As mentioned in Section 7.4, the fault injection controller was implemented as an AUTOSAR complex device driver. The only hardware dependent parts are the CAN network driver, the ECU reset command, the reading of the real-time clock and the definition of standard types. The architecture of the complex device driver fault controller was designed to make hardware dependent components easy to replace.

8.5. Reachability

Reachability is defined as the ability to reach possible fault locations on the target system [35]. The tool emulates errors that can propagate from faults originating at different parts of the system. Here, we therefore look at locations where an error can be injected.

The prototype can read and write into function parameters before or after the function is called inside the wrapper. The return value can be read or written after the targeted function call returns. Returning immediately can also be done before or after the targeted function is called to emulate different faults.

If the RTE function call is made to write data, then the error should in most cases be injected before the target function is called in order to modify the data for the receiver. The exception would be when the goal is to modify a pointer to point at an invalid location instead of the actual data that it points to. For read operations the injection should in most cases be made once the targeted function returns.

In the thesis work it is assumed that the user of the fault injection tool does not have access to the source code of software components, therefore the software components are regarded as black boxes and the fault injection tool cannot reach inside the software components.

8.6. Controllability

Controllability can refer to both time and space [35]. The space dimension concerns how much control the method has over injecting faults into specific reachable parts of the target system. The tool has much controllability when it comes to the space dimension as it can accurately read all reachable data and potentially modify it exactly as wanted, given that the corresponding fault type is implemented. If we look at the time dimension as the triggering condition, instead of as physical time, then the tool provides good controllability. A triggering condition can be defined on the state of the return and parameter values or when the function has been called a certain number of times.

Currently it is not possible to trigger on the state of combined parameters and return values. It is for example not possible to trigger when one parameter is higher than some set value and the return value has some other specified value. We however believe that the tool could be extended to support such features if needed.

Support for triggering after a certain time has elapsed is not currently implemented. It should however be easy to add a trigger that activates a fault once a certain time interval has elapsed using the real-time system clock. One simple scenario would be to activate a fault after a certain time period has elapsed since the start of the experiment.

8.7. Repeatability

Repeatability refers to being able to repeat the experiment exactly or similarly as before [35]. The fault injection concept has good repeatability since triggers, injectors and monitors are controlled by the wrapper, which means that the flow of data to and from the component is controlled by the wrapper. However, it does depend on the intent of the experiment and the determinism of the target system. In automotive systems the CAN bus is event based and subject for arbitration which will cause a jitter that might affect the sequence of events in the systems, which limits the repeatability of experiments. Also, start experiment is initiated via CAN and is affected by the jitter on the CAN bus.

8.8. Reproducibility

Reproducibility means that when the experiment is run more than once, the same or very similar results are obtained [35]. In this thesis work there has not been a focus on running a number of experiments that can give statistical significance.

8.9. Time measurements

Time measurements refer to the ability to get detailed timing information from different monitored events while the experiment is being run [35]. The monitors feature was setup to log monitors and other events, such as start and stop experiments, in an event log implemented as a static circular buffer in the embedded fault injection controller. Events are logged relative to the real-time clock counter value. Monitors inside a wrapper can record an event at nearly the same time that a trigger or a fault is activated.

8.10. Efficacy

Efficacy refers to the ability to produce significant results from the fault injection experiment [35]. In all cases where an error was injected in the experiments we got measurable results from the impact of the error on the target system.

9. DISCUSSION AND FUTURE WORK

In this chapter we summarise the current limitations of the tool and discuss possible extensions and future work.

The tool is designed to do robustness testing of application software components, and has a focus on injection of data errors, that are defined in the AUTOSAR standard [22], and also mentioned in Section 4.3.

Wrappers are well suited for injecting data errors in the interfaces of application software components. The tool can be extended to also use wrappers to inject timing errors. The tool could block for a specific time in the wrapper, before or after the targeted function is called, or delay signals with the use of a circular buffer.

The wrapper technique is not very well suited to inject program flow errors, since it regards the software components as black boxes. Since a wrapper has no access to the components internals, it cannot change the internal flow in the component. The exception would be for program flow errors caused by a data error injected into the software component's interface. In order to have better support for program flow errors, the tool could be extended to support other fault injection techniques, such as debugger-based fault injection.

The error source that later propagates into an access error can be a data error [22]. In most cases, the data that is to be passed from one software component to another gets passed as a pointer to the function and then the return value is a standard status code to let the caller know if the operation was successful or not. Some forms of access errors could be produced by the tool by altering the pointer to point to an inaccessible partition.

Further improvements can be made on the prototype in order for the tool to get the maturity of a product. For instance, database support for storing results from experiments can be added and the GUIs of the different components can be merged into a single interface. Campaigns are currently created manually by editing a campaign XML configuration file. Ideally, this step would be done using a graphical user interface.

As described in Section 8.4, the wrapper code generator is not vendor independent. The wrapper code generator can be improved to be vendor independent by extending it to combine different XML files together according to the AUTOSAR standard.

AUTOSAR software components have special runnable "init" functions that are only called on system start-up. The current wrapper implementation cannot target init functions, since the embedded controller always starts in the uninitialized state, and has to be configured before an experiment can be run. The tool needs further development to support injecting faults into these functions.

The prototype can monitor, trigger on and inject faults into variables (e.g. a parameter variable) that are 32 bits or smaller in size. This was sufficient for the experiments that we conducted, but this limitation will become an issue when larger data types are targeted. The reason for this constraint is that CAN messages are limited to a maximum data payload of 8 bytes. The current implementation of the tool uses 4 bytes for the data and 4 bytes for other information needed for the message. Support for breaking up large data structures and then send them using multiple CAN messages is a possible extension but this would also increase the intrusion on the target system.

As mentioned in Section 7.4, the embedded fault injection controller supports the CAN Calibration Protocol (CCP). CCP can be used to both read and write into memory locations on the ECU. It would be interesting to see if CCP can be used for fault injection purposes.

Support would have to be built into the campaign runner that communicates with the fault injection controller in order to instruct what memory locations should be read or written to.

It is possible to extend the tool to support fault injection into calls between basic software (BSW) components. BSW components and their interfaces are standardised, so the current wrapping technique can be extended to support robustness testing of BSW components. Large proportions of the monitoring, triggering and injection mechanisms can be re-used. As the fault injection controller is implemented using an AUTOSAR complex device driver it can communicate with all layers in the BSW. The fault injection controller and the campaign runner could therefore be re-used to setup fault injection experiments targeting BSW components.

10. CONCLUSION

This thesis describes the design and implementation of a fault injection tool for robustness testing of AUTOSAR application software components, using software implemented fault injection (SWIFI).

The fault injection tool uses a wrapper, an extra layer introduced between the component and the Run-Time Environment (RTE), to trigger fault injections, inject faults and to monitor the ports of software components during experiments. We investigated two other techniques that could be used instead of wrappers. One technique is to make use of trace hooks placed in the RTE as standardised by the AUTOSAR standard. The other technique uses code modification of the RTE source code.

The prototype consists of different parts that are used for creating and running fault injection experiments. A configurator and code generator configures and generates the wrappers based on which triggers, faults and monitors the user has selected to enable. A campaign runner performs the actual fault injections. For each fault injection experiment the campaign runner will restart the target system in order to begin each experiment in the same initial state. The campaign runner will then configure what monitors, triggers and faults to use and then start the experiment. The campaign runner uses a fault injection controller embedded on the target system, which is implemented as an AUTOSAR complex device driver. The use of a complex device driver is a good way of having a fault injection controller embedded on the target in order to get real-time performance, and without causing too much intrusion on the design process of the target system. It also facilitates porting of the tool to other hardware platforms and other basic software vendors.

Furthermore, the tool generates wrappers automatically based on the AUTOSAR XML and RTE header files, and can with some extension become independent of basic software vendors.

The tool was evaluated by injecting faults into the interfaces of two different applications. The first application is a calculator residing on a single electronic control unit, and the second application is a brake-by-wire application distributed over several embedded control units with fault handling capabilities. It was shown in the validation that the tool can emulate hardware faults by causing the same reactions in the brake-by-wire system as an open circuit fault.

REFERENCES

[1] A. Biagosch, S. Knupfer, P. Radtke, U. Näher and A. E. Zielke, "Automotive Electronics - Managing Innovations on the Road," McKinsey, N/A, 2005.

[2] Volvo Group, "Volvo Group Global - Our Values," 2012. [Online]. Available: http://www.volvogroup.com/group/global/en-gb/volvo%20group/ourvalues/Pages/volvovalues.aspx. [Accessed 22 4 2012].

[3] AUTOSAR, "AUTOSAR Technical Overview v2.2.2," AUTOSAR, Munich, 2011a.

[4] AUTOSAR, "AUTOSAR Basics," 2012. [Online]. Available: http://autosar.org/index.php?p=1&up=0&uup=0&uuup=0. [Accessed 23 April 2012].

[5] ISO, "International standard ISO 26262 - Road vehicles — Functional safety," ISO, Geneva, 2011.

[6] M.-C. Hsueh, T. K. Tsai and R. K. Iyer, "Fault Injection Techniques and Tools," *IEEE Computer*, pp. 75-82, April 1997.

[7] Y. Yangyang and B. W. Johnson, "FAULT INJECTION TECHNIQUES - A Perspective on the State of Research," in *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, A. Benso and P. Prinetto, Eds., Dordrecht, Kluwer Academic Publishers, 2003, pp. 7-39.

[8] R. Barbosa and J. Karlsson, "Experiences from Verifying a Partitioning Kernel Using Fault Injection," in *12th European Workshop on Dependable Computing, EWDC 2009*, Toulouse, 2009.

[9] R. Hexel, "FITS – A Fault Injection Architecture for Time-Triggered Systems," *Australasian Computer Science Conference (ACSC '04),* vol. 16, no. 1, p. 333–338, 2003.

[10] P. E. Lanigan and T. E. Fuhrman, "Experiences with a CANoe-based Fault Injection Framework for AUTOSAR," *In Proceedings, IEEE/IFIP International Conference on Dependable Systems and Networks*, vol. IEEE Computer Society, p. 569—574, 2010.

[11] IEEE, IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, New York: IEEE, 1990.

[12] S. T. March and G. F. Smith, "Design and natural science research on information technology," in *Decision Support Systems 15*, Minneapolis, Elsevier Science B.V., 1995, pp. 251-266.

[13] K. Peffers, T. Tuunanen, M. A. Rothenberger and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45-78, 2007.

[14] A. Cleven, P. Gubler and K. M. Hüner, "Design Alternatives for the Evaluation of Design Science Research Artifacts," in *Proceedings of 4th International Conference on Design Science Research in Information Systems and Technology*, New York, 2009.

[15] A. Avižienis, J.-C. Laprie, B. Randell and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, vol. 1, no. 1, pp. 11-33, 2004.

[16] N. Storey, "Lifecyckle models," in *Safety-Critical Computer Systems*, Harlow, Pearson Education Limited, 1996, pp. 82-88.

[17] AUTOSAR, "Virtual Functional Bus," AUTOSAR, Munich, 2011c.

[18] AUTOSAR, "AUTOSAR Layered Software Architecture," AUTOSAR, Munich, 2011b.

[19] AUTOSAR, "Model Persistence Rules for XML," AUTOSAR, Munich, 2010b.

[20] AUTOSAR, "Specification of RTE," AUTOSAR, Munich, 2010a.

[21] AUTOSAR, "Description of the AUTOSAR standard errors," AUTOSAR, Munich, 2009a.

[22] AUTOSAR, “Explanation of Error Handling on Application Level,” AUTOSAR, Munich, 2009b.

[23] J. M. Voas and G. McGraw, Software Fault Injection - Inoculating Programs Against Errors, New York: John Wiley & Sons, Inc, 1998, pp. 5-6.

[24] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk and R. K. Iyer, “NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors,” Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Urbana, 1999.

[25] D. Costa, H. Madeira, J. Carreira and J. G. Silva, “XCEPTION™ : A Software Implemented Fault Injection Tool,” in *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, A. Benso and P. Prinetto, Eds., Dordrecht, Kluwer Academic Publishers, 2003, pp. 125-139.

[26] A. Baldini, A. Benso and P. Prinetto, ““BOND”: An Agents-Based Fault Injector For Windows NT,” in *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, A. Benso and P. Prinetto, , Eds., Dordrecht, Kluwer Academic Publishers, 2003, pp. 111-123.

[27] J. A. Durães and H. S. Madeira, “Emulation of Software Faults: A Field Data Study and a Practical Approach,” *IEEE Transactions on Software Engineering,* vol. 32, no. 11, pp. 849-867, 2006.

[28] S. Han, K. G. Shin and H. A. Rosenberg, “DOCTOR: An IntegrateD Software Fault InjeCTiOn EnviRonment for Distributed Real-time Systems,” *IEEE International Computer Performance and Dependability Symposium,* pp. 204-213, 1995.

[29] A. S. Tanenbaum and M. Van Steen, Distributed Systems: Principles and Paradigms, 2nd ed., Upper Saddle River: Pearson Education Inc., 2007.

[30] T. K. Tsai and D. Jewett, “An Approach towards Benchmarking of Fault-Tolerant Commercial Systems,” in *Proceedings of Annual Symposium on Fault Tolerant Computing*, Sendai, 1996.

[31] B. P. Miller, L. Fredriksen and B. So, “An Empirical Study of the Reliability of UNIX Utilities,” *Communications of the ACM,* vol. 33, no. 12, pp. 32-44, 1990.

[32] J. E. Forrester and B. P. Miller, “An Empirical Study of the Robustness of Windows NT Applications Using Random Testing,” in *4th USENIX Windows Systems Symposium*, Seattle, 2000.

[33] P. Koopman, K. DeVale and J. DeVale, “Interface Robustness Testing: Experiences and Lessons Learned from the Ballista Project,” in *Dependability Benchmarking for Computer Systems*, K. Kanoun and L. Spainhower, Eds., Hoboken, John Wiley & Sons, Inc., 2008, pp. 201-226.

[34] J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, “MAFALDA: A Series of prototype tools for the assessment of real time COTS Micro-kernel based systems,” in *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, A. Benso and P. Prinetto, Eds., Dordrecht, Kluwer Academic Publishers, 2003, pp. 141-156.

[35] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs and G. H. Leber, “Comparison of Physical and Software-Implemented Fault Injection Techniques,” *IEEE Transactions on Computers,* vol. 52, no. 9, pp. 1115-1133, 2003.

[36] ASAM, “ASAM - Association for Standardisation of Automation and Measuring Systems,” 2011. [Online]. Available: http://www.asam.net/. [Accessed 04 May 2012].

[37] Volvo Group Trucks Technology, “DEDICATE framework description, Deliverable 2.4,” Company internal, Gothenburg, 2012.