TECHNISCHE
UNIVERSITÄT
DARMSTADT

Assessing and Enhancing Functional Safety Mechanisms for Safety-Critical Software Systems

VOM FACHBEREICH INFORMATIK
DER TECHNISCHEN UNIVERSITÄT DARMSTADT
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
EINES DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

DIPL.-ING. THORSTEN PIPER

GEBOREN AM

05. SEPTEMBER 1979 IN HEIDELBERG

REFERENT: PROF. NEERAJ SURI, PH.D.
KORREFERENT: PROF. DR.-ING. HABIL. ROMAN OBERMAISSER

TAG DER EINREICHUNG: 29. APRIL 2015
TAG DER MÜNDLICHEN PRÜFUNG: 12. JUNI 2015

D17
DARMSTADT 2015

ERKLÄRUNG

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 29. April 2015

Thorsten Piper

The best car safety device
is a rear-view mirror
with a cop in it.

— Dudley Moore.

ACKNOWLEDGMENT

First and foremost, I would like to thank my advisor Neeraj Suri for building and leading a research group, in which each member can pursue and develop his own research interests individually. Thank you for providing us this freedom and for guiding me through the time of my PhD endavour. I am also very grateful to Roman Obermaisser for accepting to be my external reviewer, and to Andy Schürr, Reiner Hähnle, and Matthias Hollick for being on my committee.

I would like to thank all of my colleagues for the intense and intensive discussions that we had throughout the years. Especially Stefan and Oliver had a large share in successfully shaping my research and in fueling new ideas after work, thank you! A special thanks to my partner in crime Daniel for the great times we had together and for putting the *fun* in our everyday office life, our research, and our teaching. Also thanks to Ahmed, Dan, Habib, Hatem, Heng, Kubilay, Majid, Matthias, Peter, Piotr, Ruben, Sabine, Tsveti, and Ute for being part of and contributing to our group.

I consider myself very lucky for having worked with some very talented students and I am very grateful for their input and support, especially Paul, Michael, Jannik, and Suman. I would also like to thank Alexander Biedermann, Lars Patzina, Sven Patzina, and Tasuku Ishigooka for providing the opportunity to work on joint papers in the areas of runtime monitoring and cyberphysical systems. Also, thanks to Thomas E. Fuhrman for the many interesting discussions during our joint project.

Last but not least, I would like to thank my family and friends for their support during this exciting time.

ABSTRACT

More and more devices of our everyday life are computerized with smart embedded systems and software-intensive electronics. Whenever these pervasive embedded systems interact with the physical world and have the potential to endanger human lives or to cause significant damage, they are considered *safety-critical*. To avoid any unreasonable risk originating from the failure of such systems, stringent development processes, safety engineering practices, and safety standards are followed and applied for their development and operation. Thereby, *functional safety mechanisms* provide technical solutions to detect faults or control failures in order to achieve or maintain a safe state. In consequence, the requirements on their dependable and trustworthy operation are correspondingly high. On this background, this thesis is concerned with the *assessment* and *enhancement* of functional safety mechanisms in software-intensive safety-critical embedded systems at the example of automotive systems based on the AUTOSAR standard.

An established technique for dependability *assessments* is fault injection (FI). The effective adaptation and application of FI to modern embedded safety-critical software systems, such as AUTOSAR, is non-trivial due to their complexity and multiple levels of abstraction that are introduced by model-based development, layered architectures, and the integration of components from various suppliers, which impact the overall customizability, usability, and effectiveness of experiments. Facing these challenges, this thesis develops a complete FI process, which includes a guidance framework for the systematic and automated instrumentation with FI test code, a FI framework for the automated execution of experiments, a detailed discussion on the derivation of fault models, and the demonstration of their effective application in two case studies that uncovered an actual deficiency in a functional safety mechanism.

Due to the high cost-saving potential, functionality of varied criticality is increasingly integrated into so-called mixed-criticality systems. To provide efficient protection of critical tasks, functional safety mechanisms benefit from accounting for different criticality levels. At the example of AUTOSAR's timing protection, we illustrate the issues emerging from the lack of criticality awareness and the resulting indirect protection of critical tasks. As mitigation, we propose a novel monitoring scheme that directly protects critical tasks by providing them with execution time guarantees and implement our approach as an *enhancement* to the existing monitoring infrastructure.

KURZFASSUNG

Eingebettete Systeme durchdringen zunehmend unseren Alltag und agieren dabei oft automatisiert und autonom. Sofern sie physisch auf ihre Umwelt einwirken und potenziell Leben gefährden oder signifikanten Schaden verursachen können, werden sie als *safety-kritisch* eingestuft. Um das Restrisiko, das von einer Fehlfunktion solcher Systeme ausgehen kann, zu minimieren, werden sicherheitstechnische Anforderungen, Standards und Entwicklungsprozesse strikt befolgt. *Funktionale Sicherheitsmechanismen* bieten dabei technische Lösungen, um Fehlerzustände erkennen, und Fehlerwirkung kontrollieren, zu können. Aufgrund der enormen Bedeutung dieser Mechanismen für die Sicherheit des Gesamtsystems, werden an ihre Zuverlässigkeit entsprechend hohe Anforderungen gestellt. Die vorliegende Arbeit beschäftigt sich vor diesem Hintergrund mit der Prüfung und Verbesserung funktionaler Sicherheitsmechanismen in software-intensiven, safety-kritischen, eingebetteten Systemen, am Beispiel von auf dem AUTOSAR Standard basierten Automobilsystemen.

Fehlerinjektion (FI) ist ein bewährter Ansatz um die Zuverlässigkeit kritischer Systeme zu prüfen und zu beurteilen. Die effektive Anwendung von FI-basierten Tests auf modernen eingebetten Softwaresystemen (wie z.B. AUTOSAR) wird aufgrund ihrer Komplexität und des hohen Abstraktionsgrads behindert, welchen modellbasierte Entwicklungsansätze, die mehrschichtige Softwarearchitektur und die Integration von Komponenten verschiedener Zulieferer verursachen. Um eine effektive Anwendung zu ermöglichen, entwickelt diese Arbeit einen umfassenden FI-Prozess, bestehend aus einem Framework für die systematische Instrumentierung mit FI-Testcode, einem Framework für die automatisierte Experimentausführung, einer ausführlichen Diskussion der Herleitung von Fehlermodellen und der Demonstration ihrer effektiven Anwendung in zwei Fallstudien, die einen Defekt in einem Sicherheitsmechanismus aufgedeckt haben.

Aufgrund des hohen Einsparpotenzials werden Funktionen unterschiedlicher Kritikalität zunehmend in sogenannte Mixed-Criticality Systeme integriert. Funktionale Sicherheitsmechanismen profitieren, zum effizienten Schutz kritischer Tasks, von einer Anpassung an diese Umgebung. Am Beispiel der AUTOSAR Timing Protection veranschaulichen wir die Problematik, die sich aus einer fehlenden Anpassung ergeben kann. Als Verbesserung schlagen wir einen neuen Schutzmechanismus vor, der kritische Tasks durch garantierte Laufzeitbudgets direkt schützt, und implementieren unseren Ansatz als Erweiterung einer bestehenden Monitoring-Infrastruktur.

PUBLICATIONS

The following previously published material has been, in parts verbatim, included in this thesis.

[PWM+12] Thorsten Piper, Stefan Winter, Paul Manns, and Neeraj Suri. „Instrumenting AUTOSAR for Dependability Assessment: A Guidance Framework." In: *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1–12.

[PWS+15a] Thorsten Piper, Stefan Winter, Oliver Schwahn, Suman Bidarahalli, and Neeraj Suri. „Mitigating Timing Error Propagation in Mixed-Criticality Automotive Systems." In: *Proceedings of the 18th IEEE International Symposium On Real-time Computing (ISORC)*, 2015.

[PWS+15b] Thorsten Piper, Stefan Winter, Neeraj Suri, and Thomas E. Fuhrman. „On the Effective Use of Fault Injection for the Assessment of AUTOSAR Safety Mechanisms." In: *Proceedings of the 11th European Dependable Computing Conference (EDCC)*, 2015.

[WPS+15] Stefan Winter, Thorsten Piper, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. „GRINDER: On Reusability of Fault Injection Tools." In: *Proceedings of the IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2015.

The following publications are related to different aspects covered in this thesis, but have not been included.

[BPP+11] Alexander Biedermann, Thorsten Piper, Lars Patzina, Sven Patzina, Sorin A. Huss, Andy Schürr, and Neeraj Suri. „Enhancing FPGA Robustness via Generic Monitoring IP Cores." In: *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)*, 2011, pp. 379–386.

[ISP+14] Tasuku Ishigooka, Habib Saissi, Thorsten Piper, Stefan Winter, and Neeraj Suri. „Practical Use of Formal Verification for Safety Critical Cyber-Physical Systems: A Case Study." In: *Proceedings of the IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2014, pp. 7–12. DOI: 10.1109/CPSNA.2014.20.

[PPP+10] Lars Patzina, Sven Patzina, Thorsten Piper, and Andy Schürr. „Monitor Petri Nets for Security Monitoring.“ In: *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems (S&D4RCES)*. Vienna, Austria, 2010, 3:1–3:6. DOI: 10.1145/1868433.1868438.

[PPP+13] Lars Patzina, Sven Patzina, Thorsten Piper, and Paul Manns. „Model-Based Generation of Run-Time Monitors for AUTOSAR.“ In: *Modelling Foundations and Applications*. Ed. by Pieter Van Gorp, Tom Ritter, and Louis M. Rose. Vol. 7949. Lecture Notes in Computer Science. **Winner of the Best Paper Award**. Springer Berlin Heidelberg, 2013, pp. 70–85. ISBN: 978-3-642-39012-8. DOI: 10.1007/978-3-642-39013-5_6.

[WGG+12] Stefan Winter, Daniel Germanus, Hamza Ghani, Thorsten Piper, Abdelmajid Khelil, and Neeraj Suri. „Trustworthiness evaluation of critical information infrastructures.“ In: *Critical Infrastructure Security: Assessment, Prevention, Detection, Response*. Ed. by Francesco Flammini. WIT Press, 2012. Chap. 8, pp. 125–140.

CONTENTS

LIST OF TABLES

LIST OF LISTINGS

Part I

INTRODUCTION

1 INTRODUCTION

The ongoing computerization of devices of our everyday life is driven by the vision of comfort and simplification. By offloading tasks to *smart* computing systems that are increasingly interconnected and that integrate data from various sensors (e.g., health monitors, smart homes, and computer vision), we strive for an automation of processes and for an autonomous interaction of these systems with our environment. Manifestations of this trend are manifold. The *internet of things* connects physical things to the internet via smart, embedded sensors. *Cyber-physical systems* integrate computation and physical processes with the aim of merging the virtual and physical worlds. Advanced driver assistance features fuse sensor information to create a virtual representation of their environment to support the driver and to provide active safety. These features are key in subsequently enabling *autonomous driving* of highly interconnected vehicles.

As these examples show, computers are already pervasively embedded in our everyday lives and this trend is going to continue. Due to the complex tasks they implement, computers are increasingly software-intensive and software is extensively used to control many aspects of our everyday life. In consequence, the complexity of these computer systems and their software is growing as their adaptation and pervasiveness progresses.

While these systems ease our lives in various ways, we are also increasingly reliant on their *dependable* operation, as individuals and as society. Especially when they directly interact with the physical world, the failure of such pervasive, embedded systems could endanger human lives or has the potential for significant damages to property or the environment. In those cases we consider these systems to be *safety-critical*. In order to guarantee the safe operation of these systems, their design and verification follows stringent processes to prevent and remove faults during development with the aim to minimize the number of residual faults. To detect and handle erroneous behavior during runtime and to control hazardous situations, *safety mechanisms* are commonly used. For systems that integrate functions of different criticality levels (so-called *mixed-criticality systems*), safety mechanisms also serve the purpose of providing *freedom from interference*, i.e., ensuring that errors in less critical functions cannot disturb more critical functions through error propagation in space and time. A thorough assessment of these safety mechanisms and their correct and effective application is key to establish verifiable trust in the safe and dependable operation of these systems.

An established technique for the dependability and robustness assessment of safety-critical systems and their components is fault injection (FI) [HTI97]. In FI experiments, faults are deliberately introduced into a system with the intent to analyze the resulting system behavior and to establish proof that the system is able to handle certain fault types. Originally, FI experiments were directly conducted on hardware, e.g., by exposing it to heavy-ion radiation and electromagnetic interferences [GKT89], or by performing direct modifications on the pin-level [AAA+90]. To increase the reproducibility of the experiments and to allow for higher experiment throughput, the *effects* of these hardware injections were subsequently modeled in software (e.g., by bit flips), with the aim to emulate hardware faults without conducting time-intensive and complex physical experiments [SVS+88; KKA95; SBK10]. These experiments are commonly known as software-implemented fault injections (SWIFI). With the growing complexity of software, FI developed further into so-called software fault injections (SFIs). The focus of SFI is now to inject faults that represent defects that are directly caused by software issues [DM06; CLN+12], such as programmer mistakes, software ageing, and concurrency and timing issues.

As the mechanisms and techniques of FI develop and progress, so do the areas where it is applied. For example, the automotive industry has lately started to adapt FI as a technique for the assessment of the correctness and effectiveness of their safety mechanisms and critical components. This adaptation is strongly driven by the functional safety standard for road vehicles ISO 26262 [Int11] that recommends FI as assessment technique. The effective adaptation and application of FI for such complex, software-intensive, and safety-critical systems is non-trivial due to several factors.

1. FI experiments necessitate the instrumentation of the target system with test code on the implementation level. Model-based development, layered architectures, and the integration of intellectual property (IP) from various suppliers into so-called mixed-IP systems introduce several levels of abstraction that impact the customizability, usability, and effectiveness of instrumentation and experiments.

2. The choice of representative fault models, i.e., faults that can and do occur during operation, has major impact on the effectiveness of FI experiments. As the specification of fault models relies to a high degree on expert domain knowledge, choosing the right fault models is challenging.

3. To minimize the influence of human error on FI experiments and to automate the experiment process for efficiency, FI frameworks are generally used to run large numbers of consecutive experiments. The framework architecture and design should

factor the various abstraction levels of the system and be able to inject faults and monitor system behavior on the complete software stack. At the same time, time-consuming process-steps, such as recompilation and reflashing, should be avoided or minimized to achieve a higher efficiency.

On this general background, this thesis

1. develops and presents a guidance framework for the instrumentation of software-intensive and safety-critical systems,
2. develops and adapts a FI framework and provides a detailed discussion on the derivation of fault models, whose effective application is demonstrated in two case studies for the assessment of timing protection safety mechanisms, and
3. develops a novel timing protection safety mechanism for mixed-criticality systems as an enhancement to the existing monitor infrastructure.

We apply and evaluate the developed techniques on a modern automotive embedded system, based on the AUTOSAR standard [AUT14a]. AUTOSAR implements a highly standardized development process and layered software architecture, for which systems are designed as abstract models that consist of *software components (SW-Cs)*, and compositions thereof, as the core building blocks.

As automotive systems are highly complex and software-intensive (up to 100 million lines of code distributed over more than 70 electronic control units (ECUs) [Cha09]), and contain many functions that are safety-critical, AUTOSAR-based systems are an appealing evaluation target and representative for highly standardized, software-intensive, component-based, safety-critical embedded systems.

This chapter is structured as follows. In Section 1.1 we discuss the role of functional safety mechanisms in the context of the overall safety concept, and continue their discussion in the context of dependability in Section 1.2. In Section 1.3 we describe how functional safety mechanisms contribute to system safety during operation and present our research questions and contributions in the context of the assessment and enhancement of functional safety mechanisms in Section 1.4.

1.1 THE ROLE OF FUNCTIONAL SAFETY MECHANISMS

This thesis is concerned with the safe operation of those systems, whose failure could result in loss of life, significant property damage, or damage to the environment, i.e., systems that are *safety-critical*, and which implement large parts of their safety concept in software. Safety, just as security, is a system property and not an isolated

property. So, whether a system is in a safe or unsafe state also depends on its operational context. This also means that safety cannot simply be added at the end of a development process (just like security), if the system has not been designed with its respective safety requirements and their implications on the system design right from the start. Therefore, a strong emphasis is usually put on the system development process and system safety engineering.

This section details the safety engineering process and introduces the terminology, which we use throughout this thesis, according to the safety standard ISO 26262 with the intent to illustrate the role of *functional safety mechanisms* in a system- and process-wide perspective. ISO 26262 is the adaptation of IEC 61508 [Int10] to comply with needs specific to the application sector of electrical and/or electronic (E/E) systems within road vehicles.

The first step in the development of a safety concept is *hazard analysis and risk assessment*. Its objective is to identify and to categorize the *hazards*[1] that malfunctions in the system can trigger and to formulate the *safety goals* related to the prevention or mitigation of the hazardous events, in order to avoid *unreasonable risk*. Techniques such as brainstorming, checklists, quality history, failure mode and effects analysis (FMEA), and field studies can be used for the extraction of hazards at the system level.

As next step, hazardous events, i.e., the combination of a hazard and a scenario that can occur during a vehicle's life (e.g., parking, maintenance), are classified according to their *severity*, *probability of exposure*, and *controllability*. Based on this classification, an automotive safety integrity level (ASIL) and a *safety goal*, i.e., a top-level safety requirement, is assigned to the hazardous event.

For the *functional safety concept*, the *functional safety requirements* are derived from the safety goals, and allocated to preliminary architectural elements of the system or to external measures. The functional safety concept addresses [Int11]:

- fault detection and failure mitigation,
- transitioning to a safe state,
- fault tolerance mechanisms, where a fault does not lead directly to the violation of the safety goal(s) and which maintains the system in a safe state (with or without degradation),
- fault detection and driver warning in order to reduce the risk exposure time to an acceptable interval (e.g., engine malfunction indicator lamp, ABS fault warning lamp), and
- arbitration logic to select the most appropriate control request from multiple requests generated simultaneously by different functions.

[1] A hazard is a potential source of harm caused by malfunctioning behavior.

To comply with the safety goals, the functional safety concept contains *safety measures*[2], either in the form of *safety mechanisms* or implemented in the system's architectural elements. Thereby, a *safety mechanism* is a technical solution implemented by E/E functions or elements, or by other technologies, to detect faults or control failures in order to achieve or maintain a safe state.

1.2 SAFETY RELIES ON DEPENDABILITY – AND VICE VERSA

In order to provide safety, a system relies on the *dependability* of the safety mechanisms, i.e., they should be free from defects and be able to tolerate faults during operation. During development, the following means are taken to attain dependability and minimize the number of residual defects [ALR+04]:

- *Fault prevention* avoids faults from being introduced into a system by using better development methodologies (e.g., defensive programming, usage of formal methods to express a specification), better programming languages (e.g., type-safe languages), and improving the qualification of developers.
- *Fault removal* techniques (e.g., static and dynamic testing methods) aim at detecting faults during development and removing them before a system is deployed.
- *Fault forecasting* estimates the present number, the future incidence, and the likely consequences of faults (e.g., based on statistical data, hot-spot analysis) to indicate when an acceptable level has been attained.

Despite these means, usually not *all* faults can be removed at design time, mostly due to the associated costs, limited development and testing time, and the lack of techniques for the complete verification of complex (software) systems. Often the trigger conditions are simply too complex (so-called Mandelbugs) and faults only manifest as the system is exposed to unforseen and untested operational conditions (e.g., system load and usage, resource depletion due to ageing). The problem of developing fault-free software systems is further hardened in the presence of real-time constraints, concurrent operations, or distributed environments.

As the avoidance, detection, and removal of all faults during design time is apparently impossible, *fault tolerance* mechanisms [ALR+04] are usually employed to tolerate the effects of residual design faults and operational faults during runtime. Their goal is to avoid and

[2] A safety measure is an activity or technical solution to avoid or control systematic failures and to detect random hardware failures or control random hardware failures, or mitigate their harmful effects.

Figure 1.1: The fundamental chain of dependability and security threats (from [ALR+04]).

mitigate failures in the presence of faults and to provide sustained correct service, although that service may be at a degraded level.

Figure 1.1 depicts the relation between *fault*, *error*, and *failure*. A fault is a defect or flaw (e.g., in the programming or design) that, when *activated*, produces an error (i.e., a fault is the cause of an error). Before a fault is activated (e.g., by a certain input) it is dormant. An error is the part of a system's total state that may lead to a failure. If the error *propagates* to the boundary of a system component, a failure occurs when the error causes the delivered service of the component to deviate from correct service.

1.2.1 *Error Propagation: Implications and Solutions*

Error propagation between different components of a system may occur, when an error reaches the service interface of the faulty component without being detected by a fault tolerance mechanism. In that case, any component using the service interface of the faulty component will receive incorrect service and the service failure appears as an external fault to that component (causation of external fault in Figure 1.1). The effect of error propagation is highly undesirable, especially in safety-critical systems, as errors in uncritical components may negatively affect the operation of critical components.

In general, two approaches are taken to mitigate the effect of error propagation and to prevent propagating errors from affecting critical components: robustness hardening and isolation.

1. *Robustness* is the ability of a component or system to handle external faults during execution. In order to make a component robust, i.e., to harden it, sanity or range checks are performed on the inputs of its interfaces and error detection and handling mechanisms are employed.

2. *Isolation* eliminates error propagation paths by separating components as much as possible, e.g., by a segregated architecture, also with the aim of eliminating common cause failures.

While the traditional approach of *physical* isolation used to work very well, it is often not contemperary anymore. For once, it scales badly and it is hard to realize in many modern systems that require a tight overall integration, such as sensor fusion scenarios in which various control processes simultaneously access signals of multiple sensors and perform computations on them. Further, physical iso-

lation often is a cost issues (e.g., in the automotive domain) as it necessitates multiple controller boards, periphery, and wiring.

John Knight [Kni02] already stated in 2002 that „the number of interacting safety-critical systems present in a single application will force the sharing of resources between systems. This will eliminate a major architectural element that gives confidence in correct operation – physical separation. Knowing that the failure of one system cannot affect another greatly facilitates current analysis techniques. This will be lost as multiple functions are hosted on a single platform to simplify construction and to reduce power and weight requirements. Techniques that provide high levels of assurance of non-interference will be required."

As an example, the integration of the historically segregated automotive systems, which have been conservatively designed following a *one function per electronic control unit (ECU)* approach, offers cost saving potential for hardware and wiring, as it entailed up to 100 federated ECUs distributed in modern luxury cars [Cha09]. In consequence, modern embedded controllers require the integration of highly complex and modular software, which can consist of both, uncritical and safety-critical software components, into so-called mixed-criticality systems.

1.3 FUNCTIONAL SAFETY MECHANISMS FOR FREEDOM FROM INTERFERENCE

The automotive industry serves as an illustrative example how highly standardized development and safety engineering processes are used to tackle (1) the challenge of software complexity and (2) the challenge of designing safe systems in a systematic way.

To standardize the development process and software architecture of automotive systems, the AUTOSAR (AUTomotive Open System ARchitecture) consortium [AUT14a; HSF+04] specifies an open industry standard for automotive software systems. The standardization is driven by the need to address the growing complexity of modern vehicular systems and to reduce development costs when introducing new software-based features. AUTOSAR is organized as a layered, modular architecture, and is based on a component- and composition-centric development process that standardizes the modeling and naming schemes within the system, including components, interfaces, data types and runnables. The standard promotes the integration of white-box and black-box components into a grey-box system, allowing for the integration and reuse of intellectual property of different suppliers (so-called mixed-IP systems).

At the same time, manufacturers comply to industry-wide safety standards and functional safety specifications, such as IEC 61508 [Int10] and in particular the functional safety standard for road vehi-

cles ISO 26262 [Int11; BFK10], in their design, development and production processes for the underlying software. This covers rigorous software design processes along with analytical and test techniques at the static software levels. Similar to the „gold standard for partitioning“ in integrated modular avionics [Rus99], ISO 26262 permits the integration of elements with differing criticality, as long as partitioning mechanisms can verifiably provide *freedom from interference* in both the spatial and temporal domains, i.e., regarding memory accesses and timing behavior. In automotive systems, partitioning is usually supported by hardware [WEK10; ZBS+14], the operating system (OS) [AUT14i; BFT09], or a combination of both (e.g. virtualization) [RM14].

AUTOSAR addresses freedom from interference specifically through a set of functional safety mechanisms [AUT14e; AUT14i] that are provided as services by the AUTOSAR OS, the Watchdog Manager, or external libraries. These mechanisms include memory partitioning, timing monitoring, logical supervision, and end-2-end protection. To support partitioning in the temporal domain, the OS provides monitoring of task execution time budgets, activation frequencies, and resource lock times.

As functional safety mechanisms assume a key role in the safety architecture, their dependability requirements are correspondingly high. They serve the purpose of (1) error detection to realize fault tolerance, but also (2) isolation by error confinement to realize freedom from interference. On this background, this thesis explores two directions in the context of functional safety mechanisms:

1. The *assessment* of the correctness and robustness of these mechanisms, i.e., how they perform in the presence of specified (targeting correctness) and unspecified (targeting robustness) faults.

2. The *enhancement* of these mechanisms to mixed-criticality systems, i.e., how can these mechanisms be effectively and efficiently applied for mixed-criticality applications.

1.4 RESEARCH QUESTIONS AND CONTRIBUTIONS

Research Question (RQ1): How do we drive the instrumentation of complex, software-intensive mixed-IP systems for dependability assessment?

Dynamic testing techniques for dependability assessment, e.g., fault injections, necessitate the instrumentation of the system under test with code to conduct the actual injections at various injection locations and to monitor the actual system response to the injected faults, e.g., to assess error detection and fault tolerance capabilities. Contemporary model-based development processes provide high usability, but at the same time introduce a high degree of abstraction from

the implementation layer, on which the actual instrumentation is conducted, with the consequence of ambiguous mappings between the model and implementation view. Layered architectures and the integration of intellectual property from various suppliers into mixed-IP systems introduce further levels of abstraction that impact the customizability, usability, and effectiveness of instrumentation and experiments.

Contribution (C1): A guidance framework for the dependability assessment of complex, software-intensive mixed-IP software systems.

To enable a systematic process for the instrumentation of complex, software-intensive mixed-IP systems, we propose to combine information from the system model and the implementation to create a hybrid view of the system, effectively establishing a relation between the model and the implementation. At the example of AUTOSAR, we demonstrate how its high degree of standardization enables the development of independent testing procedures that transcend manufacturer-specific solutions, and thereby supports a flexible, systematic, and automated approach. By operating mainly on the standardized model specification, a transition from a monolithic testing approach to a component based one, with emphasis on component isolation and communication via standardized interfaces, can be made. In Chapter 2 we develop a dependability assessment guidance framework tailored towards AUTOSAR that helps to identify the applicability and effectiveness of instrumentation techniques at (1) varied levels of software abstraction and granularity, (2) at varied software access levels - black-box, grey-box, white-box, and (3) the application of interface wrappers for conducting FI. The results of this work were presented at DSN 2012 [PWM+12].

Although our focus in this thesis is on the assessment and enhancement of functional safety mechanisms, the overall instrumentation process is generally applicable and not limited to these application scenarios. We have also applied the instrumentation approach in the context of complex runtime monitors for protocol supervision and presented the results of this joint work at ECMFA 2013 [PPP+13].

Research Question (RQ2): How can FI be effectively and efficiently applied for the assessment of functional safety mechanisms?

The automotive safety standard ISO 26262 strongly recommends the use of FI for the assessment of safety mechanisms that typically span composite dependability and real-time operations. However, with the standard providing very limited guidance on the actual design, implementation, and execution of FI experiments, most AUTOSAR FI approaches use standard fault models (e.g., bit flips and data type

based corruptions), and focus on using simulation environments. Unfortunately, the representation of timing faults using standard fault models, and the representation of real-time properties in simulation environments are hard, rendering both inadequate for the comprehensive assessment of AUTOSAR's safety mechanisms. The actual development of ISO 26262 advocated FI is further hampered by the lack of representative software fault models and the lack of an openly accessible AUTOSAR FI framework.

Contribution (C2): An open source FI framework for AUTOSAR, including fault model guidelines and the assessment of functional safety mechanisms.

Facing these challenges, we provide an open source, ready to use AUTOSAR FI tool [PMT+15] in Chapter 3 that is capable to conduct FI experiments at all layers of AUTOSAR's software architecture, i.e., application, runtime environment, and basic software. Injections in source code and binary object files (i.e., white-box and black-box) are both supported. Further, we provide guidelines for the derivation of specific fault models, injection locations and mechanisms from the abstract AUTOSAR and ISO 26262 fault models. We report our experiences in conducting a dependability assessment of a commercial implementation of AUTOSAR's timing monitoring safety mechanisms. The assessment uncovered a real deficiency in the implementation that was subsequently acknowledged and fixed by the supplier of the safety mechanisms' implementation. The results of this work are going to be presented at EDCC 2015 [PWS+15b].

Research Question (RQ3): How can the existing infrastructure of functional safety mechanisms be enhanced for mixed-criticality scenarios?

For mixed-criticality automotive systems, the functional safety standard ISO 26262 stipulates *freedom from interference*, i.e., errors should not propagate from low to high criticality tasks. To prevent the propagation of timing errors, the automotive software standard AUTOSAR provides monitor-based timing protection, which detects and confines task timing errors. As current monitors are unaware of a criticality concept, the effective protection of a critical task requires to monitor *all* tasks that constitute a potential source of propagating errors, thereby causing overhead for worst-case execution time analysis, configuration and monitoring.

Contribution (C3): A novel, criticality-aware timing protection mechanism.

Differing from the *indirect* protection of critical tasks facilitated by existing mechanisms, we propose a novel monitoring scheme in Chapter 4 that *directly* protects critical tasks from interference, by providing

them with execution time guarantees. The monitor is implemented as an enhancement to the existing monitoring infrastructure of a widely used commerical AUTOSAR OS and meant to augment existing mechanisms. Overall, our approach provides efficient low-overhead interference protection, while also adding transient timing error ride-through capabilities. The results of this work were presented at ISORC 2015 [PWS+15a].

Part II

INSTRUMENTATION

2

INSTRUMENTING AUTOSAR SYSTEMS FOR DEPENDABILITY ASSESSMENT

The AUTOSAR standard guides the development of component-based automotive software. As automotive software typically implements safety-critical functions, it needs to fulfill high dependability requirements, and the effort put into the quality assurance of these systems is correspondingly high. Testing, fault injection (FI), and other techniques are employed for the experimental dependability assessment of these increasingly software-intensive systems. Having flexible and automated support for *instrumentation* is key in making these assessment techniques efficient. However, providing a usable, customizable and performant instrumentation for AUTOSAR is non-trivial due to the varied abstractions and high complexity of these systems.

This chapter develops a dependability assessment guidance framework tailored towards AUTOSAR that helps identify the applicability and effectiveness of instrumentation techniques at (a) varied levels of software abstraction and granularity, (b) at varied software access levels - black-box, grey-box, white-box, and (c) the application of interface wrappers for conducting FI. The framework presented in this chapter forms the basis of the automated fault injection process for the assessment of the functional safety mechanisms of AUTOSAR, which is developed in Chapter 3. The content of this chapter is based on a conference paper presented at DSN 2012 [PWM+12].

2.1 AUTOMATED INSTRUMENTATION: CHALLENGES & BENEFITS

AUTOSAR (AUTomotive Open System ARchitecture) [HSF+04] is an emerging open industry standard for automotive software systems. Its development is driven by the need to address the growing complexity of modern vehicular systems and to reduce development costs when introducing new software-based features. AUTOSAR is organized as a layered, modular architecture, and is based on a component/composition-centric development process that standardizes the modeling and naming schemes within the system, including components, interfaces, data types and runnables. The standard promotes the integration of white-box and black-box components into a grey-box system, allowing for the integration and reuse of intellectual property (IP) of different suppliers (so-called mixed-IP systems).

Automobiles are safety-critical systems, i.e., systems whose failure could potentially endanger human life, property, and environment, with increasing software based functionality. In order to maintain

safety, often defined as the „absence of catastrophic consequences on the user(s) and the environment“ [ALR+04], manufacturers comply to industry-wide safety standards and functional safety specifications, such as IEC 61508 [Int10] and in particular the functional safety standard for road vehicles ISO 26262 ([Int11], [BFK10]), in their design, development and production processes for the underlying software. This covers rigorous software design processes along with analytical and test techniques at the static software levels. Moreover, experimental methods for dependability assessment (e.g., testing, fault injection, and error propagation analysis) are employed during development to analyze the system's behavior before its deployment and to ensure the fault tolerance of critical components [SK08]. Similarly, an equally widespread adoption of experimental security analyses is advisable. It has repeatedly been shown that current implementations of automotive software have severe security issues ([KCR+10], [RMM+10], [HKD11]), which can also be attributed to insufficient testing.

In practice, *instrumentation*[1] is one way to enable dependability assessment techniques within a system. To make the assessment efficient, a *flexible* and *automatic* instrumentation of the test system at different locations and varied levels of granularity is highly desirable. To systematically address the aforementioned requirements, we propose an automated process for the instrumentation of AUTOSAR systems by a framework, which provides the key features *usability*, *customizability* and *efficiency*. The implementation of such a framework for AUTOSAR is hard, mainly due to these factors:

F1) AUTOSAR systems are developed in a model based process that introduces a high degree of abstraction between the model and the actual implementation. As consequence, instrumentation at the model level has no access to implementation details (limiting customizability), while instrumenting the implementation, i.e., machine generated code, is a tedious process (limiting usability). Also, due to the degree of abstraction, elements of the model often have no singular representation in the implementation.

F2) AUTOSAR systems are composed of white-box and black-box software components as provided by various suppliers. A customizable and usable instrumentation should also be applicable to these systems, to not impact the overall efficiency of instrumentation, for instance, if an approach requires the re-compilation of the entire system. To keep the effectiveness and implications on the overall system composition and performance in mind is key.

[1] Throughout this work, we use the word instrumentation to express a modification of a program with the intent to enable an interception of data and control flow for analysis or alteration, aiming to implement the major dependability-related applications fault injection and monitoring.

2.1.1 *Contributions*

Facing these challenges, we develop an instrumentation framework for AUTOSAR systems that is usable, customizable and efficient. At the same time, we aim to establish a guidance framework on how to develop and implement a systematic instrumentation schema within the AUTOSAR environment. The key idea to address factor (F1) is to leverage collective information from the system model, provided during the development process in standardized AUTOSAR XML (ARXML) format, and the system implementation, to drive the configuration and instrumentation process.

Addressing factor (F2), we develop and advocate an interface wrapper based approach for the instrumentation realization. Wrappers are a well established concept [Voa98], that can be used to intercept inter-component communication. They are applicable to white-box, grey-box, and black-box components, *all* of which can be present simultaneously within an AUTOSAR system and are, as such, also explicitly promoted by the standard. Moreover, wrappers can implement add-on functionality and thus enable a variety of run-time testing and analysis methods, such as fault injection (FI), failure propagation analysis, and control-/ data-flow monitoring.

Having said that, our approach is the first to investigate how to systematically and automatically instrument a given system. Our contribution is to provide a guidance framework for the systematic and automated instrumentation of AUTOSAR systems that enables:

- **Usability:** Instead of requiring the user to instrument code at a low level, e.g., at the output of code generators, we enable instrumentation via high-level models. Building models is an essential abstraction step in the AUTOSAR development process to specify modular and interconnected systems. Such models are widely-used and supported by AUTOSAR design tools.

- **Customizability:** In addition, expert users of the proposed instrumentation framework are given a highly customizable interface to specify instrumentation locations that are not part of the model abstraction. We achieve this by exploiting semantic information of high-level models, e.g., the logic of generating source code from models. Furthermore, our customizable framework allows instrumentation at different software access levels, e.g., binaries (black-box) or C code (white-box).

- **Efficiency:** The proposed framework is also efficient in terms of user effort, compilation resources, execution time, and memory consumption. We implement efficiency through adaption of established SW engineering techniques such as wrappers and XML meta data. We evaluate this efficiency by conducting

fault-injection of an anti-lock braking system, which we implemented with two different AUTOSAR design tools to demonstrate efficiency and applicability across multiple vendors.

We structure this chapter as follows. We introduce the system model in Section 2.2 and review related work on AUTOSAR instrumentation in Section 2.3. In Section 2.4 we detail the development of a systematic, automated process for instrumentation. We evaluate our approach in Section 2.5 and discuss its characteristics and limitations in Section 2.6.

2.2 AUTOSAR DEVELOPMENT PROCESS AND SYSTEM MODEL

Many techniques for component level dependability assessment (e.g., fault injection) or reliability enhancement (e.g., run-time monitors), rely on accessing or modifying the actual data flow between the components [CC96; MCV00]. This section provides the foundations to understand key aspects of the AUTOSAR development process and architecture, which is essential to appreciating the difficult challenge of instrumentation and its effective usage.

AUTOSAR's focus is to provide the software architecture for distributed automotive systems. In these systems, electronic control units (ECUs) constitute nodes that implement dedicated functionality and that communicate via bus systems such as CAN, LIN and FlexRay. AUTOSAR systems are created in a model driven development process, in which the developer composes the system model typically via a graphical user interface. This model provides an abstract view on the system, making no assumptions on the distribution or mapping of resources in a later development stage. Large parts of the overall software code base are generated from this model and only a fraction of the system development is done on the actual implementation level. This approach provides the developer great usability and flexibility in terms of evolving system configuration, as the assignment of resources (i.e., mapping components to ECUs) and low level implementation become decoupled processes. On the other hand, this approach also entails a high degree of abstraction between the model and the actual implementation.

We use the example model in Figure 2.1 to introduce some of the key concepts of AUTOSAR, and to show the ambiguity of the model concepts in the implementation domain later on. The top-level element in any AUTOSAR system is a composition (depicted by the dashed box), which can be thought of as a container that contains other compositions or software components (SW-Cs). The standard defines several types of SW-Cs, such as application SW-C or sensor-actuator SW-C, and SW-Cs provide functionality to the system through *runnables*, which are timer or event (e.g., message arrival) triggered functions that are implemented in C or C++. SW-Cs com-

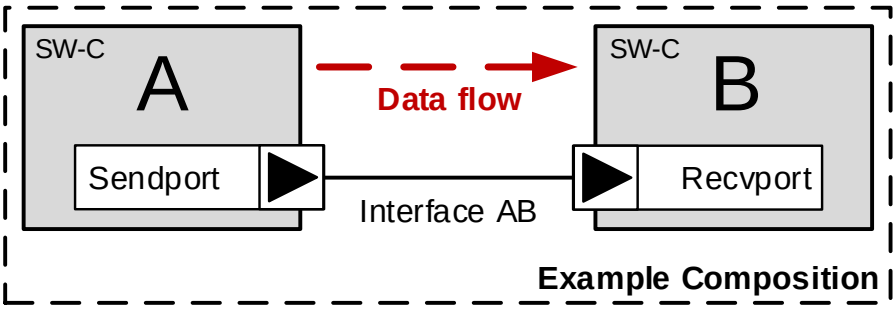


Figure 2.1: Model of two software components (SW-Cs) communicating via a sender-receiver interface.

municate with each other via standardized *port interfaces* (or simply *interfaces*, in our example *Interface AB*), which specify the communication method and provide a link between components. Interfaces are accessed through *ports*, serving as communication endpoints. As such, ports provide access to a named point-to-point connection between components that uses standard communication patterns, such as client-server or sender-receiver, to exchange data or invoke server operations.

Spotting and identifying points for communication interception seems trivial in the model, but becomes non-intuitive in the implementation mostly due to the high abstraction degree of the model. In fact, after reviewing the AUTOSAR architecture and the data flow between components within, we show that, for example, the modeling concept *port* has no singular representation within the implementation. It is important to understand that the view on the system that the model provides is inherently different from the implementation's view of the system. In the model, SW-Cs are directly connected to each other via their respective ports and port interface. But, unlike the model suggests, there is no direct communication between SW-Cs in the implementation. Instead, each SW-C invokes the API of a *runtime environment (RTE)*, which abstracts the services/primitives for inter-component communication. The RTE is a layer of the AUTOSAR software architecture, as shown in Figure 2.2. To understand how connections (and eventually communication) between components in the model manifest in the implementation, we have to understand further details of the architecture.

At the *implementation* level, AUTOSAR is organized as a layered, modular software architecture in which each layer provides an abstraction of the underlying layer and a set of services to the overlying layer. As mentioned, the RTE implements communication services for the SW-Cs and transparently abstracts from the actual communication medium or channel. In order to dispatch communication and route messages, the RTE uses the services provided by the basic software (BSW) layer, which itself is composed of several sub-layers and modules, and provides the hardware abstraction.

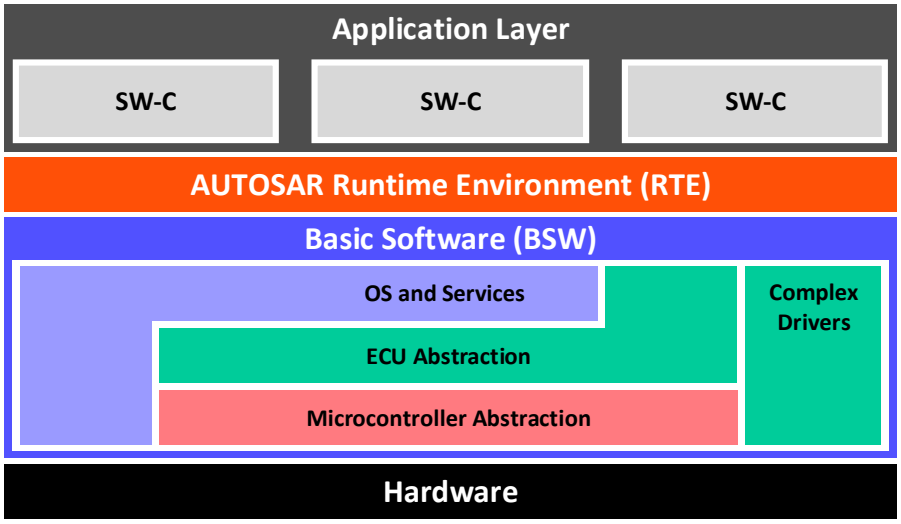


Figure 2.2: The AUTOSAR layered software architecture (adapted from [AUT11]).

Recalling the example model of Figure 2.1, the communication between components A and B can result in three distinct communication paths in the implementation, as shown in Figure 2.3. In the case where component A and B reside on the same ECU (as in the left part of the picture) the communication can either involve only the RTE or the RTE and the BSW. In the distributed case, where component A and B reside on different ECUs, the communication also involves a network, such as CAN. The direct communication between components that the model view suggests, obviously gets split into *several phases* within the implementation. To give an example, the communication process and data flow for the most simple case of communication is as follows. In phase one, SW-C A invokes an API call of the RTE to send a message to SW-C B. The interface handler of the RTE processes the call and stores the message until delivery. Phase two starts when SW-C B invokes an API call of the RTE to read the message. The interface handler of the RTE loads the stored message and delivers it to SW-C B. So, the message first passes the interface between SW-C A and the RTE, and then the interface between the RTE and SW-C B. As each of these interfaces has two communication endpoints, one within the SW-C and one within the RTE, we have the choice of four distinct locations to intercept the dataflow between component A and B – for the simplest case.

Another factor that adds to the complexity of the scenario, is the distributed development of AUTOSAR systems. AUTOSAR advocates a component-based design with standardized interfaces to support the integration of application components that are supplied by third party manufacturers, into the overall system. Third party suppliers receive, alongside with the SW-C's functional requirement specification, an interface specification that results from the code gen-

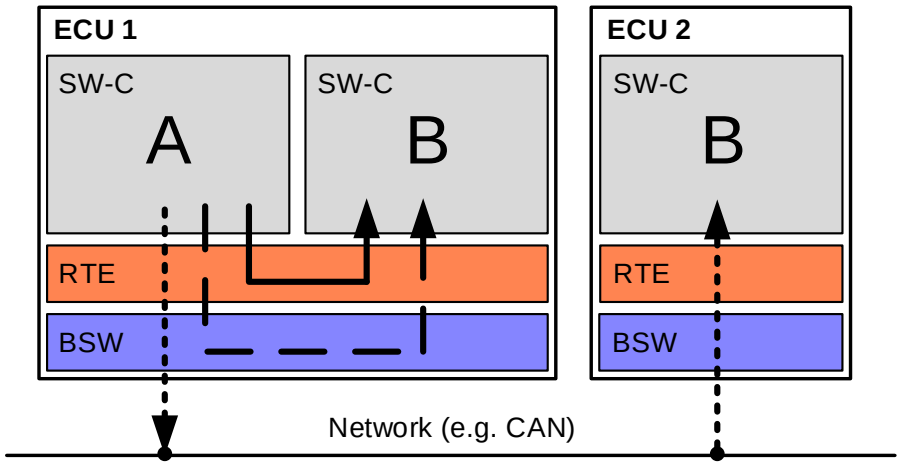


Figure 2.3: Possible data flow paths of two communicating SW-Cs at the implementation level.

eration process. They have the option to deliver the implemented functionality either as source-code (white-box) or binary object-code (black-box). Both options are explicitly supported by the AUTOSAR standard, whereas delivering the implementation in binary form aids in protecting the intellectual property of the external supplier. In addition to the different instrumentation locations (SW-C and RTE), an instrumentation approach therefore has to factor the different code access levels that might be present in the system.

We address this scenario as follows. To bridge the gap between the model and the implementation, we propose to leverage information from the model and the implementation to create a collective view of the system. After a review of related work in the following section, we explain the technical details of extracting the necessary information from the model and the implementation in Section 2.4. Furthermore, we show how to technically drive wrapper-based interface instrumentation on the access levels of source code and binary object, and provide suggestions on how to use this information to develop an instrumentation framework.

2.3 RELATED WORK ON AUTOSAR INSTRUMENTATION

In order to implement fault-tolerance extensions or to conduct fault injection experiments, several publications have dealt with the instrumentation of AUTOSAR software systems.

In [LFK09a], Lu et al. propose a fault-tolerance extension for automotive modular embedded software, which is implemented as an error monitor in an external customizable component. The external monitor instruments and interfaces the monitored system via *software hooks* provided by the AUTOSAR OS on certain events (e.g., task start,

task stop, and OS errors), based on the user's OS configuration. The approach is capable of monitoring the control- and data-flow at the OS level, with granularity restricted to task invocations. Apart from the low granularity, the approach is limited by the instrumentation at the OS level (therefore requiring OS access) and the use of software hooks, which necessitates white-box access to those parts of the OS that implement the hooks.

In [LFK09b], the authors suggest a wrapping-based approach that partly addresses above limitations. The approach is based on the same architecture, but targets the RTE as the instrumentation location, leveraging software hooks provided by the RTE. The granularity of monitoring is substantially improved to tracking interactions between SW-Cs and RTE at the interface level, and is comparable to our approach. Furthermore, the approach only requires RTE access and no longer OS access. On the other hand, white-box restrictions still apply, while implicitly necessitating the time-consuming recompilation of code, when the configuration of instrumentation changes.

Lanigan et al. [LNF10] published a feasibility study of fault injection in AUTOSAR using CANoe, a commercial tool that provides a simulation and evaluation environment for automotive applications. As with the previous approaches, the instrumentation method of choice is software hooks. The authors restrict themselves to the basic software (BSW) layer and do not instrument the RTE, as „it is mostly auto-generated by the AUTOSAR configuration tools". While the instrumentation at BSW service level provides a better granularity than at the OS level, the same access and white-box limitations as for [LFK09a] apply, due to the similarities in instrumentation method and location. As the approach targets a specific tool, the generic applicability is restricted.

In summary, the review of related work shows that all current approaches rely on hooks provided by either the BSW, the OS or the RTE, requiring at least partial source code access, and in turn, the recompilation of parts of the system for different configurations. Currently, none of the existing approaches addresses the different access levels of white-box, grey-box or black-box, which are explicitly promoted by AUTOSAR. Furthermore, the question of how to systematically and automatically instrument a given system is not investigated. Also, none of the publications provides an (experimental) evaluation of the overhead incurred by the instrumentation. We are the first to address these open issues.

2.4 INSTRUMENTING AUTOSAR SOFTWARE COMPONENTS

AUTOSAR's high-level view of inter-component communication facilitates the identification of candidate locations for instrumentation.

In order to analyze or intercept the communication on the component level, i.e., among the core building blocks of AUTOSAR systems, communication end-points of interest can be chosen from the set of ports that are used for component interconnection. Unfortunately, this simplicity of the communication model is not reflected in the *tool-generated* source code structure, for which the actual instrumentation has to be implemented. In the following we discuss how AUTOSAR's high-level communication model translates to source code constructs, along with the resulting opportunities for instrumentation.

2.4.1 *Inter-Component Communication: Model vs Code*

AUTOSAR models are stored as machine-readable specification in a XML-based data format called ARXML. A code generator translates these models into an implementation code skeleton. To illustrate the code generation process, we have modeled the system presented in Figure 2.1, which, despite its simplicity, resulted in almost 140 lines of ARXML code. From this code, an extract of the component specification of SW-C A is shown in Listing 2.1, with the intent to provide an illustrative example and give the reader a glimpse at the overall process.

```
1  <APPLICATION-SOFTWARE-COMPONENT-TYPE>
2    <SHORT-NAME>A</SHORT-NAME>
3    <PORTS><P-PORT-PROTOTYPE>
4      <SHORT-NAME>SendPort</SHORT-NAME>
5      <PROVIDED-COM-SPECS><UNQUEUED-SENDER-COM-SPEC>
6      <DATA-ELEMENT-REF DEST="DATA-ELEMENT-PROTOTYPE">
7        /rootPackage/SendPort/DataPrototype</DATA-ELEMENT-REF>
8      </UNQUEUED-SENDER-COM-SPEC></PROVIDED-COM-SPECS>
9      <PROVIDED-INTERFACE-TREF DEST="SENDER-RECEIVER-INTERFACE">
10       /rootPackage/SendPort</PROVIDED-INTERFACE-TREF>
11   </P-PORT-PROTOTYPE></PORTS>
12 </APPLICATION-SOFTWARE-COMPONENT-TYPE>
```

Listing 2.1: Component prototype specification (extract from the ARXML of the model in Figure 2.1).

The specification's key elements are the component name (line 2) and the *provide port* definition (lines 3-11), which contains references to the *data prototype* (lines 6-7) and the *interface* (lines 9-10). In [AUT14h], the AUTOSAR standard defines various interface types, which are translated to more than 20 API types during the code generation, each of which is generated with a strict naming scheme to ensure interoperability. In our example, the interface is of type *SENDER-RECEIVER-INTERFACE* (lines 9-10), and the specification results in the generation of an *Rte_Write* API. For this API, the naming scheme is defined as `Rte_Write_<p>_<o>`, where `<p>` denotes the

port name and <o> the *DataElementPrototype*. Line 4 of Listing 2.1 specifies the port name (*SendPort*) while the DataElementPrototype can be obtained by de-referencing the interface reference in lines 9-10.

```
1 <SENDER-RECEIVER-INTERFACE>
2   <SHORT-NAME>SendPort</SHORT-NAME>
3   <DATA-ELEMENTS><DATA-ELEMENT-PROTOTYPE>
4     <SHORT-NAME>DataPrototype</SHORT-NAME>
5     <TYPE-TREF DEST="INTEGER-TYPE">
6       /rootPackage/DataType</TYPE-TREF>
7   </DATA-ELEMENT-PROTOTYPE></DATA-ELEMENTS>
8 </SENDER-RECEIVER-INTERFACE>
```

Listing 2.2: Interface specification (extract from the ARXML of the model in Figure 2.1).

Listing 2.2 shows the interface specification. The name of the DataElementPrototype (*DataPrototype*) is located in line 4. By applying the port name and the DataElementPrototype to the API naming scheme, we obtain the function call signature `Rte_Write_SendPort_DataPrototype` which matches the signature of the actual generated code, which is shown in Listing 2.3.

```
1 Std_ReturnType Rte_Write_SendPort_DataPrototype(DataType data);
```

Listing 2.3: Communication primitive generated from the ARXML specification in Listings 2.1 and 2.2.

This simple example illustrates some of the key concepts of ARXML parsing. By de-referencing basic building blocks of a component, the function call signature can be derived and located in the implementation code base for instrumentation. The AUTOSAR standard defines more than 20 interface types with a multitude of options, significantly adding to the complexity of the translation process. An exhaustive implementation of all interface types is essential, if maximum compatibility needs to be achieved. Else, a limited subset that resembles the interfaces that are used throughout the concrete model suffices.

2.4.2 *Opportunities for Instrumentation*

The AUTOSAR standard document *Requirements on RTE Software* defines that the „RTE shall be generated in C“ and that the „RTE is required to support components written using the C and C++ programming languages“ [AUT14f]. Thus, C and C++ are the prevalent programming languages in AUTOSAR systems, and we focus on these for instrumentation. Examining the characteristics of the C/C++ programming languages, both languages allow either source code, header file, or binary object, as possible options for instrumentation, which correspond to the software access levels white-box, grey-box

and black-box. SW-Cs and the RTE are instrumented in a *technically* similar manner, hence we will refer to them collectively. A clear distinction has to be drawn during the evaluation and discussion of each approach though, as the instrumentation of SW-Cs and RTE differs in semantics and requirements, as discussed in Section 2.6. Thus each instrumentation location (RTE and SW-C) has three instrumentation options at the code access level, as:

Option 1: Instrumentation of Source Code (.c-files)

The source code of a component contains its *implementation*, which is located in a .c-file or .cpp-file. Source code instrumentation demands white-box access to the instrumented component and therefore has the highest requirements in terms of accessibility.

In order to instrument a component's implementation, i.e., its .c-file, all invocations of the interface function that is to be instrumented, must be replaced with calls to a wrapper. This is done by renaming all calls to *Interface_Name* to a unique and unused function name, e.g. *Wrapper_Interface_Name*. An implementation of *Wrapper_Interface_Name* then has to be provided in a separate .c-file that replicates all `#include` statements of the original .c-file (e.g., for type definitions and macros) and transparently invokes the original API function *Interface_Name*, by passing all parameters and the return value.

Option 2: Instrumentation of Header File (.h-files)

The header file of a component contains its *interface declaration*, which is located in a .h-file. Header file instrumentation requires grey-box access to the instrumented component, as the interface declaration must be accessible, but knowledge of implementation specific details is not necessary.

The interface declaration of a component, i.e., its .h-file, is instrumented by redeclaring the interface name of the function that is to be instrumented (e.g., *Interface_Name*) to a new, unique and unused function name (e.g., *Original_Interface_Name*), effectively hiding the original interface from the implementation. Similar to the instrumentation of source code, an implementation of *Interface_Name* has to be provided in a separate .c-file, that `#includes` the original header file, and invokes the original API function *Original_Interface_Name* transparently.

Option 3: Instrumentation of Binary Object (.o-files)

The binary object of a component contains its *compiled object code*, which is located in an .o-file. The instrumentation of binary objects only requires black-box access to the instrumented component and therefore has the lowest requirements in terms of accessibility.

Binary objects contain tables with information on imported and exported symbols that are used by the linker during the link phase of a program. Symbol tables can be accessed and manipulated by tools such as *objdump* and *objcopy*, both part of GNU Binutils [GNU]. By modifying the import/export table of the binary object, the linker can be instructed to link all calls of the original interface function (e.g. *Interface_Name*) to a wrapped version of the function (e.g. *Wrapper_Interface_Name*). An entry in the symbol table can be redefined, by calling *objcopy* with the `-redefine-sym` parameter, passing the original and new symbol name as additional parameters. An implementation of the wrapped interface function *Wrapper_Interface_Name* has to be provided in a similar way as for above approaches in a separate .c-file.

2.4.3 *Automating AUTOSAR Wrapper Generation*

We have implemented aforementioned instrumentation methods into a prototype AUTOSAR instrumentation framework, which was developed in C#. The development of this framework was motivated by our need for a flexible, configurable and programmatic process to drive the instrumentation of AUTOSAR systems for our own fault injection experiments. At the same time, we also wanted to verify that we can indeed achieve a *usable*, *customizable* and *efficient* approach to instrumentation.

The overall workflow is divided into three phases: parsing, configuration, and instrumentation. Model parsing is key in providing *usability* to the user, as it enables a presentation of the system on the model abstraction level. During model parsing (shown on the left side of Figure 2.4), the parser analyzes the AUTOSAR XML (ARXML) file(s) for elements that are relevant to create this presentation and to drive instrumentation. Beginning at the top-level composition (TLC), which contains the component instances (CIs) of the system, the parser de-references the component prototype specification (CPS) of all CIs.

Next, the parser extracts all references to interface type(s) and data element type(s) that are part of the component prototype. The information contained within the interface type specification, the data element type specification and the specification of the internal behavior of the component prototype are relevant for the overall process. Due to the complexity of the standard, it is not feasible to give a more detailed list of elements. Instead we advise the reader to consult the *Specification of RTE* [AUT14h] that lists all interface types and their associated signatures.

After parsing the ARXML file, we provide the user with a browsable list of the software components that compose the system and their corresponding interface functions, shown in Figure 2.5. During

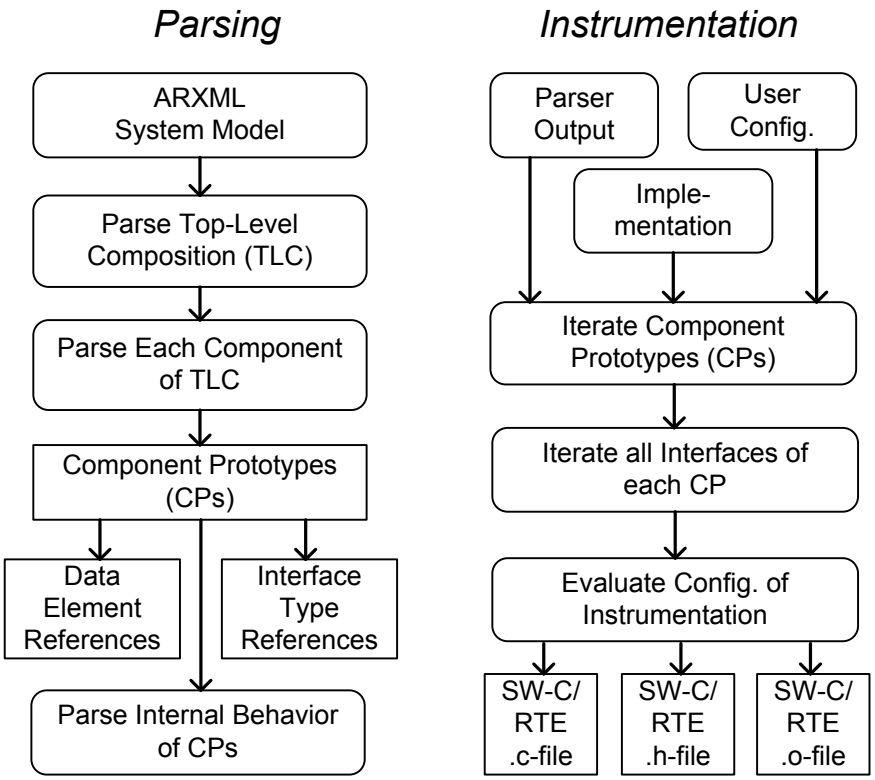


Figure 2.4: Automating instrumentation: Basic workflow of the model parsing and instrumentation phases.

the configuration phase, the user can select the various instrumentation methods (.c-file, .h-file or .o-file) and locations (SW-C or RTE) for each interface of an SW-C, as derived from the component specification in the system model, and supply code for wrapper functionality (e.g., monitor or FI). By offering the various choices of methods and location, we provide the user a *customizable* way to drive instrumentation.

Lastly, in the instrumentation phase (shown on the right side of Figure 2.4), all interfaces of each CPS are iterated, and, depending on the configuration, a method- and location-specific procedure that generates the wrapper and instruments the interface is called. A log file of the instrumentation is generated, to provide feedback and report errors.

By integrating a presentation of the various instrumentation options on the model's abstraction layer, while preserving the flexibility and customizability of working directly on the implementation level, we were able to satisfy the requirements of *usability* and *customizability*. The evaluation of the *efficiency* of our approach is provided in the following section.

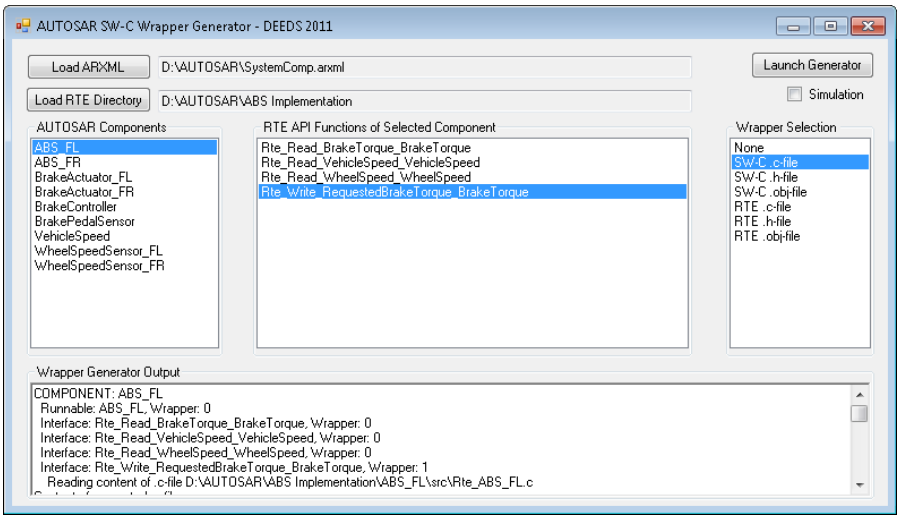


Figure 2.5: Automating instrumentation: The graphical user interface for the configuration phase.

2.5 PROOF OF CONCEPT AND EXPERIMENTAL EVALUATION

This section provides a proof of concept for our suggested instrumentation approach in a typical dependability assessment scenario. We apply the source code and binary object instrumentation options to the SW-C and RTE layers, in order to conduct a series of fault injection (FI) experiments on a simplified anti-lock braking system (ABS). The purpose and intent of these experiments is not so much the evaluation of a single, specific system, but rather to apply all of the instrumentation methods in a common application scenario to show their generic applicability. We determine the overhead of each instrumentation technique, in order to establish a relative comparison and raise the reader's awareness for the different evaluation criteria. This is a best effort approach, as, given the multitude of platforms, systems and tool-chains in the automotive domain, a generic analysis is infeasible.

For development, implementation and evaluation of the system, we used the commercial AUTOSAR tools *ETAS INTECRIO V3.2.0 Hotfix 5* [ETA] and *OptXware Embedded Architect (EA) V1.0.0.201103031241* [Opt], which enabled us to *cross-validate* our results. Although we have conducted all of our experiments on both tools, it is neither our intention nor feasible to provide a comparison of tools, due to the diverse functionality and application area of each tool. Instead, we intend to provide a relative comparison of instrumentation approaches per tool, and we aim at showing that our approach is a generic one, therefore not limited to a certain tool or implementation. For the automated instrumentation of the system, we employ the instrumentation framework prototype that we have developed according to the technical details given in Section 2.4.

2.5.1 *The Experimentation Setup*

The system on which we implement the proof of concept is a simplified anti-lock braking system, as shown in Figure 2.6. It is simplified in the sense that only two wheels are present in the model and that the internal behavior is not used in a production system (i.e., a real car). The system is nevertheless a complete and fully-fledged AUTOSAR system, aligning well with the intent of our experiments. A detailed description of the function of the system and the operating conditions is given in Section 2.5.2. In our setup we aimed to cover the instrumentation locations SW-C and RTE and the wrapper implementation options *.c-file*, *.h-file* and *.o-file*. By applying each of the 3 implementation options to each of the 2 instrumentation locations, we have 6 distinct experimentation setups possible, of which we were able to evaluate 5. We were unable to apply, and therefore experimentally validate, the *RTE .h* instrumentation method, as both tools generate the RTE's header file without interface prototype declarations.

For each of our experiments, we compare the experiment's outcome to a golden run made on a *reference setup* without any instrumentation. To map the 5 experimentation setups to the actual system, we decided to instrument the system with a fixed set of monitors and shift the fault injector location, based on the current setup. The instrumentation methods and fault injector locations are as follows:

- **No instrumentation** Reference setup.
- **SW-C .c** *BrakePedalPosition* in SW-C *BrakePedalSensor*.
- **SW-C .h** *BrakeTorque_FL* in SW-C *BrakeController*.
- **SW-C .o** *WheelSpeed* in SW-C *WheelSpeedSensor_FL*.
- **RTE .c** *VehicleSpeed* in SW-C *VehicleSpeed*.
- **RTE .h** Not evaluated.
- **RTE .o** *RequestedBrakeTorque* in SW-C *ABS_FL*.

For each setup, we have implemented three classes of wrapper behavior: (a) skeleton, (b) monitor, and (c) monitor with fault injector combined. In this context skeleton means that the wrapper implements no other behavior than pass-through. Comparing the reference system with a system implementing skeleton wrapper behavior gives information about the *overhead* of the instrumentation itself, while comparing the monitor and fault injector with the skeleton behavior, gives information about the *implementation efficiency* of the behavior. As we will see in the results later on, an inefficient implementation of wrapper functionality has a much higher impact on system overhead than the instrumentation itself.

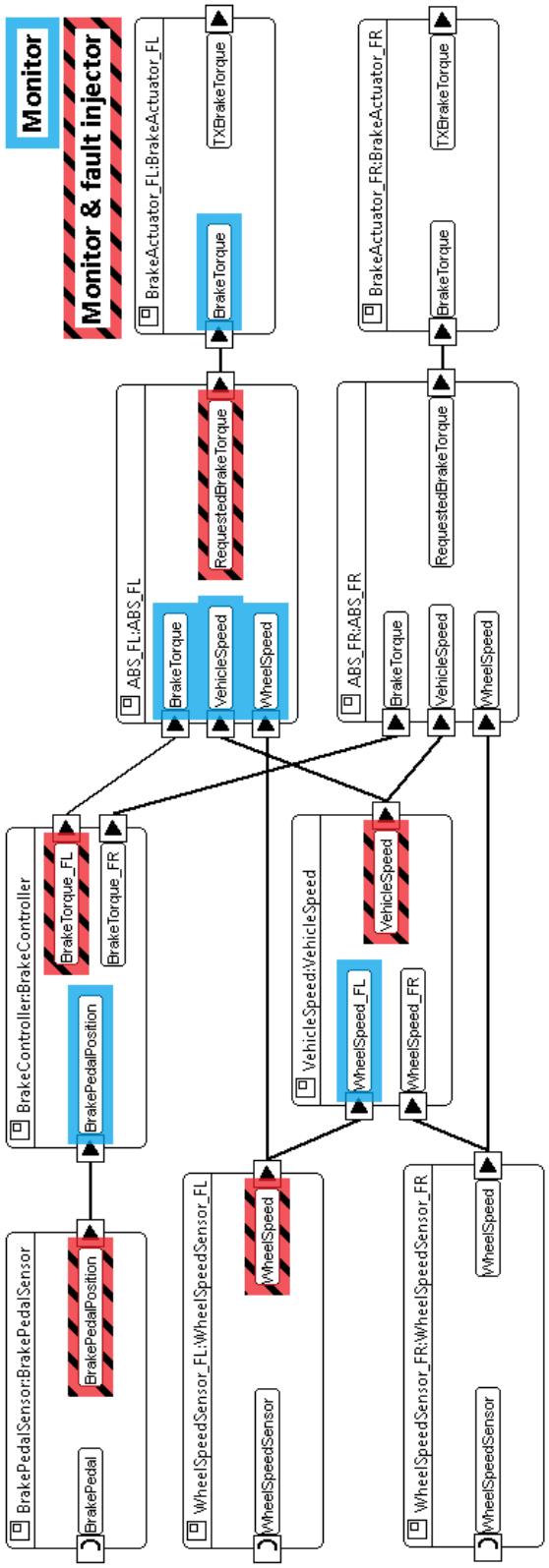


Figure 2.6: Model of an anti-lock braking system (ABS), instrumented with monitors and fault injectors at selected interfaces.

2.5.2 *ABS System and Simulator in a Nutshell*

The ABS system we consider consists of nine SW-Cs and is embedded in an environment simulator, which provides stimuli to the system and receives reactions from the system. In our case, these stimuli are the input values of the brake pedal sensor and the two wheel speed sensors. The system reacts to these stimuli by applying a certain brake torque to each wheel.

The test case we simulate is a full braking from 50 km/h to 0 km/h with a deceleration of $-7\,\mathrm{m/s^2}$ in the optimal case of non-blocking wheels and $-6\,\mathrm{m/s^2}$ in the blocking case. The runnables within the system's components are scheduled periodically every 20 ms and implemented as follows.

BrakePedalSensor polls the *BrakePedal* I/O port for a brake pedal position value, which is provided by the simulator as input stimulus. After scaling and converting the pedal position value to a suitable data type, it is sent to the *BrakeController*, which provides per-wheel brake torque values to the front left (*ABS_FL*) and front right (*ABS_FR*) ABS controllers.

Depending on the individual *WheelSpeed*, *VehicleSpeed* and *BrakeTorque*, *ABS_FL* and *ABS_FR* calculate a per-wheel brake torque that maximizes the brake retardation for the given input values. The brake torque is then sent to the wheel's respective *BrakeActuator* and fed back to the simulator, which calculates this period's deceleration based on the current simulation state and the applied brake torque.

2.5.3 *Fault Injection Experiment*

To show the application of our approach in a typical dependability assessment scenario, we conduct a series of fault injection (FI) experiments on the presented ABS system. FI [HTI97] is a widely accepted technique for experimental robustness evaluation and is applicable at varied component and interface levels. For our evaluation, we utilized SWIFI (Software Implemented FI) to instrument the software component under evaluation (CUE). During the SWIFI experiment, the data sent to a CUE via its interface is intentionally modified in a systematic way, i.e., a fault is introduced, with the intent to expose the CUE to unexpected input. Subsequently, the CUE's behavior, in response to the injected fault, as well as the overall effect on the system, is analyzed.

In order to verify the effectiveness of each instrumentation method, we utilize each to instrument the system with a set of monitors and a fault injector. We then conduct a series of injection runs on the instrumented system, by flipping a single bit of an intercepted data value when a certain trigger condition is met. In our setup, the fault injection is time-triggered at a model time of 300 ms after the simu-

lator has initiated a full application of the brake. As all interfaces in our example system transmit 16-bit values, each injection campaign consists of 17 runs; one golden run that we use for reference, and 16 fault injection runs in which we individually flip a bit at a distinct position of a 16-bit wide data value. For each test run, we compare the output of the interface monitors against the golden run in order to determine whether the monitors were able to detect the inserted fault, and to analyze its impact on system behavior.

Before presenting our results, our choice of the single bit flip *fault model* requires a short discussion on its relevance and representativeness. AUTOSAR is a new standard that manufacturers are just starting to adapt and use in production systems. Therefore the knowledge on actual fault types within those systems is severely limited, and consequently so is the knowledge on fault models. Whether this, or other fault models, are realistic or relevant for AUTOSAR is an interesting question that currently can not be answered due to the novelty of the system and the lack of respective (experience) data. To analyze the relevance of various fault types in AUTOSAR is potentially an interesting field of future research for the dependability community. For our instrumentation approach this has the implication that we can currently only assume that there are faults in AUTOSAR that can be addressed by FI at the interface level.

Having said that, the results of our experiments are listed in Table 2.1. For each test run, we provide the number of detected deviations from the golden run as a measure of the fault's overall impact on the system. The error persistence indicates, for how long the fault's effects were detectable in the system. In summary, all the fault injections, for each test setup and instrumentation method, manifest as detectable deviations from the golden run thus verifying the effectiveness of each approach. Injections into the lower 8 bits have only minor impact on system behavior and are tolerated by the system within one or two periods of execution time. Of all tested interfaces, *WheelSpeed* and *VehicleSpeed* are most susceptible to variations in the lower bit range. The peak value of detected deviations, on the other hand, is reached by injecting into the upper range of most significant bits of the *BrakePedalPosition*, *RequestedBrakeTorque* and *WheelSpeed* interfaces. The repeatedly measured cutoff of the error persistence at 1,940 ms is owed to the car being at full stop at that time.

It is noteworthy to highlight that the AUTOSAR component robustness assessment coverage for the number of detected deviations across all bit positions were similar for both SW-C and RTE, and at the .c, .h and .o levels. The deviation stems in each case from a variation of the fault injector location and not from a conceptual weakness or strength of one or the other approach. This result is important as the equivalent dependability coverages result in giving the system eval-

Table 2.1: SWIFI experiments: Detected deviations and exposure times for different injection locations and bit flip positions.

Injection location	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BrakePedalPosition																
Detected deviations	4	4	4	4	4	4	4	4	4	4	4	4	4	4	454	644
Error persistence (ms)	40	40	40	40	40	40	40	40	40	40	40	40	40	40	1840	1940
BrakeTorque_FL																
Detected deviations	2	2	2	2	2	2	2	2	2	2	2	2	2	2	452	452
Error persistence (ms)	40	40	40	40	40	40	40	40	40	40	40	40	40	40	1840	1840
WheelSpeed																
Detected deviations	3	3	3	3	3	428	3	428	449	458	243	458	243	243	647	243
Error persistence (ms)	20	20	20	20	20	1840	20	1840	1840	1840	1940	1840	1940	1940	1940	1940
VehicleSpeed																
Detected deviations	2	2	2	2	2	2	2	2	436	451	242	457	457	457	454	457
Error persistence (ms)	20	20	20	20	20	20	20	20	1840	1840	1940	1840	1840	1840	1840	1840
RequestedBrakeTorque																
Detected deviations	2	2	2	2	2	2	2	2	5	5	5	425	5	647	645	645
Error persistence (ms)	40	40	40	40	40	40	40	40	40	40	40	1840	40	1940	1940	1940

uator the desired instrumentation choices as based on the access and implementation/execution criteria of Section 2.5.4 and Section 2.6.

2.5.4 *Instrumentation Overhead*

The instrumentation of a system obviously entails overhead either in space (e.g., memory consumption) or time (e.g., execution time). In this section, we determine the overhead of each instrumentation technique in three categories: implementation, runtime, and memory. Given the multitude of platforms, systems, and tool-chains in the automotive domain, this is a best effort approach that aims to establish a relative comparison between the instrumentation methods and raise the reader's awareness for the different evaluation criteria.

2.5.4.1 *Implementation*

Implementation overhead describes the expected time and effort to implement an approach by hand. We measure the implementation overhead of each instrumentation method using SLOCCount [Whe], a set of tools for counting physical source lines of code (SLOC). Table 2.2 and Table 2.3 list the SLOC of each component and the RTE respectively, for various instrumentation methods.

As the numbers for SW-C reveal, *SW-C .h* has the highest implementation overhead, followed by *SW-C .c* and *SW-C .o*. Recalling from Section 2.4, this is not surprising as *SW-C .h* requires the redeclaration of interfaces, the implementation of interface wrappers and the declaration of the interface wrappers. Implementing *SW-C .c*, the redeclaration of interfaces is not part of the process, whereas *SW-C .o* only requires the implementation of interface wrappers.

For the RTE the figures show a different picture, due to the way the tools generate the RTE. As a declaration of interfaces is omitted in the generated code, *RTE .c* and *RTE .o* both only require the implementation of interface wrappers, and therefore share the same overhead.

As the overhead of functionality within the wrappers depends on their implementation, no general statement on their overhead can be made. To provide an example though, the monitors we use consume 1 SLOC per wrapper, whereas the fault injector consumes 9 SLOC.

2.5.4.2 *Runtime*

We employ ETAS INTECRIO and OptXware EA to simulate actual system behavior on a PC platform. As both tools do not provide an accurate emulation of the time of the simulated target system, we use the Windows API function *QueryPerformanceCounter* to measure the current CPU tick count, eventually establishing a *relative* comparison of the runtime of the different approaches. Figure 2.7 and Figure 2.8 depict boxplots of the accumulated runtime of all instrumented in-

Table 2.2: Overhead in source lines of code (SLOC) of instrumented software components for different instrumentation methods.

Instrumentation	ABS_FL	BrakeActuator_FL	BrakeController	BrakePedalSensor	VehicleSpeed	WheelSpeedSensor_FL
ETAS INTECRIO						
None	144	48	62	51	98	49
SW-C .c	+26	+8	+14	+8	+14	+8
SW-C .h	+30	+9	+16	+9	+16	+9
SW-C .o	+22	+7	+12	+7	+12	+7
OptXware EA						
None	141	45	59	48	95	46
SW-C .c	+26	+8	+14	+8	+14	+8
SW-C .h	+30	+9	+16	+9	+16	+9
SW-C .o	+22	+7	+12	+7	+12	+7

Table 2.3: Overhead in source lines of code (SLOC) of instrumented RTE for different instrumentation methods.

Instrumentation	ETAS INTECRIO	OptXware EA
RTE .c	+67	+67
RTE .h	not evaluated	
RTE .o	+67	+67

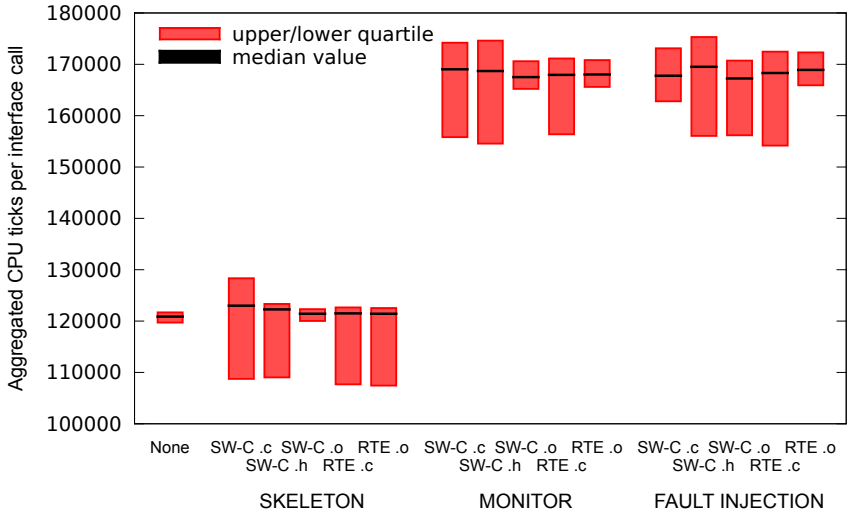


Figure 2.7: ETAS INTECRIO: Relative comparison of the execution time of instrumentation methods, grouped by implemented functionality.

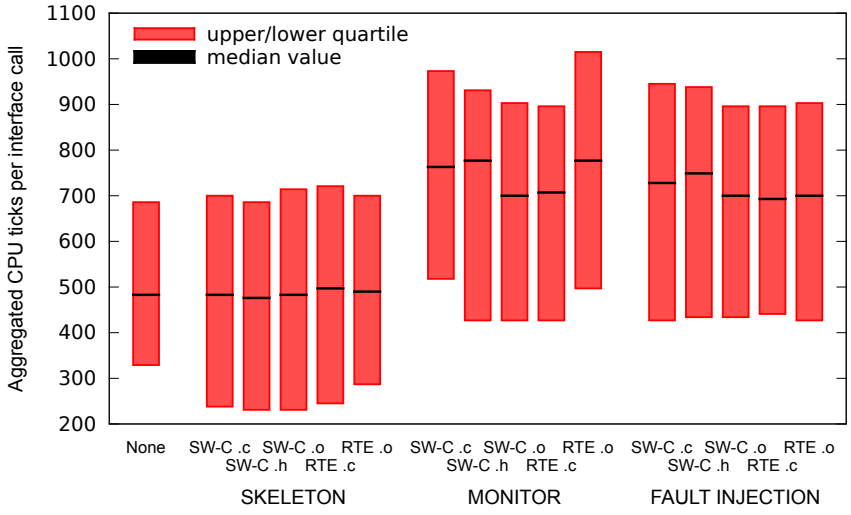


Figure 2.8: OptXware EA: Relative comparison of the execution time of instrumentation methods, grouped by implemented functionality.

terface calls in CPU ticks, for ETAS INTECRIO and OptXware EA, respectively. The boxplots' quartiles are at 25 % and 75 % of measured runtimes. The median value is at 50 % and marked by a black line. The difference of magnitudes in the CPU ticks scale is caused by the different simulation approaches of each tool. While OptXware EA directly simulates RTE and application layer behavior, ETAS INTECRIO executes the target system on a virtual PC target.

The measurements show that the instrumentation method has no significant impact on the overall runtime. Visible minor variations can be attributed to slight deviations of system load during the experiments, caused, e.g., by background applications. Also, the sole instrumentation with wrappers without implemented functionality (skeletons) causes only slight overhead below 1 %.

The main contributor to runtime overhead is therefore the runtime of the functionality that is implemented in the wrappers. In our monitor implementation, we directly write the monitored values to the disk in each invocation. As disk I/O is an expensive operation, we measure an overhead of about 50 % for OptXware EA and about 38 % for ETAS INTECRIO. The fault injector on the other hand, adds no measurable overhead. As the above example shows, providing a time-efficient implementation of wrapper functionality is crucial in real-time systems. It should also be noted that the actual systemwide overhead is considerably smaller, as the above percentages are relative to individual interface calls.

2.5.4.3 *Memory*

We evaluate the overall instrumentation memory overhead, which consists of added code segment size and data segment size, with the tool *objdump*, which is part of GNU Binutils [GNU]. The object files of each of the system's components were compiled without optimizations (compiler switch `-O0`), in order to have a worst case estimation and to disregard compiler specific optimizations.

Our analysis shows that the instrumentation with wrappers causes no data segment size overhead and that the text segment size overhead is independent of the instrumentation method. A detailed breakdown of components' text segment size and the introduced relative overhead is provided in Table 2.4.

The figures show that the relative overhead in text segment size ranges between 1.5 % and 15.0 % per wrapper, and is therefore largely dependent on the implementation complexity of each component. In absolute values, each wrapper consumes approximately 33 bytes for ETAS INTECRIO and 30 bytes for OptXware EA. This difference is caused by the different compilers used by each tool, with INTECRIO relying on *MinGW GCC 3.4.2 (mingw-special)* and EA relying on *Cygwin GCC 3.4.4 (cygming special)*.

Table 2.4: Text segment size of the (instrumented) object files of various software components in bytes.

	ETAS INTECRIO				OptXware EA			
Objectfile	**Text segment size**		**Overhead (%)**		**Text segment size**		**Overhead (%)**	
	plain	instrumented	overall	per wrapper	plain	instrumented	overall	per wrapper
Rte_ABS_FL.o	1808	1920	6.2	1.5	1808	1920	6.2	1.5
Rte_BrakeActuator_FL.o	336	368	9.5	9.5	328	356	8.5	8.5
Rte_BrakeController.o	512	576	12.5	6.3	504	564	11.9	6.0
Rte_BrakePedalSensor.o	320	368	15.0	15.0	328	364	11.0	11.0
Rte_VehicleSpeed.o	896	960	7.1	3.6	892	952	6.7	3.4
Rte_WheelSpeedSensor_FL.o	336	384	14.3	14.3	336	372	10.7	10.7

Table 2.5: Relative comparison of instrumentation method and location for different quality attributes.

	Instrumentation method			Instr. location	
Attribute	.c-file	.h-file	.o-file	RTE	SW-C
Intrusiveness	★	★★	★★★	★	★★★
Implementation effort	★	★★	★★★	★★★	★★
Automation complexity	★★★	★★★	★	★★★	★★
Required system access	★	★★	★★★	★	★★★
Scalability / usability	★	★	★★★	★★★	★★

Legend: ★ poor; ★★ fair; ★★★ good

2.6 DISCUSSION

The experimental results of the previous section have shown that all instrumentation methods are comparably effective to enable the implementation of dependability assessment techniques at the component level, and therefore have to be considered equally viable. Consequently, we draw the conclusion that qualitative aspects can become the determining factor in choosing the right instrumentation option and location. To this end, we discuss the qualitative characteristics of each instrumentation method, with the intention to guide the evaluator in his decision of *how* and *where* to instrument a system and which tradeoffs to consider. In the second part of the discussion, we cover the current limitations of our approach, specifically for multiple component instantiations and shared memory communication.

2.6.1 *Qualitative Aspects of SW-C Instrumentation Methods*

In the following, we introduce a set of quality attributes, which we use to establish a qualitative comparison between the different instrumentation methods. A summary of the comparisons is provided in Table 2.5.

Intrusiveness describes to which degree the instrumentation penetrates the system. Thereby, we consider the system viewpoint (i.e., which layer is affected and what is the layer's criticality) and the implementation viewpoint (i.e., which parts of the implementation are changed). Although the instrumentation with wrappers is an automatic process, the implementation of functionality within the wrappers is a manual or semi-automatic process and therefore error-prone. To minimize possible negative effects of such errors, a low intrusiveness is desirable. Due to the RTE's vital role as communication hub, approaches targeting the RTE are consequently considered more intrusive than the ones targeting the SW-C. Furthermore we consider

changes to the actual implementation more intrusive than changes to the interface declaration or the link information of the object file. The least intrusive instrumentation method is therefore *SW-C .o-file* and the most intrusive one is *RTE .c-file*.

Implementation effort considers the amount of changes entailed by each instrumentation method and serves as an estimate for the effort of manually instrumenting a system, as well as the amount of changes induced by automatic generation. With reference to the technical implementation details of Section 2.4.2, we assess that the instrumentation of *.c-file*s requires a higher effort than *.h-file*s and *.o-file*s. Also, instrumenting the RTE generally requires less effort than instrumenting SW-Cs, as SW-Cs reside in distributed locations, whereas the RTE resides in a central location. Therefore, the least implementation effort is required by *RTE .o-file* and the most by *SW-C .c-file*. For a general estimate of implementation effort, it should be kept in mind that regardless of the effort of wrapper instrumentation, the effort of implementing functionality into the wrappers has to be considered as well.

Automation complexity provides an estimate of the effort to implement the instrumentation method into a generator. During the implementation of the wrapper generator presented in Section 2.4.3, we made the experience that binary instrumentation is the most complex generation task to implement. This is due to the black-box constraint put by binary objects, which requires the deduction and generation of the complete interface declaration from the system specification contained in the system's ARXML file. The implementation (.c-file) and interface declaration (.h-file) on the other hand, both contain the declaration, either implicitly or explicitly, making this generation step obsolete, and only requiring a technically similar parsing of source files. Due to the central location of the RTE, mentioned in the previous paragraph, the least automation complexity is required by *RTE .c-file* and *RTE .h-file* and the most by *SW-C .o-file*.

The **required system access** characterizes the requirements of each instrumentation method on the accessibility and visibility of the system and its implementation. We distinguish *white-box*, i.e., all source code is accessible to the system evaluator, *grey-box*, i.e., parts of the source code (e.g. header files) are accessible, and *black-box*, i.e., no source code is accessible. Furthermore, we distinguish between SW-Cs and the RTE, with access to the RTE usually being available to the integrator only. Accordingly, *RTE .c-file* has the highest requirements on system access and *SW-C .o-file* the lowest ones.

Scalability describes, how well each instrumentation method scales to larger systems. As the scalability of an instrumentation method has a high influence on its **usability**, we consider them collectively. The main overhead in large scale projects can be accounted to the configuration of the instrumentation and the system build process, and

not the instrumentation itself. Therefore, all instrumentation methods scale comparatively well, with a slight advantage for methods targeting the RTE (due to its central location), and a notable advantage for black-box instrumentation methods. As black-box methods do not modify any source code, a time-consuming recompilation of code can be omitted, and only the linkage of binary objects has to be performed.

Summarizing our observations in Table 2.5, there is a clear trend showing advantages in all categories for black-box instrumentation over grey-box and white-box, except for automation complexity. The choice of instrumentation location, i.e., whether to instrument the SW-C or RTE, is not as clear though. As SW-C has advantages in the categories intrusiveness and required system access, and RTE has advantages in the other categories, the determining factor is, how each category is weighted by the system evaluator, also considering his software access level and the application scenario.

2.6.2 *Limitations*

During experiments with different AUTOSAR example systems, we realized that there exist two classes of systematic limitations to our proposed approach of wrapping AUTOSAR components. The first limitation only affects a subset of the presented instrumentation methods and is related to the possible implementation of the communication between SW-Cs and the RTE via shared memory. The second limitation affects all of the presented methods, but is only relevant in systems which make use of multiple instantiations of a component. Both limitations are discussed in the following.

2.6.2.1 *Shared Memory Communication*

The communication between SW-Cs and the RTE is not necessarily always implemented via function calls (which was our intuitive assumption) but can also be implemented via shared memory communication for performance reasons. Whether communication is implemented via function calls or shared memory, depends on the implementation of the RTE generator (and is therefore tool dependent) and the communication interface-type (e.g., implicit/explicit access, client/server or sender/receiver model, etc.) and its configuration.

The implementation of the communication mechanism impacts the applicability of some of our approaches. Namely, *SW-C .o-file*, as well as all RTE instrumentation methods, are unable to cover shared memory communication. In order to cope with shared memory communication, we propose two feasible workarounds. The first workaround is *runnable wrappers*, which can be implemented on the SW-C and RTE side. The second one is a *task-based monitor* which is implemented by

associating a monitoring task on the operating system level, with the runnable to be monitored. Runnables are the executable parts of a software component that implement actual functionality. By wrapping a runnable's invocation, we are able to access all data that the runnable has access to via shared memory, either before (relevant for reads) or after (relevant for writes) the runnable invocation.

As both workarounds are conceptually different from interface wrappers, we did not include them in our evaluation. Preliminary experimental results show that both approaches are suitable for systematic and automatic instrumentation of AUTOSAR systems using shared memory communication.

2.6.2.2 *Multiple Instantiation of Components*

Our approach is also limited for models that make use of multiple instantiations of component prototypes and therefore employ code-reuse. The issue in such a scenario is that we are currently unable to distinguish between different instances of a component prototype, as the interface implementation (due to code-reuse) is only present once, irrespective of the number of instances. For each component instance, the RTE holds a unique data structure (termed RTE instance) that is passed to the component runnables as a parameter on invocation. As these data structures contain no naming information, it is difficult to obtain self-awareness for the active SW-C instance.

A simple workaround is to move the instrumentation location from the receiving SW-C's interface to the sending SW-C's interface, or vice versa. This workaround is only feasible as long as the component that the instrumentation is moved to is not a multiply instantiated component itself. Due to the fixed memory layout of automotive systems, an alternate approach leveraging pointer address information of the RTE instance data structures, to distinguish between multiple instances, is conceivable. We will address this and alternate solutions in future work.

2.7 CONCLUSION

In this section, we have shown how to develop a usable, customizable and efficient instrumentation framework for the dependability assessment of AUTOSAR systems. Our approach provides *usability* as we enable the user to define their instrumentation requirements on the model level instead of the implementation, which largely consists of automatically generated code. Our approach is *customizable* as we provide expert users with the ability to further tune and refine their instrumentation choice factoring implementation details, and thus test certain *aspects* of an interface, which are represented by the different instrumentation options and instrumentation locations within the software stack. Our approach is *efficient* as the use of

interface wrappers infers low timely and spatial overhead as shown by our experimental results.

Factoring the different software component access levels that are prevalent in AUTOSAR systems (black-box and white-box), we enable the instrumentation at the source code level (i.e., implementation and interface specification) and the binary object level. As proof of concept, we have conducted a series of fault injection experiments on an anti-lock braking system (ABS), which showed the generic applicability of the different instrumentation techniques, providing freedom of choice to the user on the different techniques. The experimental evaluation furthermore yielded the result that the varied techniques were comparably efficient, and the cross-validated fault injection experiments showed that a black-box instrumentation technique was as effective as a white-box technique while requiring less access to the system and being less intrusive. To guide the reader in his decision of instrumentation location and instrumentation options, we have discussed the qualitative criteria of code access, intrusiveness, automation complexity, and implementation effort.

In addition, we have identified systematic limitations of our approach and sketched possible solutions to resolve them. The implementation and evaluation of such solutions is up to future work, as is the instrumentation of other locations in the AUTOSAR software stack, such as the basic software. As our approach is potentially able to implement control-flow monitoring, which will be supported by version 4 of the AUTOSAR standard, with the advantage of finer granularity and the added benefit of data-flow monitoring, we also plan to pursue this option in future work.

Part III

ASSESSMENT

3

ASSESSING SAFETY MECHANISMS EFFECTIVELY USING FAULT INJECTION

The automotive safety standard ISO 26262 strongly recommends the use of fault injection (FI) for the assessment of safety mechanisms that typically span composite dependability and real-time operations. However, with the standard providing very limited guidance on the actual design, implementation, and execution of FI experiments, most AUTOSAR FI approaches use standard fault models (e.g., bit flips and data type based corruptions), and focus on using simulation environments. Unfortunately, the representation of timing faults using standard fault models, and the representation of real-time properties in simulation environments are hard, rendering both inadequate for the comprehensive assessment of AUTOSAR's safety mechanisms. The actual development of ISO 26262 advocated FI is further hampered by the lack of representative software fault models and the lack of an openly accessible AUTOSAR FI framework. We address these gaps by (a) adapting the open source FI framework GRINDER [WPS+15] to AUTOSAR and (b) showing how to effectively apply it for the assessment of AUTOSAR's safety mechanisms. The FI process presented in this chapter advances and builds upon the instrumentation approach that we developed in Chapter 2. The content of this chapter is based on a conference paper accepted at EDCC 2015 [PWS+15b].

3.1 IMPEDIMENTS TO FAULT INJECTION-BASED ASSESSMENTS

In the automotive domain, many innovative functions, such as crash prevention, advanced driver assistance features, and vehicular communication systems, are enabled by software. Consequently, the software code base is growing in both complexity and size, in some cases reaching 100 million lines of code that are distributed across more than 70 electronic control units (ECUs) and interconnected by more than 5 different bus systems [Bro06; Cha09]. In order to manage the complexity of these systems and to reduce the development and integration costs for new vehicle features, the automotive industry widely adopts standardized software architectures and development processes such as AUTOSAR (AUTomotive Open System ARchitecture) [AUT14a].

To warrant the trust that motorists put in the safe operation of their vehicles, many functions in automotive systems are designed and developed following stringent dependability and safety requirements. The recommended guidelines for the design, development

and integration of such systems are provided by the functional safety standard for road vehicles ISO 26262 [Int11]. To aid automotive system developers in meeting these safety requirements, the AUTOSAR standard specifies a set of *functional safety mechanisms* [AUT14e], such as memory partitioning and timing monitoring. The verification and validation of the implementation and application of these functional safety mechanisms, which are usually supplied by third party vendors, is essential to the dependable and safe operation of these systems and to prevent hazards such as Toyota's unintended acceleration issues [Koo14].

Among the available dependability assessment techniques, fault injection (FI) [HTI97] is widely adopted and ISO 26262 strongly recommends its use to validate that functional and technical safety mechanisms are correctly and effectively implemented. Despite this explicit recommendation, published work on AUTOSAR FI (cf. Section 3.3) is currently not addressing the comprehensive assessment of the correctness and effectiveness of AUTOSAR's safety mechanisms. Moreover, the predominantly used standard fault models, such as bit flips and data type dependent corruptions, are limited in their representation of timing and software faults, thus hampering their applicability to such an assessment. We argue that this situation originates from ISO 26262 recommending FI without providing appropriate guidance on the design, implementation and execution of FI experiments. The actual assessment is further hampered by the lack of an openly accessible AUTOSAR FI framework. A similar observation was made by Silva et al. [SBC+13], who conclude that „although many fault injection tools exist, none is really a ready to use tool, thus a common framework would be a major breakthrough". As state of the art, little documented experience exists on how to effectively apply FI for validating the implementation of AUTOSAR safety mechanisms.

Contributions

- Facing these challenges, we provide an open source, ready to use AUTOSAR FI tool [PMT+15] capable of conducting FI experiments at all layers of AUTOSAR's software architecture, i.e., application, runtime environment and basic software. Injections in source code and binary object files (i.e., white-box and black-box) are both supported.
- We report our experiences in conducting a dependability assessment of a commercial implementation of AUTOSAR's timing monitoring safety mechanisms. The assessment uncovered a real deficiency in the implementation that was subsequently acknowledged and fixed by the supplier of the safety mechanisms' implementation.

- We provide guidelines for the derivation of specific fault models, injection locations and mechanisms from the abstract AUTOSAR and ISO 26262 fault models.

We structure this chapter as follows. We give a brief introduction to AUTOSAR's system model and functional safety mechanisms in Section 3.2, followed by a review of related work on AUTOSAR FI in Section 3.3. In Section 3.4, we discuss the AUTOSAR fault models that are currently used or provided by the standard. The adaptation of the open source FI framework GRINDER to AUTOSAR and the instrumentation of AUTOSAR systems is described in Section 3.5. We demonstrate the applicability and effectiveness of FI for the assessment of AUTOSAR safety mechanisms in a case study in Section 3.6, including a detailed specification of the used fault models.

3.2 AUTOSAR: SYSTEM MODEL AND FUNCTIONAL SAFETY MECHANISMS

To familiarize the reader with basic concepts of AUTOSAR, this section provides a brief introduction to its system model and functional safety mechanisms. The description of the system model condenses information from Section 2.2 to provide sufficient context for the description of the functional safety mechanisms. For a thorough treatment of the system model, we recommend to revisit Section 2.2.

3.2.1 *System Model*

AUTOSAR systems are designed as abstract models, in which *software components (SW-Cs)* are the core building blocks. They contain functional entities called *runnables*, whose execution is triggered by recurring timers or aperiodic events (e.g., message arrival). SW-Cs interact with their environment via port interfaces that are connected in the model through a virtual functional bus (VFB).

During the system configuration phase, this abstract representation of the system is subsequently mapped to one or more electronic control units (ECUs). At an ECU, the AUTOSAR software architecture is organized in layered form as depicted in Figure 3.1. Closest to the hardware is the basic software (BSW) layer, which provides hardware abstractions for the microcontroller and the ECU and hosts the operating system (OS) amongst other system and communication services. The runtime environment (RTE) provides the SW-Cs of the application layer with an interface to BSW services. Moreover, the RTE implements a transparent communication abstraction that maps the virtual connections of the VFB to actual communication channels. The application layer comprises a set of SW-Cs, each of which contains one or more runnables. As runnables are not directly schedu-

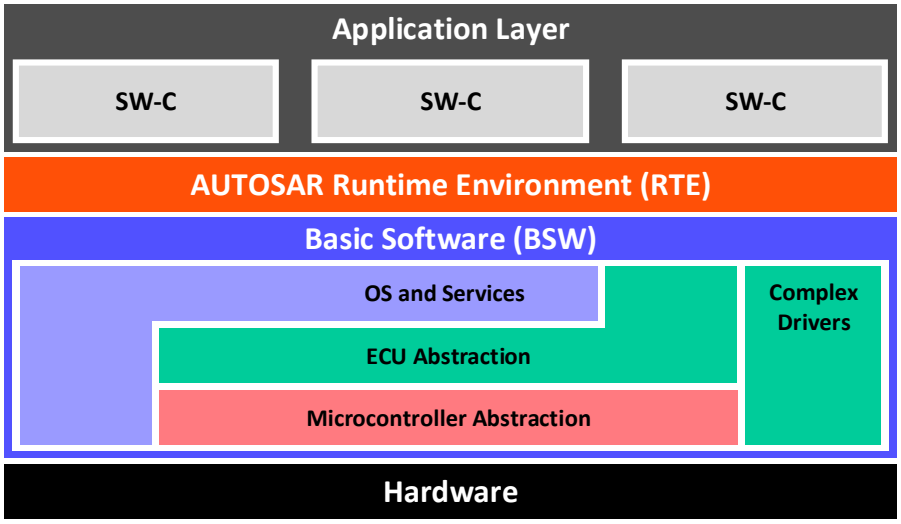


Figure 3.1: The AUTOSAR software architecture (adapted from [AUT11]).

lable by the OS, they are grouped as *tasks* by the system integrator, usually taking the execution periods of the runnables and the criticality of their SW-C into account. To avoid unintentional interactions between tasks of different criticality (e.g., by error propagation), AUTOSAR offers a set of functional safety mechanisms to monitor and isolate tasks, which are described in the following subsection.

3.2.2 *Functional Safety Mechanisms*

For the co-existence of tasks with different criticality, i.e., different automotive safety integrity levels (ASILs), on the same system, ISO 26262 requires *freedom from interference* in both space and time. This means that lower ASIL tasks must not interfere with higher ASIL ones, for example through error propagation. To realize freedom from interference, isolation mechanisms are used to establish fault containment regions. The *Overview of Functional Safety Measures in AUTOSAR* [AUT14e] (a document which recently superseded the *Technical Safety Concept Status Report* [AUT14i]) specifies the following functional safety mechanisms to assist with the prevention, detection and mitigation of hardware and software faults to ensure freedom from interference between tasks.

- **Memory Partitioning.** To prevent low ASIL tasks from wrongfully accessing memory of higher ASIL tasks (e.g., by corrupting their content), arbitrary tasks may be grouped to so-called *OS applications* which are subsequently executed in separate memory partitions, i.e., the code executing in one partition cannot modify memory of a different partition. Memory partitioning allows to protect read-only memory segments as well as memory-mapped hardware.

- **Timing Monitoring.** The safety of a system often depends on the timely execution of actions and reactions such as object recognition for crash avoidance or crash detection for airbag inflation during an accident. The OS provides a set of monitoring mechanisms to detect conformance (e.g., whether tasks are dispatched at the specified time) or deviance (e.g., when tasks violate their execution time budgets or monopolize OS resources).

- **Logical Supervision.** To detect control flow errors, i.e., any divergence of a program's execution sequence from its error-free execution sequence, checkpoints are placed throughout a *supervised entity* at design time. When encountering a checkpoint, the Watchdog Manager is notified and verifies that the sequence of encountered checkpoints is valid. In addition, temporal monitoring mechanisms such as aliveness and deadline monitoring are implemented using checkpoints.

- **End-2-End Protection.** To ensure the integrity of data transmitted between SW-Cs, both within the same ECU as well as across networks, the end-2-end protection library enables the sender to protect data prior to transmission and the receiver to detect and handle errors in the communication link at runtime. Several standardized profiles offer different sets of protection mechanisms (e.g., CRC, sequence counter, alive counter) that are suitable for diverse requirements.

The functional safety mechanisms aim at detecting and mitigating the erroneous behavior of tasks and are implemented by the OS or as BSW services, except for end-2-end protection, which is implemented by a library. Given the broad scope of these mechanisms ranging from the BSW to the application layer, any framework for their comprehensive FI-based assessment naturally benefits from access to the complete AUTOSAR stack for the flexible placement of fault injectors and monitors of system behavior.

3.3 RELATED WORK ON AUTOSAR FI

Following a standalone exposition of state of the art AUTOSAR FI techniques, either based on simulation, hardware, or software, this section summarizes the viability of the different approaches for assessing AUTOSAR safety mechanisms and comments on the utilized fault models.

3.3.1 *Simulation-based FI*

In 2010, Lanigan et al. [LNF10] used the commercial off-the-shelf (COTS) tool Vector CANoe to build a FI framework for AUTOSAR. This early work on AUTOSAR FI primarily comments on the feasibility and practical obstacles of building such a framework and does not report any specific results of dependability analyses of AUTOSAR-based (sub-)systems or provide specific fault models. In CANoe, AUTOSAR systems are executed in a simulation environment on a host PC. Faults can be injected in various components of the basic software (BSW) layer via a set of hooks that either suppress calls to certain API functions (suppression hooks) or manipulate specific data structures of an API (manipulation hooks). Hooks are manually placed in the code and the actual implementation of the fault models is provided by an external library that is linked to the simulation environment. The authors conclude that „CANoe is a suitable fault--injection environment for some faults, but that other faults cannot be represented using the level of abstraction that CANoe provides".

Another simulation based FI framework is from Baumgarten et al. [BOR+14] where application-level software components (SW-Cs) of existing AUTOSAR models are annotated and extended by so called *fault ports*. These extended models are translated to C-code using dSpace TargetLink which is a code generation tool that operates on Simulink and Stateflow models. Faults are implemented as additional code in the TargetLink model and triggered during simulation by signaling the corresponding fault port(s). Consequently, faulty behavior is added in form of functional blocks, which are then able to disturb the normal behavior according to the provided fault implementations. Annotating models with fault ports is currently a manual process and the authors leave automated annotation for future work. The actual simulation can be performed on two different levels, either in the integrated simulation environment of dSpace SystemDesk to simulate the whole AUTOSAR architecture or at the software unit level using test tools for C code. The authors evaluate their approach in a front-lights controller setting, considering physical chip damage, stuck-at, crash and message loss as fault models. The effects of the faults are emulated on the application layer by disabling the execution of SW-Cs, by forcing port interfaces to specific values, or by omitting messages.

To identify threats to functional safety early in the system design stage, Pintard et al. [PFK+13; PFL+14] propose fault injection analyses (FIA) on pre-implementation design artifacts to complement fault injection experimentation (FIE) on an actual (prototype) implementation. To conduct FIA, a functional model is created from the system requirements using data flow diagrams, state charts, UML or AADL, depending on the available level of detail that the specification pro-

vides. Using this model, failure modes of functions or components are assumed and their effect on the system operation analyzed, similar to failure mode and effect analysis (FMEA) and failure mode, effect and criticality analysis (FMECA). Subsequently, the results of FIA are used as guidance to perform FIEs, for example by identifying critical components.

Vedder et al. [VAV+14] develop a method and tool for combining property-based testing (PBT) and FI for testing safety-critical systems. PBT automatically generates test cases from a specified property of a system, i.e., it generates test input values and at the same time acts as an oracle for the expected output. The authors use the commercially available tool QuickCheck for generating the test cases in conjunction with their own tool FaultCheck, a C++ library with an optional wrapper for C, that conducts the actual injections. The tool is evaluated on an AUTOSAR E2E library implementation in an isolated simulation environment.

Overall, while simulation-based approaches are useful to conduct FI analyses and experimentation in the early stages of system development, they are inherently limited in their representation of detailed lower-level fault models and timing conditions necessary for addressing real-time operations. As the meaningful application of many safety mechanisms depends on an exact representation of time and real-time constraints, a comprehensive assessment of timing-dependent safety mechanisms is infeasible using simulation-based approaches.

3.3.2 *Hardware-based FI*

Cunha and colleagues [Cun13; BSC13] developed the fault injection tool csXception® for scan-chain implemented fault injection (SCIFI) on an ARM Cortex-M3 microcontroller, which is capable of injecting faults at the hardware level (i.e., processor and auxiliary registers, flash memory and SRAM). The considered fault models are bit flips, reset values and specific values, which can be activated based on various fault trigger conditions. The tool is evaluated in an anti-lock braking system (ABS) case study where the focus of the evaluation is the general application of the tool rather than addressing actual dependability properties of the ABS system.

Salkham et al. [SPS12; SPS13] present a FI framework implemented as AUTOSAR software components (SW-Cs) and complex device driver (CDD). The FI controller and monitoring services are implemented on the application layer as SW-Cs and isolated from the rest of the system using memory partitioning. The CDD contains a collection of FI modules that implement specific error types for each target component. The approach is evaluated in two example scenarios: communication errors that are modeled by disabling the CAN bus circuit

and NVRAM errors that are modeled by corrupting CRC bits. While the experiment control logic is implemented as software elements, hardware mechanisms are utilized to perform actual injections.

Hardware-based FI commonly has the advantage of handling low level implementation details. However, to cover dependability properties that are related to the interaction of hardware and software elements or to accelerate experiments [CB89; CC96], e.g., by using software states as trigger conditions for injections, software based control as in [SPS12; SPS13] is often required. To exercise precise control over software interactions and software timing in our study, we also chose to implement the injection mechanisms in software.

3.3.3 *Software-based FI*

Islam et al. [ISA+13] proposed the BeSafe framework for benchmarking the functional safety of AUTOSAR systems on three different abstraction layers: model, software and hardware. The benchmark framework supports FI at the software level through a proprietary tool called B-FEAT. The tool is capable of intercepting calls to/from SW-C interfaces, facilitating data type and fuzzing error models to evaluate the robustness of SW-Cs. The resilience of SW-Cs with respect to transient bit flip hardware faults is benchmarked using the FI tool GOOFI-2 [SBK10]. On the model level the tool MODIFI [SVE+10] is used to conduct dependability evaluations of Simulink models early in the development phase, mainly to assess error detection and recovery mechanisms. The considered fault models are data type and fuzzing on the software level, bit-flips on the hardware level and bit-flips and sensor faults on the model level. Unfortunately, the preliminary evaluation of the framework has not been conducted on an AUTOSAR system. However, the reported assessment of a CRC mechanism suggests that the framework potentially could be used to assess AUTOSAR's end-to-end (E2E) protection mechanisms.

With the aim to assess the robustness of AUTOSAR COTS components that are available as binaries only, Islam et al. [IKH+14] present a technique and tool prototype for binary-level fault injection (BLFI). Contrary to the binary-level instrumentation approach presented in [PWM+12], no AUTOSAR specific information is used to drive the instrumentation, thereby expanding its applicability to all AUTOSAR layers including the BSW. The broader application scope comes at the cost of losing the reference to the underlying system model, which impacts the usability of the instrumentation if the system model is used to select instrumentation locations. The tool is evaluated on a blinking LED warning system, using data-type based and fuzzing fault models.

3.3.4 *Summary Comments*

We observe that the fault models used in related work are predominantly *standard* fault models[1], such as bit flips and data type dependent corruptions, that have been adopted from studies for different target systems and application scenarios. It is surprising that no *AUTOSAR-specific* fault models are used, as the choice of fault models heavily impacts the effectiveness of the FI experiments [WSS+11; NCD+13] and domain-specific fault models are expected to yield better results.

We also observe that, despite an explicit recommendation in the ISO 26262 standard, no studies on FI-based assessments of the mechanisms in AUTOSAR's technical safety concept exist, apart from the E2E library.

Finally, we observe that many FI approaches are simulation based and as such unable to adequately represent real-time properties and timing behavior of the system. As we demonstrate in Section 3.6, assessing real-time properties is essential to many safety-critical applications and AUTOSAR's timing monitoring mechanisms, which constitute a significant fraction of the technical safety concept.

3.4 AUTOSAR FAULT MODELS

A fault model is a representation of a possible internal or external fault [ALR+04] that a system may be exposed to. In the context of FI experiments, a fault model is characterized by the fault location (*where* to inject), the fault type (*what* to inject), and the fault timing (*when* to inject) that possibly also includes an expected workload or system state. The selection of *representative* fault model(s), i.e., faults that may and do occur during the development and operation of the tested system, is crucial as non-representative faults can significantly affect the injection results [NCD+13] by hampering (a) their accuracy and usefulness [JSM07] and (b) the effectiveness of the experiments to reveal robustness vulnerabilities [WSS+11]. Consequently, developing and using a realistic fault model is one of the biggest challenges for any FI schema [SBC+13].

The use of standard fault models such as *bit flips* and *data type dependent corruptions* is currently predominant in related work on AUTOSAR FI (cf. Section 3.3). While these established fault models adequately represent specific abstraction levels and classes of faults (and their effects), their applicability to represent complex software and also timing faults is limited. For the *effective* assessment of AUTOSAR's safety mechanisms, the use of standard fault models should

[1] In Section 3.4 we further elaborate why the „standard fault models" are still prevalent.

therefore be combined with fault models that allow a more direct mapping of application level software and timing faults.

Before the release of the *Overview of Functional Safety Measures in AUTOSAR* [AUT14e] end of 2014, no central document provided documentation on the applicable fault models for the evaluation of AUTOSAR's functional safety mechanisms. Moreover, representative fault models are invariably based on deep domain knowledge such as known failures, identified hazards and (non-functional) safety, and dependability requirements–in summary information that is not necessarily publicly available to the research community. We argue that both of these factors have contributed to the slow adoption of more representative software and timing fault models for AUTOSAR FI. In addition, an opinion that has been prevalent for a long time in the automotive community is that software faults are generally covered by, and detected during, the verification phase of development, due to the systematic nature of these faults. Given the growing complexity of automotive software, it is questionable whether this view still holds, or for how long.

Even though the release of applicable fault models for AUTOSAR's functional safety mechanisms in [AUT14e] is a step in the right direction, the models are still at a very abstract level as they are directly adapted from ISO 26262, which is a generic standard for road vehicles and not AUTOSAR in particular. Concrete examples of software defects are thus omitted and instead only the effects of faults are exemplified. To give an example, the only information that AUTOSAR and ISO 26262 provide for fault models applicable to *timing monitoring* is „blocking of execution, deadlocks, livelocks, incorrect allocation of execution time, and incorrect synchronization between software elements". This information serves as suggestion for *what* to inject at best, while guidance on the actual application of these fault models, i.e., *where* and *when* to inject, is still missing.

Other documents, such as the *Description of the AUTOSAR standard errors* [AUT14c], improve on this but are still work in progress. While having the purpose of giving an overview of dysfunctional behavior, clarifying error handling mechanisms and giving the failure modes coverage of the different mechanisms, the scope of the document is currently limited to the CAN communication stack and the memory stack. As the specification of AUTOSAR fault models is still an ongoing process, one may fall back to studies on representative software faults from other domains in the meantime [CC96; DM06; NCD+13].

Surprisingly, the consideration of simultaneous fault models is currently missing completely in AUTOSAR, although ISO 26262 explicitly considers dual-point and multi-point failures, i.e., failures resulting from the combination of two or several independent faults that leads directly to the violation of a safety goal [Int11]. Until the standard incorporates multi-point faults, we refer to the work of Win-

ter et al. [WTS+13], who provide a comprehensive study and new approaches to assess the degree to which systems are vulnerable to multiple-fault conditions.

As consequence of the ongoing development of the AUTOSAR standard and the reliance on deep domain knowledge to specify fault models, any FI framework utilized for the assessment of the functional safety mechanisms should be easily and flexibly extensible to account for future standard revisions and domain specific requirements. For the FI framework that we present in Section 3.5, we therefore use a fault model library to offer this flexibility, while at the same time enabling the re-use of fault models that have already been implemented.

3.5 APPLYING THE OPEN SOURCE FI FRAMEWORK GRINDER FOR AUTOSAR FI

Following the discussions on AUTOSAR-specific FI and fault models, we now detail the application of the open-source FI framework GRINDER for AUTOSAR FI. After presenting the FI workflow, we discuss GRINDER's adaptation to our AUTOSAR evaluation environment and comment on the instrumentation methodology for using GRINDER with AUTOSAR.

The workflow of an FI experiment typically comprises the following three phases.

1. **Configuration:** The workload is prepared and the target system is instrumented with injector(s) and detector(s) according to the experiment's specification. The workload should match the evaluation target and also ensure that all injection(s) are activated by meeting their respective trigger condition(s) (i.e., *when* to inject).

2. **Execution:** The target system is executed until the injection of fault(s) and the collection of perturbation data is successfully completed, or until a stop criterion (e.g., a timeout) is satisfied.

3. **Evaluation:** The experiment outcome, for example logs and traces that were collected during its execution, is analyzed.

As this workflow offers much potential for automation, FI frameworks are typically employed for the automated and efficient execution of campaigns (series of experiments) with the positive side-effect of precluding human error from adversely affecting experiment results.

For the assessment of AUTOSAR's functional safety mechanisms, we faced the challenge of which framework to use. Specifically for AUTOSAR FI, none of the existing frameworks is publicly available, nor do the existing frameworks fulfill the requirements for such an

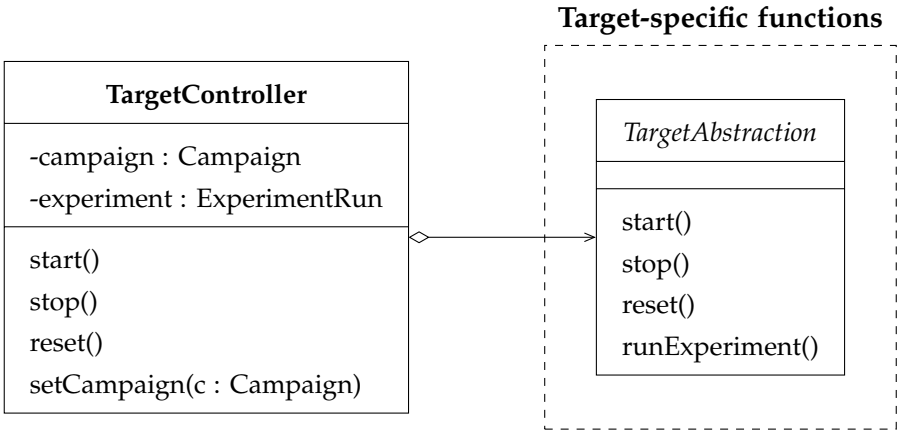


Figure 3.2: The *TargetController* class and the *TargetAbstraction* interface [WPS+15].

assessment (cf. Section 3.3). While considering publicly available generalist tools such as FAIL* [SHK+12] and LLFI [TP13] as potential alternatives, we realized that the adaptation of these tools was infeasible. FAIL* targets architecture simulators for x86 and ARM that do not suit the PowerPC architecture of our evaluation system, and LLFI relies on the LLVM compiler infrastructure, whose use would require the modification of large parts of the existing AUTOSAR development environment and tool chain. In conclusion, the adjustment of openly available tools to suit the purpose of our assessment would likely have outweighed any effort for re-implementing a custom tool from scratch. Consequently, we adapted the open source FI framework GRINDER to AUTOSAR, which entailed modest implementation and configuration overhead [WPS+15]. In the following two subsections, we detail the adaptation of GRINDER to an AUTOSAR system and the instrumentation of AUTOSAR systems for FI experiments.

3.5.1 *Adapting GRINDER to an AUTOSAR System*

GRINDER is an open source general-target FI tool that is written in Java and built around an extensible architecture with a simple interface for target abstraction, extension and customization, and which has reusability as one of its primary design goals. With the intent of making a ready-to-use AUTOSAR FI framework publicly and freely available, we release our AUTOSAR adaptation of GRINDER as open source [PMT+15].

GRINDER is adapted to a new target system (e.g., an AUTOSAR system) by providing a target-specific implementation of the so-called *TargetAbstraction* interface, by which GRINDER interacts with target systems (cf. Figure 3.2). The TargetAbstraction specifies a simple set of target-specific functions that GRINDER's *TargetController* class re-

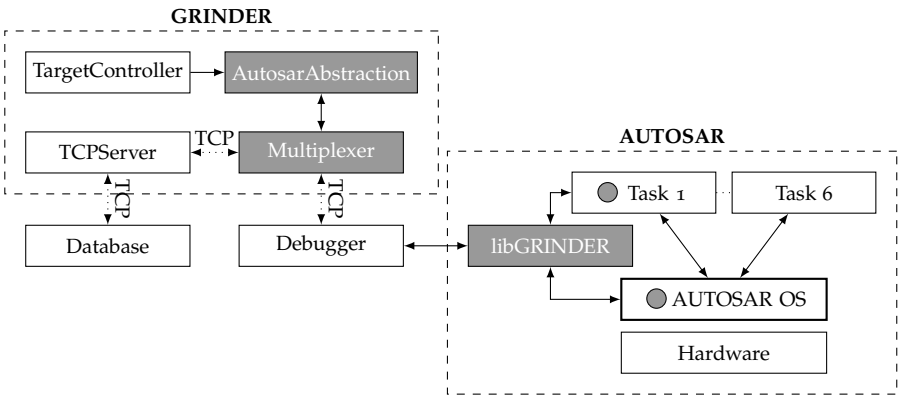


Figure 3.3: Adapting the GRINDER FI framework to AUTOSAR (adapted from [WPS+15]).

quires to control target systems: *start()*, *runExperiment()*, *reset()*, and *stop()*. The specification was driven by the observation that on an abstract level the progression of FI experiments across different tools and targets is the same: target initialization, workload invocation, fault injection and data collection [WPS+15], closely matching the description of the FI workflow in the beginning of the section. After *starting* the target, an experiment is *run* and data is collected for analysis. After experiment completion, the target is *reset* to a known stable state to avoid the impact of undetected residual injection effects on subsequent experiments. These steps are repeated until each experiment of the current campaign has been executed. Afterwards, the target is *stopped* and exchanged or reconfigured, if this is required for subsequent experiments.

GRINDER's architecture resembles the general FI tool architecture presented by Hsueh et al. [HTI97]. Its integration in an AUTOSAR FI setting is depicted in Figure 3.3 where components that had to be either developed or adapted for GRINDER's use with the AUTOSAR system, i.e., the *AutosarAbstraction*, the *Multiplexer* for communication with the target, and the fault model library *libGRINDER*, are colored gray. The details on each of these components are provided below.

Using GRINDER, the experiments are directly configurable regarding the location of fault injectors, for the placement of monitors, for selecting employed fault models and for logging of campaign data. The experiment and campaign configurations are stored in a MySQL-compatible *Database* (e.g., MariaDB). To transmit the experiment configurations from GRINDER to the target and experiment data back from the target to GRINDER, a communication channel between GRINDER and the target is used. The *TCPServer* provides a TCP-based communication interface to (a) handle incoming configuration requests from the target by fetching and sending the configuration of the next executable experiment from the database and (b) store

log data from the target for the currently executing experiment in the database.

The AUTOSAR target system that GRINDER is adapted for runs on a Freescale XKT564L evaluation board[2], which hosts a 32-bit dual core Power Architecture microcontroller. As the XKT564L target is not equipped with an Ethernet interface to directly interact with GRINDER's TCPServer, the board is connected to a host computer via its JTAG/Nexus hardware debugging interface. On the host computer, the Green Hills MULTI[3] *Debugger* is utilized by GRINDER to interact with the hardware, and a *Multiplexer* handles interactions of the AutosarAbstraction and the TCPServer with the target through the debugging interface and vice versa. On the AUTOSAR target, the generic and extensible C library *libGRINDER* implements a compatible communication interface for debugger-based message exchange. The library further provides pre-configured injector, detector, and logging logic for the use in interceptors, i.e., probes in the target system that can be used to inject faults or monitor the system's state (they are indicated by gray circles in Figure 3.3).

The AutosarAbstraction implements GRINDER's TargetAbstraction interface as follows.

- **start()** initializes the experiment environment by starting a new instance of the MULTI debugger, connecting to the debugger via TCP and establishing a connection between the debugger and the evaluation board using the debugger's *connect* command. A valid target configuration in MULTI is required for the connect command to succeed.

- **runExperiment()** prepares the target system by verifying that the correct binary is loaded and starts the execution of the target system. Furthermore, a *variable watch*[4] is used by the target to indicate a communication request, for example to retrieve configuration options or to store log information. As runExperiment() has full access to the debugger's features, additional functionality may be implemented if needed.

- **reset()** instructs the debugger to halt the system and set it to the initial state. Since the debugger is always able to reset the target system, this method works well to reset the entire system without further interaction.

- **stop()** terminates the experiment environment by disconnecting the target system and shutting down the MULTI debugger.

[2] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=XKT564L
[3] http://www.ghs.com/products/MULTI_IDE.html
[4] A hardware breakpoint that constantly inspects a watched variable during program execution for write access.

It is noteworthy, that the presented AutosarAbstraction is applicable for debugger-based interaction of the host computer and the AUTOSAR system, using Greenhills MULTI and the Nexus debug interface, as a direct TCP connection is unavailable. While both, the MULTI toolchain and the Nexus interface, are widely used, other AUTOSAR evaluation setups may feature slightly different tools or interfaces. For such scenarios, we estimate that the adaptation of the provided AutosarAbstraction to a different setup entails low overhead as only the communication channel needs to be adapted accordingly.

3.5.2 *Instrumenting AUTOSAR Systems for FI: What is special about AUTOSAR?*

Fault injection relies on mechanisms to access and modify the actual data- and control-flow within a system. These mechanisms are typically implemented as *interceptors* that are inserted in the system by instrumentation. Interceptors may implement arbitrary functionality, such as altering data (e.g., bit-flips, fuzzing, etc.), modifying the control flow (e.g., call other functions) or monitoring the system (e.g., for logging data).

As shown in Chapter 2, the instrumentation of AUTOSAR systems is complex due to the following factors:

- AUTOSAR systems are developed as models that highly abstract from the actual implementation, which comprises a mixture of tool-generated and hand-written code. As consequence, instrumentation at the model level cannot leverage implementation specifics (thereby limiting customizability), while instrumenting the implementation, i.e., mostly tool-generated code, is a tedious process if performed manually (thereby limiting usability).
- Given the degree of abstraction, elements of the model often have no singular representation in the implementation. Thus, the view on the system that the model provides is inherently different from the actual implementation. Moreover, as AUTOSAR promotes the integration of white-box and black-box components by various suppliers, who distribute their software either as source code or binary-only (to protect their intellectual property), the chosen instrumentation approach should work on source code and binary objects.

For the assessment of the functional safety mechanisms, the instrumentation of all AUTOSAR layers is beneficial, as it enables the injection of faults that manifest at the SW-C, RTE or BSW layers, and the flexible placement of monitors to observe system behavior and fault effects. An automated framework, which leverages development artifacts from the system model and the implementation to drive the

instrumentation process (such as the one presented in Chapter 2), can help with the instrumentation on the SW-C and RTE levels.

On the BSW level and specifically for the OS, a different approach has to be chosen, as no standardized model of the OS exists to augment and drive the instrumentation process. Our analysis of actual AUTOSAR OS code has shown that the situation is further complicated by the extensive use of macros throughout the OS for performance reasons. This directly impacts the applicability of static analysis to derive potential instrumentation locations, e.g., for interface injections [Voa98], as constructs that syntactically look like function calls can turn out to be macros that lack type safety and have no type declarations. Macros, in contrast to actual functions, do not have explicit signatures (prototypes), return types or parameter types.

Despite these impediments, a tool-supported instrumentation still has the advantage of the automated generation of interceptor code. Hence, we have chosen to extend the static analysis tool CIL (C Intermediate Language) [NMR+02] with a plug-in for the instrumentation of AUTOSAR OS source code, which we complement with manual instrumentation when needed. For binary level instrumentation, the approaches presented in [CLN+12; IKH+14] are conceivable.

We do emphasize that recurrent instrumentation can negatively impact the experiment efficiency especially when running experiments on actual embedded hardware due to the necessary time-consuming re-flash cycles. Our approach to avoid re-flashing, or at least to minimize its usage, is to instrument the system upfront with all interceptors that are potentially required for various test campaigns and to selectively enable or disable them at runtime in a pre-experiment configuration phase. This is achieved by assigning a unique identifier to each interceptor, whose state (on or off) can be configured on a per-experiment basis.

3.6 FAULT INJECTION CASE STUDY

Having outlined the FI framework and instrumentation approach, we now present our case study where we demonstrate the assessment of AUTOSAR's timing monitoring safety mechanisms in two scenarios using GRINDER. As mentioned earlier, the intent is to highlight the viability of the approach and its effectiveness in locating a bug in the timing monitoring implementation of a commercial AUTOSAR OS. For simplicity of illustration and communicating the insights, we focus on a small subset of the conducted FI experiments.

In the conducted experiments, we concentrate on assessing safety mechanisms specifically related to timing monitoring. This choice was motivated by the following factors.

- *Timing monitoring* is one of the newest safety mechanisms added in AUTOSAR and, except for a programmable timer interrupt,

the monitoring functionality is entirely software based. With the discussion on the limitation of classical FI to address timing issues (cf. Section 3.3), this constitutes a good target to detail how faults related to task timing can be accurately injected with software.

- *Memory protection* is based on COTS hardware that is widely applied in other domains. Except for project-specific misconfigurations, we did not see much potential for software-related dependability issues.
- *Logical supervision features*, such as aliveness and deadline monitoring, that are implemented by a watchdog are well-established and were already used in systems based on the AUTOSAR predecessor OSEK.
- *End-2-end protection* is library-based and does not necessarily require an AUTOSAR environment for testing, as was shown by Vedder et al. [VAV+14].

In summary, we considered the assessment of timing monitoring to offer a pertinent case study for FI applicability and of its effectiveness.

AUTOSAR's timing monitoring consists of four discrete mechanisms that, in general, can be used independently, but may require a combined use to attain guarantees on specific system timing characteristics.

- *Execution time monitoring* checks a task's execution time against a fixed time budget. When the budget is exceeded without the task finishing its execution, a timing error is detected.
- *Inter-arrival time monitoring* monitors a task's activation frequency within a statically configured time frame. When an activation threshold is exceeded, a timing error is detected.
- *Resource locking time monitoring* checks the locking time of resources by tasks against a fixed time budget. When the budget is exceeded without the task releasing the resource, a timing error is detected.
- *Interrupt locking time monitoring* checks the locking time of interrupts by a task against a fixed time budget. When the budget is exceeded without the task re-enabling interrupts, a timing error is detected.

For the case study, targeting correctness and robustness as the drivers, we illustrate the FI approach and its evaluation over two example scenarios (i.e., a simple case of execution time monitoring and a complex timing interaction scenario) to demonstrate its broad applicability.

Scenario 1: Task timing errors are provoked to (a) assess the *correctness* of the error detection and error mitigation of *execution time*

monitoring, i.e., whether the mechanism detects the timing errors and mitigates their effect, and (b) analyze *error propagation* within the system with and without timing monitoring enabled.

Scenario 2: The interaction between *execution time monitoring* and *resource locking time monitoring* is investigated. Both mechanisms share a common timer (embedded hardware usually has a limited number of timers), and the timer for execution time monitoring is reset and overwritten when a monitored resource is acquired. Arbitrary task timing and resource usage patterns are injected to (a) assess the *correctness* of the mechanisms, and (b) to assess their *robustness* by aiming to provoke situations in which the re-activation of the original timer fails.

The following sections detail the progression of the FI setup and its execution. We start with the detailed specification of the used fault models. Subsequently, the evaluation setup is presented in Section 3.6.2 and the results of the evaluation are discussed in Section 3.6.3.

3.6.1 *Deriving Fault Models for the Case Study*

In the following, we describe how we derived the fault models used for the case study. As the fault models provided by AUTOSAR and ISO 26262 are very abstract, the intent is to provide these examples as guideline for other FI-based assessments of AUTOSAR safety mechanisms.

Scenario 1 - Assessing Execution Time Monitoring

In the first scenario, we aim to trigger timing errors of arbitrary tasks by altering their control flow to either call a timed loop, thereby extending their runtime by a fixed offset (transient fault), or an infinite loop, thereby blocking execution completely (permanent fault). The trigger condition of the injection should be freely configurable in order to analyze the effects of error propagation throughout the system at workload-dependent times. The fault model for this scenario is specified as follows.

- *Fault type:* A loop that consumes a defined (possibly infinite) amount of CPU time to inject transient and permanent timing faults with high accuracy.
- *Fault location:* In the control flow of a monitored task, e.g., in its implementation (when source code is available) or its invocation.
- *Fault timing:* During different phases of the workload. The injection is triggered when a counter of the number of task invocations reaches a configurable threshold.

Scenario 2: Assessing Timing Monitor Interactions

For the second scenario, we manually reviewed the source code of the implementation of timing monitoring in a commercial AUTOSAR OS to identify potential robustness issues that we could further analyze with FI. We should highlight, that although having profound knowledge of AUTOSAR and the C programming language, this was our first encounter with AUTOSAR OS code. As such, our analysis was not influenced or guided by in-depth knowledge, and we chose the generic approach of identifying assumptions that were potentially made by the developers (OS experts could have used a more refined approach). We checked for assumptions regarding the outcome or return value of an operation, shared resource usage, potential time-of-check to time-of-use issues (i.e., race conditions), assumptions on variable initialization or state, and more, which resulted in eleven potential FI targets. In the following, we detail the FI target that we use in our case study and that uncovered a deficiency in the implementation of the timing monitoring safety mechanisms, which was subsequently acknowledged and fixed by the supplier.

In our code review, we had noticed that resource lock monitoring uses the same timer as execution time monitoring to detect resource lock errors accountable to the excess of a lock time budget. When a task acquires a monitored resource, the timer for execution time monitoring is reset and the remaining execution time budget is stored and compared to the lock budget of the resource. The smaller value (i.e., the shorter time frame) is then used as the new timeout value and the resource lock timer is activated. Whenever the resource lock is released by the task, execution time monitoring is re-activated. Although sharing a timer, or resources in general, is common in embedded systems, it also increases complexity due to synchronization issues.

To assess whether any resource usage and task timing patterns could potentially lead to a failure of the re-activation of the execution time monitor, we iterated over various timings for the resource lock time, execution time and the resource lock time budget. To emulate different time budget configuration efficiently, we directly inject in the monitoring mechanism's kernel data structure. This FI-based approach has two advantages over a conventional approach. Firstly, it enables the injection of *unexpected* budget configurations for the purpose of robustness testing, which is restricted by the configuration tool to sound budget settings. Secondly, it greatly accelerates the assessment, as the workload (i.e., task timing behavior) and the configuration of timing monitoring budgets can be directly adjusted between experiments. Normally, changing the configuration of timing monitoring on the target is a tedious and time-intensive process. It entails modifying the configuration in a GUI, generating kernel

source code, compiling the binary image and flashing it to the target. The same process applies to modifications of a task's timing behavior with additional adjustments needed for the actual implementation.

For the assessment of timing monitor interactions, we employ two fault models specified as follows. The first fault model is adapted from the previous scenario to flexibly induce different resource lock times during a task's execution. This is achieved by altering its control flow between the acquisition and release of a monitored resource.

- *Fault type:* A loop that consumes a defined (possibly infinite) amount of CPU time to emulate transient and permanent timing faults with high accuracy.
- *Fault location:* Between the acquisition and release of a monitored resource.
- *Fault timing:* Between the acquisition and release of a monitored resource. The injection is only triggered on the first acquisition of a resource.

The second fault model aims to modify the configured resource lock budget by injecting arbitrary budget values in the monitoring mechanism's kernel data structure as follows.

- *Fault type:* An arbitrary, potentially unsound resource lock budget. A series of lock budgets is generated by iterating over a fixed time interval with a configurable step size. Of this series, one budget is injected per experiment.
- *Fault location:* In the kernel data structure holding the monitored task's timing characteristics and budget configuration.
- *Fault timing:* Before or upon the resource allocation of the monitored task. The injection must have occurred before the budget configuration is evaluated by the monitoring mechanism.

3.6.2 *Evaluation Setup*

The case study example is a simple adaptive cruise control (ACC) system and depicted in Figure 3.4. The ACC comprises four software components that contain one or two runnables with periods of 10 ms and 40 ms each. Further, the SW-C *Environment* provides environmental stimuli to the ACC. It is noteworthy that although this case study example is purely hypothetical and does not represent any particular real design it is intended to represent plausible mixed-IP[5] and mixed-criticality integration scenarios relevant to the automobile industry.

[5] Mixed-IP systems integrate intellectual property (IP) by various suppliers.

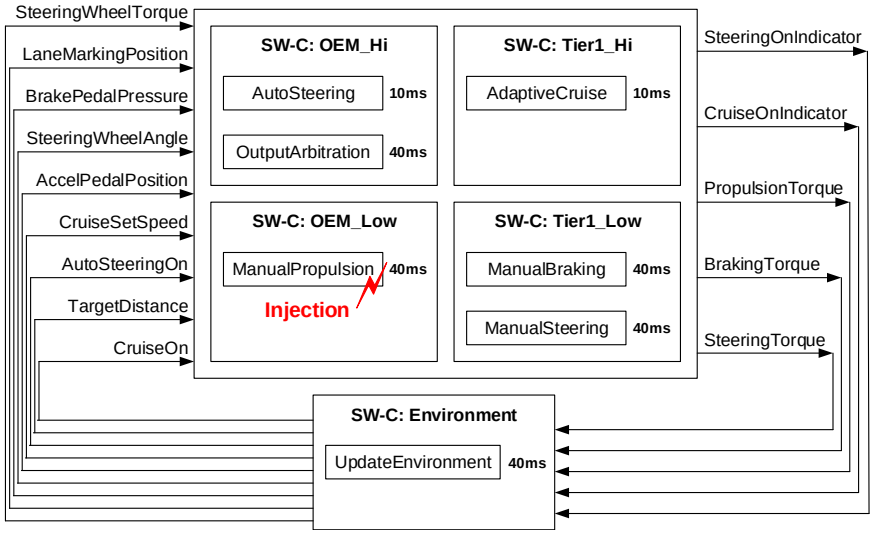


Figure 3.4: The adaptive cruise control (ACC) case study example.

Table 3.1: Task configuration of ACC case study example.

Task name	Priority	Runnable(s)
OEM_Hi_10ms	100	AutoSteering
Tier1_Hi_10ms	90	AdaptiveCruise
OEM_Hi_40ms	80	OutputArbitration
Environment_40ms	70	UpdateEnvironment
OEM_Low_40ms	60	ManualPropulsion
Tier1_Low_40ms	50	ManualBraking, ManualSteering

Table 3.1 lists the runnable to task assignment and the configuration of the six tasks of the system. The tasks are ordered from highest to lowest priority, which AUTOSAR schedules following a fixed-priority preemptive approach, i.e., when a tasks becomes ready that has a higher priority than the currently running task, the running task is preempted and the new task is executed.

In order to enable error reporting and a flexible reaction to different errors, AUTOSAR specifies the so-called *ProtectionHook* interface as part of its error handling process [AUT14d; AUT14g]. The ProtectionHook is invoked whenever one of the safety mechanisms detects an error. The detected *error type* is passed as parameter, based on which further analysis and mitigation steps may be initiated. The user-supplied return value of the ProtectionHook defines whether the OS performs further actions (e.g., task termination or ECU shutdown) or ignores the error. Throughout the case study, we use the information provided by the ProtectionHook to determine if and when an error was detected. Moreover, to flexibly enable or disable timing monitoring (e.g., to compare system behavior or error propagation),

we modify the return value of the ProtectionHook depending on the experiment configuration using an injector.

3.6.3 *Experimentation and Results*

In the following, we illustrate and discuss the results of our experiments on the basis of two timing monitoring assessment scenarios.

Scenario 1: Assessing Execution Time Monitoring

In the first scenario, we evaluate the *error detection* and *error mitigation* of execution time monitoring for timing errors caused by a *permanent* hang of task *OEM_Low_40ms*. In order to inject an infinite loop in the control flow of the task *OEM_Low_40ms*, we place an injector in the runnable *ManualPropulsion*. The injection is triggered, whenever the number of invocations of the runnable passes a configurable threshold.

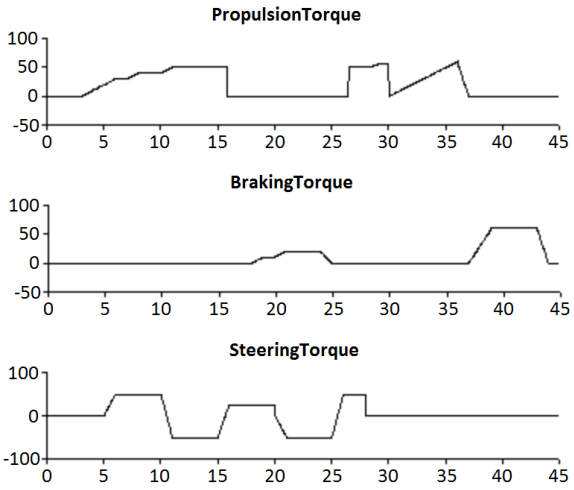
To enable the comprehensive observation of the system's reaction to fault injections and to analyze error propagation effects, we require access to signal traces within the system. For this reason, we have instrumented the SW-Cs *Environment* and *OEM_Hi* with 20 interceptors that are capable of logging all relevant signals within the system.

The scenario consists of three FI campaigns:

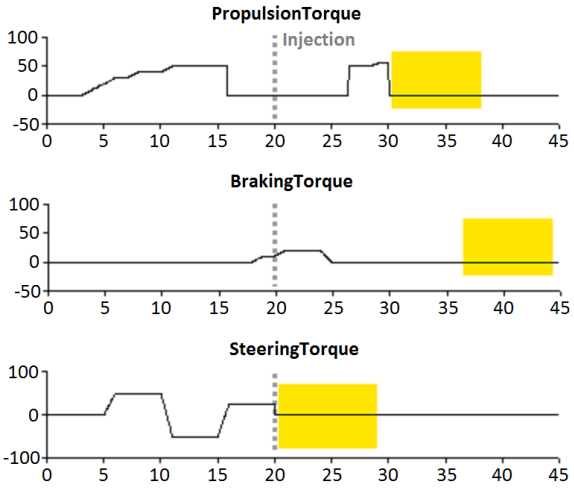
1. The fault-free golden run.
2. A series of experiments in which faults are injected at fixed, workload-dependent times and execution time monitoring is *disabled*.
3. The same series of experiments with execution time monitoring being *enabled*.

Using the flexible configuration approach of libGRINDER (cf. Section 3.5), no re-compilation or re-flashing was necessary for the different campaigns. Instead, we used the same binary image for all three campaigns and adjusted the configuration at runtime. Each FI experiment runs for 45 seconds, which is the time during which workload stimuli are provided by the *Environment*. In the following, we discuss one experiment from each campaign in detail to evaluate the effectiveness of *error mitigation* due to execution time monitoring.

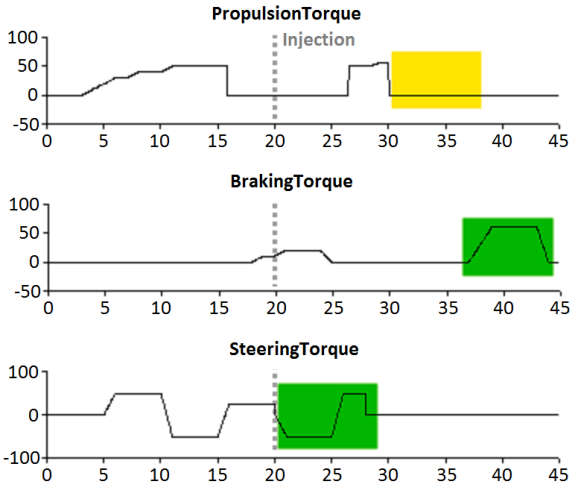
Figure 3.5 depicts the traces of the three signals *PropulsionTorque*, *BrakingTorque* and *SteeringTorque*. These three traces were chosen from the 20 available traces because the signals are affected by the fault injection either directly or indirectly due to error propagation. The fault-free golden run is shown in Figure 3.5a, whereas Figure 3.5b and Figure 3.5c depict the signals in the presence of a permanent



(a) Fault-free *golden run*.



(b) Monitoring disabled, error propagates.



(c) Monitoring enabled, propagation mitigated.

Figure 3.5: Scenario 1: Signal traces for a fault injection of an infinite loop (i.e., a *permanent* timing fault) in task *OEM_Low_40ms* at 20 seconds.

task hang injected at 20 s where execution time monitoring is either disabled or enabled.

For the scenario in Figure 3.5b where monitoring is disabled, all three signals have observable deviations from the golden run at different times in their signal. The change of *PropulsionTorque* (from 30 s to 37 s) is directly accountable to the FI, as *PropulsionTorque* is a composite signal[6] that also comprises input from the *ManualPropulsion* runnable that we injected into. The change of *BrakingTorque* (from 37 s to 44 s) and *SteeringTorque* (from 20 s to 28 s) is caused by error propagation from the task *OEM_Low_40ms*, which we injected into, to the task *Tier1_Low_40ms*. *Tier1_Low_40ms* comprises the runnables *ManualBraking* and *ManualSteering* whose outputs contribute to the composite signals *BrakingTorque* and *SteeringTorque*. As direct consequence of the propagation of the timing error, critical functionality provided by both runnables is lost.

In Figure 3.5c the same injection scenario is depicted with execution time monitoring enabled. The monitoring mechanism correctly detects the timing error of task *OEM_Low_40ms* that is caused by the injected fault. Despite the detection of the error, the signal of *PropulsionTorque* still deviates from the golden run, as the injected fault is permanent. Therefore, the monitoring mechanism repeatedly detects the persisting timing error and, in consequence, kills the erroneous task in each period after the injection. While the chosen strategy of killing the task is inadequate to mitigate the permanent timing fault in this case, it still successfully mitigates the propagation of the timing error to other tasks. The *BrakingTorque* and *SteeringTorque* signals, which were affected by error propagation without timing monitoring in the previous scenario, are now identical to the golden run.

In summary, execution time monitoring correctly detected errors due to injected timing fault in all of our experiments. Moreover, it successfully mitigated the effects of error propagation to lower priority tasks whenever monitoring was enabled, thus preventing the loss of critical functionality.

Scenario 2: Assessing Timing Monitor Interactions - Finding the Bug!

In our review of the timing monitoring implementation of a commercial AUTOSAR OS (cf. Section 3.6.1) we had noticed that execution time monitoring and resource lock time monitoring share a common timer to signal timing errors. If both monitoring mechanisms are enabled, the timer used by the execution time monitor is reset and overwritten by the resource lock monitor when a monitored task acquires a monitored resource. The new timer value is determined in the resource lock monitor's implementation by comparing the remaining execution time and resource lock budgets, and using the smaller

[6] A composite signal combines inputs from various sources.

value as the new budget. This ensures that budget violations, i.e., timing errors, are detected at the earliest point in time. In the error-free case, the resource is released after use and the cleanup routine of the resource lock monitor reinstates the execution time monitor. In the error case, the OS's error handling mechanisms are invoked.

In this assessment, we investigate the precedence relationship between both monitoring mechanisms with the intent to uncover any cases in which the re-activation of the original timer fails. Contrary to the previous scenario, our focus is not on the analysis of overall system behavior and error propagation effects, but on the *correctness* of the timing error detection and the *robustness* of the OS's timing monitoring mechanisms. Consequently, we do not require extensive logging mechanisms for signal traces in this scenario. Instead, we only instrument the *ProtectionHook* to provide logging of errors that are detected by the OS. Furthermore, we place injectors at two locations in the software stack. The first injector is placed between the resource acquisition and resource release in the runnable *ManualPropulsion* of task *OEM_Low_40ms* to inject arbitrary timing faults while holding a monitored resource. The second injector is placed in the resource lock monitor setup routine to inject arbitrary resource lock budget configurations. The aim of the budget injections is twofold: Firstly, we evaluate the robustness of the monitoring mechanism by injecting unsound budget values that are normally prohibited by the GUI-based configuration (e.g., resource lock budgets that are bigger than execution time budgets). Secondly, we inject sound budget values to change the configuration of the monitoring mechanism on-the-fly during runtime, thus avoiding the time-consuming steps of reconfiguration, recompilation and flashing of the binary.

This scenario consist of one campaign, in which we inject an infinite loop (i.e., a permanent timing fault) between the acquisition and release of a resource in the runnable *ManualPropulsion*. Per experiment, one value of a series of sound and unsound resource lock budget values is injected directly in the data structures of the monitoring mechanism when the resource is acquired and the monitor is initialized.

For all of our experiments, the timing monitoring mechanisms detected that a timing error had occurred and also the point in time of the error was detected correctly. After reviewing the log data, and to our surprise, the distribution of the reported error types (execution time vs. resource lock time) did not match the distribution expected from the injected budget values. As the series of injected budget values was generated using the execution time monitor budget as median value, we expected a fifty-fifty distribution of each error type, i.e., for those experiments where the resource lock budget was smaller than the execution time budget, we expected a resource lock

error, and for the opposite case an execution time error. Instead, all error types were logged as resource lock errors.

In order to verify that the observed mismatch of the distribution is not only accountable to the injection of unsound budget values, we analyzed the root cause for the mismatch further. As a result, we discovered that the type of timing error is misidentified whenever a timing error occurs while holding a locked resource and, at the same time, the remaining execution time budget is lower than the remaining resource lock budget. Whenever these constraints are met, errors accountable to execution time violations are reported as resource lock errors, affecting both, sound and unsound, configuration conditions.

This deficiency of the implementation is critical, as the handling of errors by the OS and the user (through means of the ProtectionHook) relies on the correctness of the supplied error type. A wrongfully identified error may therefore directly impact the error analysis and, in consequence, may lead to the execution of inappropriate mitigation actions. We reported the discovered issue to the vendor of the AUTOSAR OS implementation, who was able to reproduce it. The vendor acknowledged the issue as a bug and fixed it in subsequent releases of the OS.

Case Study Summary

In the case study, we have presented two scenarios for the FI-based assessment of AUTOSAR's timing monitoring mechanisms using the FI framework GRINDER. In the first scenario, we evaluated the correctness of the error detection and error mitigation of execution time monitoring for permanent timing faults. The monitor detected all injected timing faults correctly and was able to mitigate error propagation successfully, thus preserving critical functionality. In the second scenario, the correctness and robustness of execution time monitoring and resource lock monitoring was evaluated by injecting a permanent resource lock timing fault in combination with the injection of sound and unsound resource lock budgets. While the monitoring mechanisms correctly detected the presence of an error and its point in time, the source of the error was misidentified under certain conditions, which potentially affects error mitigation actions negatively. As the focus of our work was on providing the FI assessment mechanisms and guidance on their use, we only conducted around 200 experiments overall, which comprise the basis for this case study. To put this number into context, we should note that comprehensive FI studies sometimes comprise hundreds of thousands of experiments and that consequently the amount of experiments that we conducted can be considered very low. It is therefore even more impressive that, as a result of our FI experiments, we uncovered a bug in a commercial AUTOSAR OS implementation, which emphasizes and justifies

the use of FI as an effective method for the assessment of AUTOSAR's safety mechanisms.

3.7 CONCLUSION

Innovation in the automotive sector is mainly driven by software, which is leading to automotive software systems of massively increased complexity. To foster reusability, portability and interoperability of automotive software components, the AUTOSAR industry standard promotes a modular software architecture for automotive systems. At the same time, the ISO 26262 standard addresses safety considerations for automotive control systems, covering both hardware and software aspects, which has led to the addition of a safety concept to AUTOSAR comprising several safety mechanisms. While the ISO 26262 explicitly recommends fault injections (FI) to assess such mechanisms, it provides little guidance on how experiments should be performed and how suitable fault models can be identified.

In this chapter we described the process of implementing and applying FI for the validation of AUTOSAR's safety mechanisms, according to recommendations outlined by the ISO 26262 standard. To conduct the FI experiments, we have adapted the open source FI framework GRINDER to AUTOSAR. By making our implementation openly available, we provide a ready to use FI framework for AUTOSAR and hope to foster the development and use of FI for this target environment. To fill the gap between the ISO standard's requirement to apply FI and the absence of suitable fault models, we have provided a detailed discussion on how we derived fault models for our assessment, which targets a commercial implementation of AUTOSAR's timing monitoring safety mechanisms. The conducted experiments uncovered an actual bug in the interaction of two timing monitoring mechanisms that could lead to a misidentification of the source of a timing error and negatively impact the effectiveness of error mitigation. The bug was subsequently acknowledged and fixed by the supplier of the safety mechanism's implementation.

In summary our results demonstrate that (1) FI is an effective method to assess automotive suppliers' implementations of AUTOSAR safety mechanisms, (2) suitable fault models for these systems can be derived from their functional specification and their intended usage context, and (3) using these fault models actual deficiencies in the implementation can be identified with a modest amount of experiments.

Part IV

ENHANCEMENT

4

ENHANCING TIMING PROTECTION FOR MIXED-CRITICALITY SYSTEMS

For mixed-criticality automotive systems, the functional safety standard ISO 26262 stipulates *freedom from interference*, i.e., errors should not propagate from low to high criticality tasks. To prevent the propagation of timing errors, the automotive software standard AUTOSAR provides monitor-based timing protection, which detects and confines task timing errors. As current monitors are unaware of a criticality concept, the effective protection of a critical task requires to monitor *all* tasks that constitute a potential source of propagating errors, thereby causing overhead for worst-case execution time analysis, configuration and monitoring. Differing from the *indirect* protection of critical tasks facilitated by existing mechanisms, we propose a novel monitoring scheme that *directly* protects critical tasks from interference, by providing them with execution time guarantees. The monitor is implemented as an enhancement to the existing monitoring infrastructure of a widely used commerical AUTOSAR OS and meant to augment existing mechanisms. Overall, our approach provides efficient low-overhead interference protection, while also adding transient timing error ride-through capabilities. The content of this chapter is based on a conference paper presented at ISORC 2015 [PWS+15a].

4.1 INDIRECT VS. DIRECT TIMING PROTECTION

The automotive industry is encountering growing interest in the development and integration of mixed-criticality systems [BD14], i.e., systems containing components with varying degrees of assurance on timing and safety. This trend arises from the increasing multitude and complexity of innovative (often software based) driver assistance features while dealing with resource constraints of limited space, energy capacity and distribution, weight and, fundamentally, costs. Essentially, the integration of the historically segregated automotive systems, which have been conservatively designed following a *one function per electronic control unit (ECU)* approach, offers cost saving potential for hardware and wiring, as it entailed up to 100 federated ECUs distributed in modern luxury cars [Cha09].

Similar to the „gold standard for partitioning“ in integrated modular avionics [Rus99], the functional safety standard for road vehicles ISO 26262 [Int11] permits the integration of elements with differing criticality, as long as partitioning mechanisms can verifiably provide

freedom from interference in both the spatial and temporal domains, i.e., regarding memory accesses and timing behavior. In automotive systems, partitioning is usually supported by hardware [WEK10; ZBS+14], the operating system (OS) [AUT14i; BFT09], or a combination of both (e.g. virtualization) [RM14].

The established AUTOSAR standard [AUT14a] for automotive software addresses freedom from interference through a set of safety mechanisms in its Technical Safety Concept [AUT14i] that are provided as services by the AUTOSAR OS. To support partitioning in the temporal domain, the OS provides monitoring of task execution time budgets, activation frequencies, and resource lock times. The violation of a monitor policy constitutes a timing error, whose propagation is prevented by stopping the responsible task and freeing locked resources.

AUTOSAR schedules tasks based on a fixed-priority preemptive scheme [ABD+95], in which the processor executes the highest priority task among the tasks ready for execution. Without timing protection, timing errors, i.e., any fault that leads to a violation of the specified worst-case execution time (WCET) of a task, may propagate from high priority to low priority tasks. Due to rate monotonic priority assignment [LL73; LSD89], the priority and criticality of a task are usually unaligned, i.e., the most critical task is not necessarily assigned the highest priority. Thus, criticality inversion [NLR09] may occur through timing error propagation. AUTOSAR's timing protection mechanisms, specifically execution time monitoring, aim at detecting and confining timing errors to the task where they originate. To protect critical tasks from the effects of timing errors arising in less critical tasks, all such less critical tasks require execution time monitoring.

This approach has a number of undesired implications in mixed-criticality systems:

- Critical tasks are protected *indirectly* by confining timing errors in less critical tasks. To protect a given critical task, *all* less critical tasks must be individually and correctly monitored, which is an error-prone process.
- Monitoring causes configuration and run-time overhead (cf. Section 4.5), which can be reduced by focusing on critical tasks only.
- Worst-case execution times of uncritical or less critical tasks are often over-approximated [FFR12; Ves07], which, when enforced via monitors, may impact overall system utilization negatively.

Contributions

On this background, we propose a novel, criticality-aware AUTOSAR run-time monitoring scheme that guarantees critical tasks a configurable execution time budget to *directly* protect them from timing errors of other tasks. The budget guarantee is enforced by monitoring a task's preemption budget (PB), which determines how long a task may be preempted without compromising its timely execution. By focusing only on critical tasks, we reduce the overhead for run-time monitoring and WCET analysis of non-critical tasks. Further, any unused computation time of critical tasks may be spent by erroneous non-critical tasks to eventually finish (transient ride-through). To put this contribution in context, we highlight that currently AUTOSAR lacks support for mixed criticality scenarios by its monitoring mechanisms. Our work helps to provide this support.

This chapter is structured as follows. We review the work related to our problem scope in Section 4.2. In Section 4.3, we provide background on AUTOSAR that is essential in understanding the proposed preemption budget monitor, which is presented in Section 4.4. In Section 4.5, we evaluate the efficiency and overhead of our approach in a case study, and provide our conclusion in Section 4.6.

4.2 RELATED WORK

Following a standalone exposition of the body of work relevant to our problem scope, we summarize its viability on addressing timing error propagation.

Related to our approach is the work by de Niz et al. [NLR09], who identify a criticality inversion problem, for which less critical tasks may block more critical tasks in fixed-priority preemptive systems, if criticality and priority are not aligned. The authors first try to address criticality inversion by the criticality driven priority assignment scheme *Criticality As Priority Assignment* (CAPA), which has the drawback that more critical tasks with long periods may block the execution of less critical tasks with short periods, resulting in deadline misses. As an improvement, they introduce zero-slack scheduling, a scheme that is based on a dual mode task model (normal and overload) with the aim to maximize resource utilization, while providing protection from criticality inversion.

The *period transformation* approach proposed by Sha et al. [SLR86] slices critical tasks with longer periods than less critical tasks in sections, such that each of those sections has a shorter period than any less critical task. Scheduling such a sliced set with a rate monotonic approach will result in task criticalities and priorities being aligned. However, this comes with the drawback of increased system manage-

ment overhead, additional complexity of sharing data across slices, additional development effort for task slicing, and the basic requirement that tasks must be sliceable.

Ficek et al. [FFR12; FSF+13] developed a design workflow that provides guidance on how to effectively apply the previously discussed CAPA and period transformation approaches, together with AUTOSAR's execution time monitoring facilities to the overall system design, in order to ensure freedom from interference for critical tasks.

Baruah et al. [BBD11; BBD13] propose an extension to the fixed-priority preemptive scheduling approach, in which a system executes either in LO-criticality (normal) or HI-criticality mode. If any task violates its assigned execution time budget, a switch to the HI-criticality mode occurs, and task priorities are re-assigned in such a way that a set of predefined critical tasks remains schedulable.

In summary, the related work on mixed criticality systems (a comprehensive review is given by Burns and Davis [BD14]) focuses mostly on scheduling algorithms and schedulability analysis and not on timing errors, or how to prevent their propagation at run-time by monitoring. Additionally, these approaches do not directly conform to the AUTOSAR model, thus limiting their usage as such. The work of Ficek et al. [FFR12; FSF+13] is an exception in this respect. Contrary to their work, the approach proposed in this chapter does not require a (re-)design of the system to provide execution time guarantees and freedom from interference to critical tasks.

4.3 AUTOSAR SYSTEM MODEL

This section provides the reader with a background on AUTOSAR's scheduling model, task model, and timing protection that are essential in understanding the problem scope and the solution of preemption budget monitoring proposed in Section 4.4. The provided information focuses on timing aspects and is mostly complementary to Section 2.2 and Section 3.2.

The AUTOSAR OS [AUT14g] is a statically configured multitasking OS where all system objects, such as tasks and resources, are allocated at build time. The OS schedules tasks in a fixed-priority preemptive manner [ABD+95] executing the highest priority task among all ready tasks. For the assignment of task priorities [MNS+10], a rate monotonic scheme [LL73; LSD89] is commonly used, where the period or deadline of a task determines its priority (i.e., the shorter the period/deadline, the higher the priority). Task priorities are generally static, with the exception of the priority ceiling protocol [SRL90], which is used to avoid priority inversion when resources are shared across tasks. In such situations, the priority of tasks that hold a shared resource is temporarily raised to the priority ceiling of the resource.

We define a task τ_i as a tuple

$$\tau_i = (P_i, C_i, T_i, D_i, \zeta_i)$$

where:

- P_i is the static priority,
- C_i is the worst-case execution time (WCET),
- T_i is the period,
- D_i is the deadline, and
- ζ_i is the criticality of the task.

In our model, higher values of P_i and ζ_i indicate higher priority and criticality, respectively. To achieve a clearer presentation, we assume that $D_i = T_i$ throughout this chapter, unless stated otherwise. The WCET C_i constitutes the uninterrupted, maximum possible execution time of a task, and is either identified analytically (via static code analysis), experimentally (via tracing/measurement) or through simulation [KP05; GE07]. Depending on the assurance that the employed method provides, a buffer value is usually added to the WCET to compensate for inaccuracies. Thorough WCET analysis is essential in determining the system schedule and conducting schedulability analysis, which proves whether the schedule meets the overall timing requirements and ensures that all tasks meet their deadline under *error-free* conditions.

4.3.1 *Timing Error Propagation*

In the presence of a timing error, which we define as a task τ_i exceeding its WCET C_i, the schedule's underlying assumptions are invalidated. Undetected timing errors may impact the timing of fault-free tasks through *error propagation*, and lead to failures in the form of deadline violation (not finishing the execution until D_i) of fault-free tasks. To illustrate such a scenario, we assume a system with tasks τ_A, τ_B and τ_C, as shown in Table 4.1.

Table 4.1: Timing properties of the example system.

Task τ_i	WCET C_i	Deadline D_i	Priority P_i
A	2 ms	7 ms	3
B	2 ms	7 ms	2
C	2 ms	7 ms	1

Figure 4.1 depicts the task timing. During the first period from 0 ms to 7 ms, all tasks are error-free and finish before their deadline.

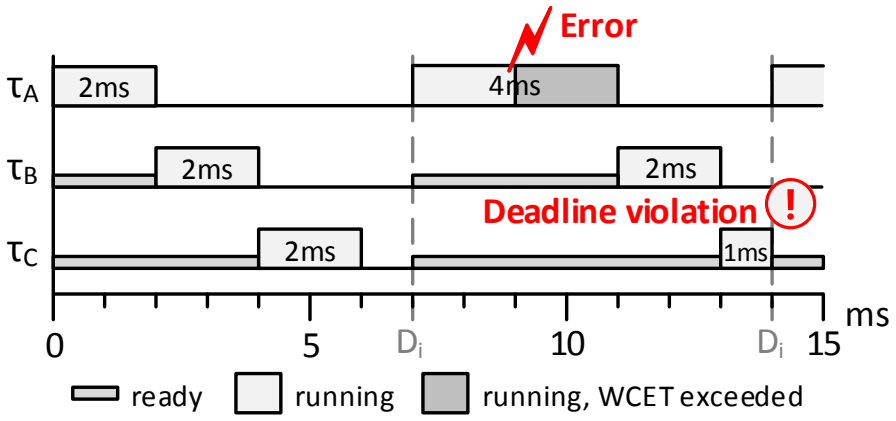


Figure 4.1: Deadline violation of τ_C due to a propagated timing error.

In the second period from 7 ms to 14 ms, τ_A is subject to a timing error at 9 ms (indicated by the red bolt) that prolongs its execution time by 2 ms. The error of τ_A propagates and delays the execution of τ_B and τ_C by 2 ms. When τ_C starts to execute at 13 ms, the remaining execution time does not suffice to finish until its deadline at 14 ms. The consequence is a deadline violation and timing failure of τ_C.

4.3.2 *AUTOSAR Timing Protection*

The propagation of timing errors from one task to another can be detected and prevented by timing protection mechanisms (TPMs) that monitor task run-time behavior. AUTOSAR specifies and implements the following TPMs as OS services [AUT14g; AUT14i], which are selectively enabled on a per-task basis.

- **Execution time monitoring** monitors a task's execution time and compares it to a budget. When the budget is exhausted without the task having finished, a timing error is detected.
- **Inter-arrival time monitoring** monitors a task's activation frequency within a statically configured time frame. When an activation limit is exceeded, a timing error is detected.
- **Locking time monitoring** limits the blocking time of tasks imputable to priority ceiling [SRL90] or disabling interrupts. For each resource and each task, a lock time budget can be specified, and locking a resource for longer than its budget for that task constitutes a timing error.

Upon detection of a timing error, it can be locally confined by killing the task or its task group (OS application). Killing a task results in an abortion and failure of the task.

AUTOSAR detects timing errors accountable to WCET violations by execution time monitoring (ETM). As AUTOSAR's monitoring

mechanisms currently lack support for mixed-criticality scenarios, ETM can only *indirectly* protect critical tasks by individually monitoring all tasks from which errors could potentially propagate, which is an error-prone process. In addition, ETM is inefficient compared to a criticality-aware monitoring scheme in mixed-criticality scenarios with few critical and many non-critical tasks, due to the overhead that monitoring the non-critical tasks entails. Recalling our previous example (Table 4.1), we assume that task τ_C is critical, while tasks τ_A and τ_B are non-critical. To guarantee freedom from interference to τ_C, both τ_A and τ_B require monitoring using ETM, while a *direct* criticality-aware monitor would only require monitoring τ_C. In Section 4.5, we compare the run-time overhead of ETM and a novel criticality-aware monitoring scheme that we propose in the next section.

4.4 PREEMPTION BUDGET MONITORING

As augmentation to the existing monitoring infrastructure of AUTOSAR, we propose a monitoring scheme that is specifically suited to mixed-criticality systems. It is based on the idea that instead of monitoring the timing behavior of all non-critical tasks that constitute potential sources of timing errors, it would be more efficient to only monitor critical tasks with the aim of providing a guaranteed execution time budget to them. Differing from the *indirect* protection of critical tasks facilitated by existing mechanisms, this approach *directly* protects critical tasks from interference. In consequence of this paradigm shift, non-critical tasks are eligible for transient error ride-through (see Section 4.4.2).

The execution time guarantee is provided by monitoring the preemption time of critical tasks, i.e., the time spent in the *ready* state waiting for execution. If the preemption time exceeds a threshold value, which we term *preemption budget* (PB), the immediate start of the monitored task is enforced by re-queueing it with the highest priority in order to prevent interference by other tasks. For systems with only one critical task τ_i, the task's PB_i can intuitively be defined as $D_i - C_i$, as the task needs to start its *uninterrupted* execution C_i time units before its deadline D_i.

For systems with several critical tasks, determining PB_i is more complex, as potential preemptions through other PB-monitored tasks with a higher *precedence* (explained below) need to be factored in. The problem can be thought of as *response time analysis* [ABR+93], but with an inverted time line. We therefore make an argument along the lines of [ABR+93] and define the response time R_i as the sum of the WCET C_i and the total worst-case interference I_i of other critical tasks with higher precedence on τ_i through preemption.

$$R_i = C_i + I_i \tag{4.1}$$

The PB_i of a task τ_i then follows as

$$PB_i = D_i - R_i \tag{4.2}$$

We argue that the interference on task τ_i from a task τ_j through preemption is nC_j, where n denotes how often τ_j executes within R_i, or in other words, how often its period T_j fits in R_i. For non-integer values of $n = R_i/T_j$, n has to be rounded up by the ceiling function, to account for the fact that τ_j will always preempt τ_i for its full execution time C_j (due to fixed-priority preemptive scheduling). The worst-case interference from a task τ_j on task τ_i is therefore given by

$$\left\lceil \frac{R_i}{T_j} \right\rceil * C_j$$

Consequently, the total interference I_i of other tasks on τ_i follows as

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j \tag{4.3}$$

where $hp(i)$ is the set of tasks that might preempt τ_i due to higher monitor precedence. The precedence between PB-monitored tasks is defined as follows. The higher a task's criticality, the higher its precedence. For tasks of equal criticality, their precedence is defined in alignment to their priority.

Combining eq. (4.1) and eq. (4.3), the unknown term R_i appears on both sides of the equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j$$

which can be iteratively solved as follows.

Let R_i^n be the nth approximation to R_i.

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil * C_j \tag{4.4}$$

Starting with $R_i^0 = C_i$, the iteration terminates when $R_i^{n+1} = R_i^n$. The iteration is guaranteed to converge if the processor utilization is $\leqslant 100\,\%$ [ABR+93].

In Table 4.2 we provide an example for the calculation of the PB. The example system consists of three tasks: τ_B is of high criticality, τ_C is of medium criticality, and τ_A is non-critical.

The PBs are computed according to the precedence relations defined above, i.e., from the highest critical task to the lowest. Therefore, we start the computation of the PB for task τ_B. As task τ_B is the highest critical task, the set $hp(B)$ is empty, and using eq. (4.4) we determine $R_B = R_B^0 = C_B = 2\,\text{ms}$. Using eq. (4.2) it follows

Table 4.2: Assigning preemption budgets in a mixed-criticality example.

Task τ_i	WCET C_i	Period T_i	Prio. P_i	Crit. ζ_i	PB_i
A	2 ms	6 ms	2	0	n.a.
B	2 ms	8 ms	1	2	6 ms
C	3 ms	12 ms	0	1	7 ms

that $PB_B = 8\,ms - 2\,ms = 6\,ms$. The next critical task is τ_C, for which we have to factor in that it might be preempted by the PB monitor of task τ_B due to its higher precedence. Therefore, the $R_C = R_C^1 = C_C + \left\lceil \frac{C_C}{T_B} \right\rceil * C_B = 3\,ms + 2\,ms = 5\,ms$ and $PB_C = 12\,ms - 5\,ms = 7\,ms$.

In the presence of a timing error, the PB monitor would force the execution of τ_B at 6 ms after its activation, providing τ_B a guaranteed execution time of $C_B = 2\,ms$. For τ_C, the PB monitor would force its execution at 7 ms, providing τ_C a guaranteed execution time of $C_C = 3\,ms$, and accounting for a possible preemption by τ_B for up to $C_B = 2\,ms$.

4.4.1 *Integration with AUTOSAR Task State Model*

We integrate the PB monitor into the AUTOSAR task state model as depicted in Figure 4.2.

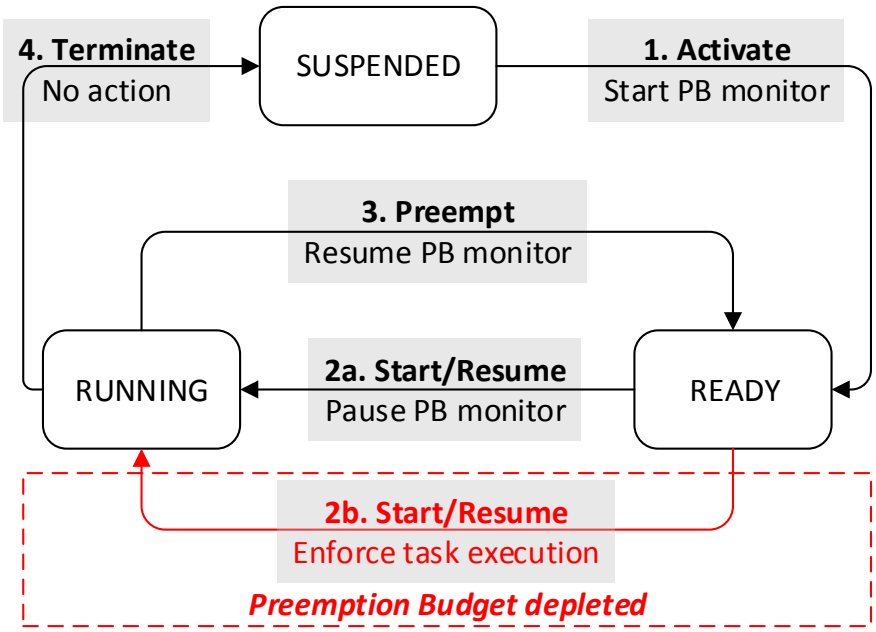


Figure 4.2: Task state transitions and corresponding PB monitor actions.

After system start-up, tasks are in the *suspended* state. Once a task is activated, it enters the *ready* state and the PB monitor is started

(*1. Activate*). As long as the task is in the *ready* state, its PB is consumed. In the error-free case, the task will eventually be dispatched (*2a. Start/Resume*) and enter the *running* state, from which it will either be preempted (*3. Preempt*) and be resumed later, or it will terminate after finishing its execution (*4. Terminate*). If the task leaves the *ready* state, the monitor will be paused. It will be resumed once the task re-enters the *ready* state.

In the erroneous case (*2b. Start/Resume*), the task will deplete its PB and a timer interrupt will trigger. To guarantee the critical task its WCET as execution time, it is either immediately started or enqueued with the highest priority, depending on whether higher precedence tasks have their PB depleted as well. The case study in Section 4.5 provides further details on the monitor's implementation and its performance in a transient and a permanent timing error scenario.

4.4.2 *Transient Error Ride-through*

A transient error is a temporary error that disappears after a limited amount of time. For example, a task with a nominal WCET of 2 ms may exceed its WCET by 1 ms and run for 3 ms before it finishes. As long as the task finishes before its deadline, it has not failed – the task made a *transient error ride-through*. For tasks that are protected using AUTOSAR's ETM, transient error ride-throughs are infeasible, as the monitor strictly enforces the configured execution time budget. Consequently, the task is forced to fail, although it could eventually have finished, depending on the overall CPU utilization and timing constraints.

Contrary to ETM, PB monitoring allows for transient error ride-throughs. We again consider our example system as specified in Table 4.2, but assume that only task τ_C is critical. For this modified scenario $PB_C = 12\,ms - 3\,ms = 9\,ms$, which is monitored. As shown in Figure 4.3, task τ_B is subject to a timing error at 10 ms (indicated by the red bolt) and continues to execute.

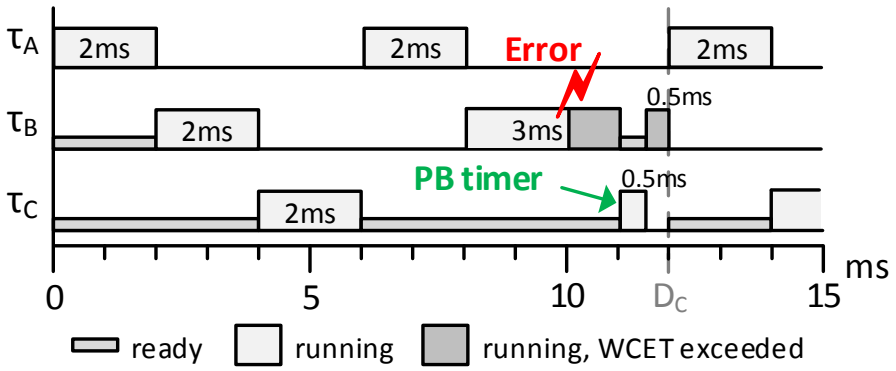


Figure 4.3: Example of a transient error ride-through of task τ_B.

At 11 ms, the PB of task τ_C is exhausted (remember that the PB is only consumed in the ready state) and the monitor enforces τ_C to execute. τ_C finishes its execution after 0.5 ms and has therefore not consumed all of its WCET. The remaining 0.5 ms are used by τ_B to eventually finish its execution at 12 ms, which is before its deadline at 16 ms. Accordingly, τ_B performed a transient error ride-through.

4.4.3 *Applicability to Multi-core Systems*

Although much research has gone into multi-core mixed criticality scheduling (e.g. [BCL+14]), AUTOSAR uses an approach based on static partitioning [AUT14b; MB09], in which a static task set is assigned to each partition/core. A possible explanation is provided by Reinhardt and Morgan [RM14], as „automotive software development has yet to exploit parallel processing in an efficient manner because legacy code, designed to run on single core systems, is difficult to adapt to run in parallel on multi-core systems." So, although we consider a single-core environment throughout this chapter, the given analysis should be equally applicable to multi-core AUTOSAR systems with a static task partitioning.

4.4.4 *Limitations and Possible Solutions*

The partitioning of a system in different criticality levels using PB monitoring is subject to a set of constraints in order to avoid effects similar to those of the CAPA approach discussed in Section 4.2. To illustrate the problem, we assume an example system with three tasks of different criticality, as shown in Table 4.3.

Table 4.3: Example of conflicting task constraints in a three criticality system.

Task τ_i	WCET C_i	Period T_i	Prio. P_i	Crit. ζ_i
A	2 ms	6 ms	2	0
B	2 ms	8 ms	1	1
C	10 ms	30 ms	0	2

According to the task precedence relationship, we first calculate the PB of τ_C and then of τ_B. It follows that $PB_C = 30\,ms - 10\,ms = 20\,ms$. For τ_B, we calculate $PB_B = 8\,ms - 2\,ms - 10\,ms = -4\,ms$. As PB_B is negative, assigning a PB to τ_B is infeasible. Consequently, no execution time guarantee can be given to τ_B. The explanation is straight-forward: as τ_C has a higher monitor precedence, τ_C could preempt τ_B whenever its PB_C is depleted. As τ_C has a WCET C_C of 10 ms, τ_B can impossibly meet its deadline $D_B = T_B = 8\,ms$ in that case.

The described effect cannot occur for dual criticality systems (non-critical and critical), as the set of critical tasks simply represents an ordered subset of the overall system and remains schedulable, if the overall system was schedulable. For systems with more than two criticalities, the effect does not occur as long as task criticalities are aligned with task priorities. For all other cases, we propose the following solutions:

- Invert the precedence of conflicting tasks (in our example τ_B and τ_C), so that the task with the shorter period τ_B can preempt τ_C. To continually provide freedom from interference to the more critical task τ_C in such a scenario, the execution time of τ_B would require additional monitoring with ETM.
- Re-arrange task priorities by period transformation to align criticalities and priorities (similar to the approach proposed by Ficek in [FSF+13]).

4.5 CASE STUDY

In this section, we evaluate our implementation of preemption budget monitoring (PBM) and compare it in terms of memory and run-time overhead to execution time monitoring (ETM). Further, we assess the effectiveness of PBM in direct comparison to ETM under the following timing error scenarios:

- **Transient:** A task exceeds its worst-case execution time to eventually finish its execution before its deadline.
- **Permanent:** A task is stuck, for example in a livelock or deadlock, and is therefore unable to finish its execution.

Our test system is a simplified adaptive cruise control (ACC) that consists of seven tasks, as shown in Table 4.4. We assume that the deadline of each task matches the task's respective period, and that τ_6 is the only critical task in our system.

We have implemented the test system for the XKT564L evaluation board by Freescale [Fre] using a widely used, commercial AUTOSAR tool suite for system integration. The XKT564L hosts a 32-bit dual core Power Architecture microcontroller unit (MCU) with 1 MiB of flash memory and 128 KiB of RAM (both ECC). The MCU is specifically developed for safety-critical applications and both cores are operated in lock-step mode to detect hardware run-time errors.

4.5.1 *PBM - Implementation Details*

Our PBM implementation is an extension to the monitoring infrastructure of a widely used, commercial AUTOSAR OS. We closely re-

Table 4.4: Task configuration of ACC case study.

Task	WCET C_i	Period T_i	Priority P_i	Critical
τ_1	30 µs	250 µs	7	-
τ_2	50 µs	250 µs	6	-
τ_3	145 µs	500 µs	5	-
τ_4	15 µs	500 µs	4	-
τ_5	20 µs	500 µs	3	-
τ_6	15 µs	1,000 µs	2	✓
τ_7	20 µs	1,000 µs	1	-

semble the structure of the existing monitoring mechanisms to support a seamless integration of our approach. Following the task state diagram (cf. Figure 4.2, Section 4.4), we added monitor hooks to the kernel's *enqueue* and *dispatch* functions, in order to handle task activation, start/resume and preemption in the monitor. Further, we implemented a handler for PB depletion (cf. Figure 4.2, transition 2b) to enforce the execution of critical tasks if necessary.

Table 4.5: Static overhead of PBM.

	Baseline	with PBM	Overhead
Kernel SLOC	25,736	25,985	+159 (0.6 %)
Flash ROM	43,903	44,319	+416 (0.9 %)
RAM	47,952	47,980	+28 (0.06 %)

In Table 4.5 we review the static overhead of PBM on the source code and binary levels. We utilized the tool SLOCCount [Whe] to determine the source lines of code (SLOC) of the baseline kernel, and compare it to the version with PBM as a measure for the implementation complexity of PBM. Overall, PBM increases the SLOC of the AUTOSAR OS kernel by 159 lines, which is an increase of 0.6 %.

The static flash ROM and RAM consumption was obtained using the tool *objdump* from the GNU Binutils toolsuite [GNU]. PBM consumes an additional 416 bytes (0.9 %) of flash ROM, and 28 bytes (0.06 %) of RAM.

4.5.2 *Timing Error Scenarios*

We evaluate the efficiency of PBM and compare it to ETM in a transient and a permanent timing error scenario. The errors are injected at run-time using a software-implemented fault injection (SWIFI) tool prototype for AUTOSAR that extends previous work [PWM+12]. For both error scenarios, the injection location is task τ_5 and the trigger

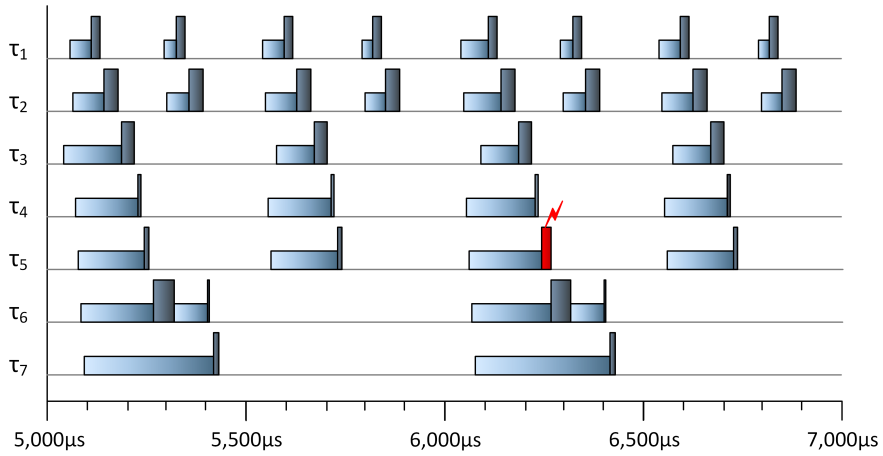
condition is set to the third execution of the task after system start (activated around 6,100 µs).

The three graphs in Figure 4.4 depict the transient error scenario for ETM and PBM. The graphs cover two full periods of the tasks with the longest period in the system (τ_6, τ_7). In the first period (5,000 µs to 6,000 µs), the error-free timing is shown. In the second period (6,000 µs to 7,000 µs), a transient timing error that lasts 100 µs is injected at 6,250 µs in τ_5. Figure 4.4a shows how ETM detects the error after τ_5 has consumed its WCET. As ETM strictly enforces the configured budget, τ_5 gets killed by the OS and thus fails. Figure 4.4b shows the same scenario for PBM. As the preemption budget of τ_6 is not consumed until 6,900 µs, the PBM does not interfere with τ_5, which eventually finishes its execution successfully. Figure 4.4c depicts a slightly different scenario with two error injections, in order to highlight the interaction between PBM and the erroneous task. The first injection at 6,250 µs serves the sole purpose of delaying the execution of τ_6. For the second injection at 6,750 µs, we observe that PBM gets activated around 6,900 µs and preempts τ_5. After executing for less than its WCET, τ_6 finishes and τ_5 uses the remaining budget from τ_6 to successfully finish. In all three scenarios, ETM and PBM effectively prevented task failures of the critical task τ_6, while the latter two scenarios demonstrate the transient ride-through capabilities of PBM.

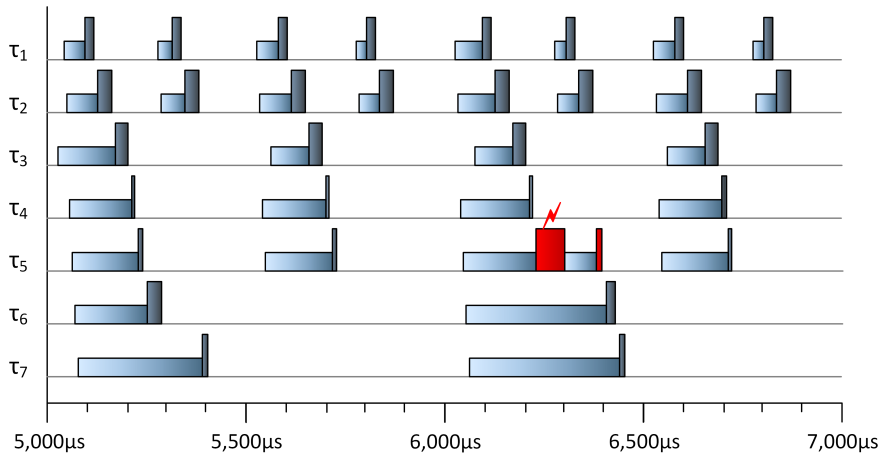
The two graphs in Figure 4.5 illustrate the permanent error scenario for ETM and PBM. The graphs cover four full periods of the task with the longest period in the system to better observe the effect of the permanent error. In the first period (5,000 µs to 6,000 µs), the error-free timing is shown. After the first period, a permanent timing error is injected in τ_5 at 6,250 µs. Figure 4.5a shows how ETM repeatedly detects the error after τ_5 has consumed its WCET and kills the task. Figure 4.5b shows the same scenario for PBM, which repeatedly detects an imminent error propagation from task τ_5 to τ_6 and prevents it by assigning τ_6 its guaranteed execution time budget. Both monitors, ETM and PBM are effective in preventing task failures of the critical task τ_6 that are accountable to timing error propagation. At the same time, ETM and PBM are incapable of mitigating the failure of τ_5 that is evoked by its permanent error. To address such error scenarios, we recommend to additionally employ a monitor that provides complementary mitigation strategies, such as an external watchdog.

4.5.3 *Comparison of run-time overhead*

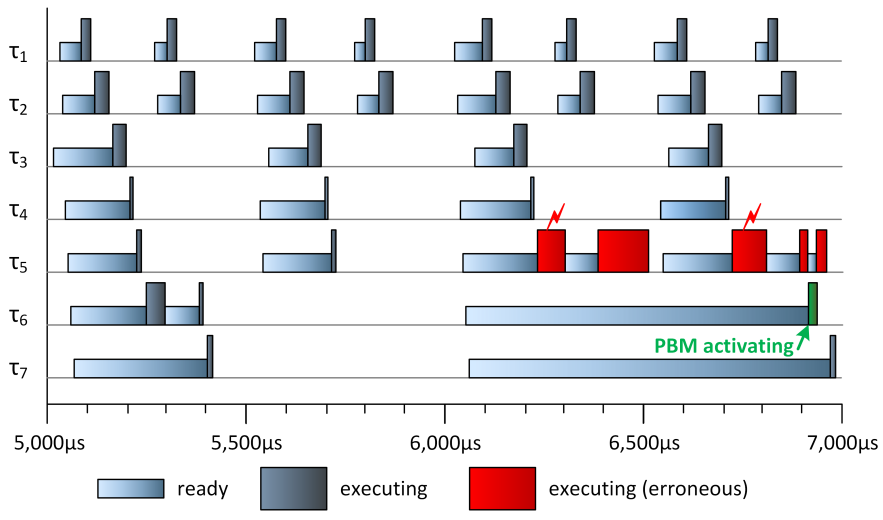
Monitoring the timing behavior of tasks entails run-time overhead whenever the monitor is invoked. In general, ETM is invoked at task start, preemption, resume, and termination. Correspondingly, PBM



(a) ETM: Prevents transient ride-through

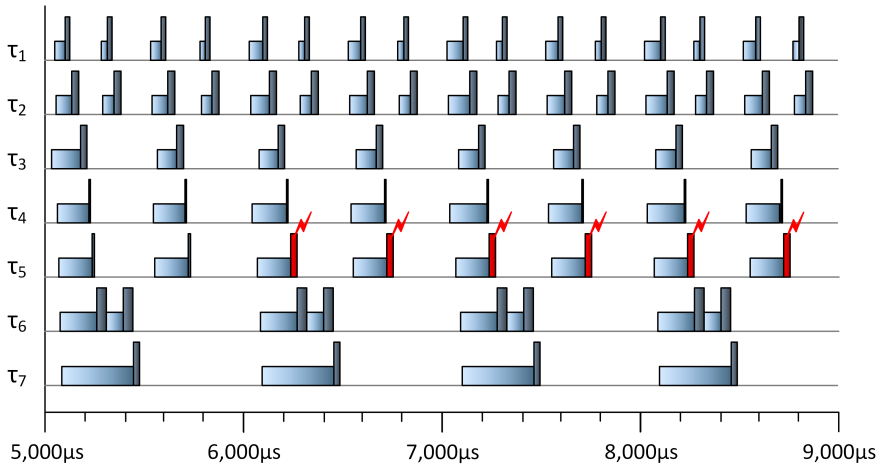


(b) PBM: Transient ride-through (case I)

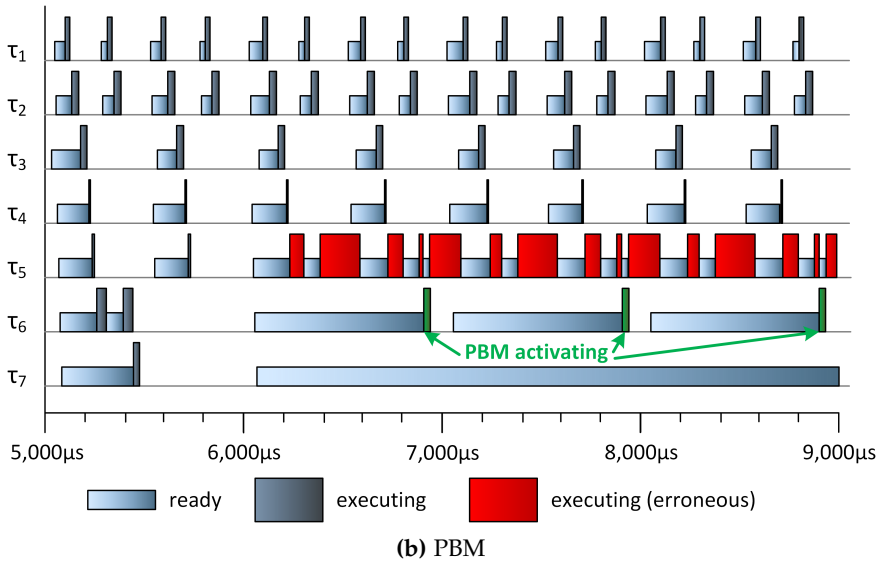


(c) PBM: Transient ride-through (case II)

Figure 4.4: Transient timing error scenario.



(a) ETM



(b) PBM

Figure 4.5: Permanent timing error scenario.

is invoked at task activation, start, preemption, and resume. As our case study only has few preemption/resume events, we focus our evaluation on activation, start and termination. It can be expected that the overall overhead grows linearly for ETM and PBM alike for systems with many preemptions, due to structural similarity of the monitors.

Our measurements over 500 activation, 500 start and 500 termination events demonstrate a very constant run-time behavior. For both monitors, the monitor start (at each task activation for PBM and each task start for ETM) consumes 2 µs on average, while the monitor stop routine (at each task start for PBM and each task termination for ETM) consumes 2.2 µs on average. Therefore, the monitor overhead for one period of a task's execution is fixed at 4.2 µs for ETM and PBM alike.

Table 4.6: Monitoring overhead for ETM and PBM.

			ETM, per		PBM, per	
Task	ET_{max}	Period T_i	ET_{max}	Period	ET_{max}	Period
τ_1	24 µs	250 µs	17.5 %	1.7 %	-	-
τ_2	38 µs	250 µs	11.1 %	1.7 %	-	-
τ_3	113 µs	500 µs	3.7 %	0.8 %	-	-
τ_4	9 µs	500 µs	46.7 %	0.8 %	-	-
τ_5	13 µs	500 µs	32.3 %	0.8 %	-	-
τ_6	12 µs	1,000 µs	-	-	35.0 %	0.4 %
τ_7	16 µs	1,000 µs	-	-	-	-
Systemwide overhead				5.9 %		0.4 %

Table 4.6 compares the overhead for ETM and PBM in our test system. For ETM, each task with a higher priority than the critical task τ_6 requires monitoring, to prevent error propagation to τ_6, while for PBM, only the critical task τ_6 requires monitoring. In direct comparison to the tasks' measured maximum execution time ET_{max}, the monitoring overhead is between 3.7 % and 46.7 % per monitored task.

To put these figures into a systemwide perspective, a comparison to the period of a task is more meaningful than to ET_{max} because the period determines how often a task (and thus its monitor) is executed. For ETM, the overhead per period is between 0.8 % and 1.7 %, which results in an aggregated systemwide overhead of 5.9 %. For PBM, the overhead per period is 0.4 %, which results in an aggregated systemwide overhead of 0.4 %.

4.5.4 *Summary*

Our case study showed that PBM and ETM both protect critical tasks from failures due to timing error propagation in transient and permanent error scenarios. For transient errors, only PBM enables non-critical tasks to perform a transient error ride-through. In terms of overhead, PBM outperforms ETM by a magnitude. The expected benefit of using PBM over ETM in other scenarios largely depends on the distribution of critical and non-critical tasks, with a preference for PBM in scenarios with a low number, and for ETM in scenarios with a high number, of critical over non-critical tasks.

4.6 CONCLUSION

We have presented preemption budget monitoring (PBM), a novel monitoring approach that guarantees freedom from interference in the temporal domain to critical tasks in mixed-criticality systems. We have implemented our approach as an extension to the existing monitoring infrastructure of a widely used, commercial AUTOSAR OS with a 0.9 % increase in binary code size and less than 0.1 % increase in memory consumption. The evaluation of our approach in an adaptive cruise control scenario showed that PBM effectively prevents the propagation of timing errors from non-critical to critical tasks with a run-time overhead that is a magnitude lower than existing approaches. PBM achieves these impressive results by monitoring only critical tasks and avoiding the overhead of monitoring non-critical tasks. Therefore, we expect PBM to perform equally well for any mixed-criticality systems, in which few critical tasks require protection from possible failures of many non-critical tasks. Furthermore, in contrast to existing approaches, PBM enables transient ride-through to allow non-critical tasks to recover from transient timing errors and thereby improves overall system reliability.

Part V

SUMMARY AND CONCLUSION

5

SUMMARY AND CONCLUSION

Functional safety mechanisms assume a pivotal role in implementing the safety concept of a system, as they provide technical means for error detection and failure mitigation. For mixed-criticality systems, which integrate tasks of different criticality levels, they also serve the purpose of providing freedom from interference, i.e., preventing the propagation of errors from low to high criticality tasks. In consequence, the requirements on their dependable and trustworthy operation are correspondingly high.

In this thesis, we have explored two directions in the context of functional safety mechanisms:

1. The *assessment* of the correctness and robustness of these mechanisms, i.e., how they perform in the presence of specified (targeting correctness) and unspecified (targeting robustness) faults.
2. The *enhancement* of these mechanisms to mixed-criticality systems, i.e., how can these mechanisms be efficiently applied for mixed-criticality applications.

In conclusion, we have addressed each research question of Section 1.4 as follows.

Research Question (RQ1): How do we drive the instrumentation of complex, software-intensive mixed-IP systems for dependability assessment?

The FI-based assessment of software systems requires the instrumentation with test code for injectors and detectors. As the manual instrumentation of (often tool-generated) code is tedious and error-prone, the instrumentation process is usually automated. Model-based development, layered architectures, and the integration of components from various suppliers increase the complexity of a system and introduce multiple levels of abstraction. As the fault location (e.g., application component, runtime environment), the fault type (e.g., bit flip, fuzzing), and the accessibility of the injection location (i.e., source code or binary) can vary greatly across different fault models, instrumentation frameworks are facing the challenge of providing usability, customizability, and efficiency to the user.

Contribution (C1): A guidance framework for the dependability assessment of complex, software-intensive mixed-IP software systems.

In Chapter 2, we have developed an automated instrumentation framework for the dependability assessment of AUTOSAR-based systems. Our approach provides:

1. **Usability:** We enable the user to define their instrumentation requirements on the model level instead of the implementation level, which largely consists of automatically generated code.
2. **Customizability:** We provide expert users with the ability to further tune and refine their instrumentation choice, factoring implementation details, and thus test certain *aspects* of an interface, which are represented by the different instrumentation options and instrumentation locations within the software stack.
3. **Efficiency:** Our experimental results have shown that the use of interface wrappers for instrumentation infers low overhead in space and time.

We factor the different software component access levels (i.e., black--box, grey-box, and white-box) by enabling the instrumentation at the source code level (i.e., implementation and interface specification) and the binary object level. To demonstrate our approach and to show the generic applicability of the different instrumentation techniques, we have conducted a series of FI experiments on an anti-lock braking system. The cross-validated experimental evaluation yielded the result that the varied techniques were comparably efficient and that black-box instrumentation was as effective as white-box instrumentation while requiring less access to the system and being less intrusive. To provide guidance on the decision of instrumentation location and instrumentation options, we have discussed the qualitative criteria of code access, intrusiveness, automation complexity, and implementation effort.

In addition, we have identified systematic limitations of our approach and sketched possible solutions to resolve them. The implementation and evaluation of the proposed solutions is left for future work. The results of this work were presented at DSN 2012 [PWM+12].

The developed instrumentation approach and framework is not limited to FI scenarios. We have also demonstrated its application in the context of a model-based generator for complex runtime monitors, which was presented as joint work at ECMFA 2013 [PPP+13].

Research Question (RQ2): How can FI be effectively and efficiently applied for the assessment of functional safety mechanisms?

Functional safety mechanisms provide technical solutions to detect faults or control failures in order to achieve or maintain a safe state. Due to this pivotal role, the requirements on their dependability are correspondingly high. Fault injection (FI) is a versatile and established technique to assess a system's dependability in the presence of operational perturbations caused by hardware and software faults. The effective and efficient application of FI to modern software-intensive embedded systems is non-trivial, due to the complexity of these systems and the difficulty of identifying representative fault models. On the example of the AUTOSAR and ISO 26262 standards, we observe that, although FI is explicitly recommended to assess functional safety mechanisms, the standards provide little guidance on how experiments should be performed and how suitable fault models can be identified.

Contribution (C2): An open source FI framework for AUTOSAR, including fault model guidelines and the assessment of functional safety mechanisms.

In Chapter 3, we have developed and examined the complete FI process for the assessment of AUTOSAR's safety mechanisms, according to recommendations given by the ISO 26262 standard. To fill the gap between the ISO standard's recommendation to apply FI and the absence of suitable fault models, we have provided a detailed discussion on how we derived fault models for our assessment, which targets a commercial implementation of AUTOSAR's timing monitoring safety mechanisms. For the automated execution of FI experiments, we have adapted the open source FI framework GRINDER to AUTOSAR, which we have made openly available [PMT+15]. By providing this ready to use FI framework for AUTOSAR, we hope to foster the further development and use of FI for this target environment.

The conducted case studies uncovered an actual bug in the interaction of two timing monitoring mechanisms that could lead to a misidentification of the source of a timing error and negatively impact the effectiveness of error mitigation. The bug was subsequently acknowledged and fixed by the supplier of the safety mechanism's implementation. In summary our results demonstrate that

1. FI is an effective method to assess the implementation of functional safety mechanisms,
2. suitable fault models can be derived from functional specifications and the intended usage context, and
3. using these fault models actual deficiencies in the implementation can be identified with a modest amount of experiments.

Research Question (RQ3): How can the existing infrastructure of functional safety mechanisms be enhanced for mixed-criticality scenarios?

Mixed-criticality systems integrate tasks of different criticality levels. In order to ensure that errors originating in less critical functions can not propagate to disturb more critical functions, functional safety mechanisms are employed to provide freedom from interference, i.e., preventing error propagation from low to high criticality tasks. To provide efficient protection of critical tasks, functional safety mechanisms benefit from accounting for different criticality levels. At the example of AUTOSAR's timing protection, we illustrate the issues emerging from the lack of criticality awareness and the resulting indirect protection of critical tasks.

Contribution (C3): A novel, criticality-aware timing protection mechanism.

In Chapter 4, we have presented preemption budget monitoring (PBM), a novel monitoring approach that provides freedom from interference for the timing of critical tasks in mixed-criticality systems. To implement our approach, we have enhanced the existing monitoring infrastructure of a widely used, commercial AUTOSAR OS. Our approach causes a 0.9 % increase in binary code size and less than 0.1 % increase in memory consumption.

We have evaluated our approach for transient and permanent timing errors in an adaptive cruise control scenario. Our results have shown that PBM effectively prevents the propagation of timing errors from non-critical to critical tasks with a run-time overhead that is a magnitude lower than existing approaches. PBM achieves these impressive results by monitoring only critical tasks and avoiding the overhead of monitoring non-critical tasks. We expect PBM to perform equally well for any mixed-criticality systems, in which few critical tasks require protection from possible failures of many non-critical tasks. Furthermore, in contrast to existing approaches, PBM enables transient ride-through to allow non-critical tasks to recover from transient timing errors and thereby improves overall system reliability.

BIBLIOGRAPHY

[AAA+90] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. „Fault Injection for Dependability Validation: A Methodology and Some Applications." In: *Software Engineering, IEEE Transactions on*, vol. 16, no. 2 (Feb. 1990), pp. 166–182.

[ABD+95] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. „Fixed Priority Pre-emptive Scheduling: An Historical Perspective." In: *Real-Time Systems*, vol. 8, no. 2-3 (Mar. 1995), pp. 173–198.

[ABR+93] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. „Applying new scheduling theory to static priority pre-emptive scheduling." In: *Software Engineering Journal*, vol. 8, no. 5 (Sept. 1993), pp. 284–292.

[ALR+04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. „Basic Concepts and Taxonomy of Dependable and Secure Computing." In: *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[AUT11] AUTOSAR Release 3.2. *Technical Overview*. Document ID 067. Apr. 2011. URL: http://www.autosar.org/fileadmin/files/releases/3-2/main/auxiliary/AUTOSAR_TechnicalOverview.pdf.

[AUT14a] AUTomotive Open System ARchitecture (AUTOSAR). *Release 4.2.1*. Oct. 2014. URL: http://www.autosar.org/.

[AUT14b] AUTOSAR Release 4.1 Rev 3. *Guide to Multi-Core Systems*. Document ID 631. Mar. 2014. URL: http://www.autosar.org/fileadmin/files/releases/4-1/software-architecture/general/auxiliary/AUTOSAR_EXP_MultiCoreGuide.pdf.

[AUT14c] AUTOSAR Release 4.2.1. *Description of the AUTOSAR standard errors*. Document ID 377. Oct. 2014. URL: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_ErrorDescription.pdf.

[AUT14d] AUTOSAR Release 4.2.1. *Explanation of Error Handling on Application Level*. Document ID 378. Oct. 2014. URL: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_ApplicationLevelErrorHandling.pdf.

[AUT14e] AUTOSAR Release 4.2.1. *Overview of Functional Safety Measures in AUTOSAR*. Document ID 664. Oct. 2014. URL: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_FunctionalSafetyMeasures.pdf.

[AUT14f] AUTOSAR Release 4.2.1. *Requirements on Runtime Environment*. Document ID 083. Oct. 2014. URL: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/rte/auxiliary/AUTOSAR_SRS_RTE.pdf.

[AUT14g] AUTOSAR Release 4.2.1. *Specification of Operating System*. Document ID 034. Oct. 2014. URL: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/system-services/standard/AUTOSAR_SWS_OS.pdf.

[AUT14h] AUTOSAR Release 4.2.1. *Specification of RTE*. Document ID 084. Oct. 2014. URL: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf.

[AUT14i] AUTOSAR Release 4.2.1. *Technical Safety Concept Status Report*. Document ID 233. Oct. 2014. URL: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_TR_SafetyConceptStatusReport.pdf.

[BBD11] Sanjoy K. Baruah, Alan Burns, and Robert I. Davis. „Response-Time Analysis for Mixed Criticality Systems.“ In: *Proceedings of the 32nd Real-Time Systems Symposium (RTSS)*, 2011.

[BBD13] Sanjoy Baruah, Alan Burns, and Robert I. Davis. „An Extended Fixed Priority Scheme for Mixed Criticality Systems.“ In: *Proceedings of the 1st Workshop on Real-Time Mixed Criticality Systems (ReTiMiCS)*, 2013.

[BCL+14] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. „Mixed-criticality scheduling on multiprocessors.“ In: *Real-Time Systems*, vol. 50, no. 1 (Jan. 2014), pp. 142–177.

[BD14] Alan Burns and Robert I. Davis. *Mixed Criticality Systems - A Review*. Tech. rep. Department of Computer Science, University of York, UK, 2014.

[BFK10] Marc Born, John Favaro, and Olaf Kath. „Application of ISO DIS 26262 in practice.“ In: *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety (CARS)*, 2010, pp. 3–6. DOI: 10.1145/1772643.1772645.

[BFT09] Dominique Bertrand, Sébastien Faucou, and Yvon Trinquet. „An analysis of the AUTOSAR OS timing protection mechanism.“ In: *Proceedings of the 14th IEEE International Conference on Emerging Techonologies and Factory Automation (ETFA)*, 2009.

[BOR+14] Guilherme Baumgarten, Markus Oertel, Achim Rettberg, and Marcelo Götz. „First Results of Automatic Fault-Injection in an AUTOSAR Tool-chain.“ In: *Proceedings of the 12th IEEE International Conference on Industrial Informatics (INDIN)*, 2014, pp. 170–175.

[Bro06] Manfred Broy. „Challenges in Automotive Software Engineering.“ In: *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006, pp. 33–42.

[BSC13] Ricardo Barbosa, Nuno Silva, and João Mário Cunha. „csXception®: First Steps to Provide Fault Injection for the Development of Safe Systems in Automotive Industry.“ In: *Dependable Computing*. Ed. by Marco Vieira and João Carlos Cunha. Vol. 7869. Lecture Notes in Computer Science. Springer, 2013, pp. 202–205.

[CB89] Ram Chillarege and Nicholas S. Bowen. „Understanding Large System Failures - A Fault Injection Experiment.“ In: *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS)*, 1989, pp. 356–363. DOI: 10.1109/FTCS.1989.105592.

[CC96] Jörgen Christmansson and Ram Chillarege. „Generation of an Error Set that Emulates Software Faults Based on Field Data.“ In: *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS)*, 1996, pp. 304–313. DOI: 10.1109/FTCS.1996.534615.

[Cha09] Robert N. Charette. „This Car Runs on Code.“ In: *IEEE Spectrum* (Feb. 2009). URL: http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code.

[CLN+12] Domenico Cotroneo, Anna Lanzaro, Roberto Natella, and Ricardo Barbosa. „Experimental Analysis of Binary-Level Software Fault Injection in Complex Software.“ In: *Proc. of the 9th European Dependable Computing Conference (EDCC)*, 2012, pp. 162–172.

[Cun13] João Mário Quintas Cunha. „Fault injection for the evaluation of critical systems.“ MA thesis. Universidade do Minho, 2013. URL: http://hdl.handle.net/1822/27841.

[DM06] João A. Durães and Henrique S. Madeira. „Emulation of Software Faults: A Field Data Study and a Practical Approach.“ In: *Software Engineering, IEEE Transactions on*, vol. 32, no. 11 (Nov. 2006), pp. 849–867. DOI: 10.1109/TSE.2006.113.

[ETA] ETAS GmbH. *INTECRIO*. URL: http://www.etas.com/en/products/intecrio.php.

[FFR12] Christoph Ficek, Nico Feiertag, and Kai Richter. „Applying the AUTOSAR timing protection to build safe and efficient ISO 26262 mixed-criticality systems.“ In: *Proceedings of Embedded Real Time Software and Systems (ERTS²)*, 2012.

[Fre] Freescale Semiconductor, Inc. *MPC564xL: Ultra-Reliable Dual-Core 32-bit MCU for Automotive and Industrial Functional Safety Applications*. URL: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC564xL.

[FSF+13] Christoph Ficek, Maurice Sebastian, Nico Feiertag, Kai Richter, Marek Jersak, and Karsten Schmidt. „Software Architecture Methods and Mechanisms for Timing Error and Failure Detection According to ISO 26262: Deadline vs. Execution Time Monitoring.“ In: *Proceedings of the SAE World Congress*. 2013-01-0174, 2013.

[GE07] Jan Gustafsson and Andreas Ermedahl. „Experiences from Applying WCET Analysis in Industrial Settings.“ In: *Proceedings of the10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2007.

[GKT89] Ulf Gunneflo, Johan Karlsson, and Jan Torin. „Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation.“ In: *Digest of Papers of the 19th International Symposium on Fault-Tolerant Computing (FTCS)*, June 1989, pp. 340–347. DOI: 10.1109/FTCS.1989.105590.

[GNU] GNU Binutils. URL: http://www.gnu.org/software/binutils/.

[HKD11] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. „Security threats to automotive CAN networks–Practical examples and selected short-term countermeasures.“ In: *Reliability Engineering & System Safety*, vol. 96 (2011), pp. 11–25. ISSN: 0951-8320. DOI: 10.1016/j.ress.2010.06.026.

[HSF+04] Harald Heinecke, Klaus-Peter Schnelle, Helmut Fennel, Jürgen Bortolazzi, Lennart Lundh, Jean Leflour, Jean-Luc Maté, Kenji Nishikawa, and Thomas Scharnhorst. „AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures.“ In: *Convergence International Congress & Exposition On Transportation Electronics*, 2004, pp. 325–332.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. „Fault Injection Techniques and Tools.“ In: *IEEE Computer*, vol. 30, no. 4 (4 Apr. 1997), pp. 75–82. ISSN: 0018-9162.

[IKH+14] Mafijul Md. Islam, Nithilan Meenakshi Karunakaran, Johan Haraldsson, Fredrik Bernin, and Johan Karlsson. „Binary-Level Fault Injection for AUTOSAR Systems.“ In: *Proceedings of the 10th European Dependable Computing Conference (EDCC)*, 2014, pp. 138–141.

[Int10] International Electrotechnical Commission. *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. 2010.

[Int11] International Organization for Standardization. *ISO 26262: Road vehicles – Functional safety*. Geneva, Switzerland, 2011.

[ISA+13] Mafijul Md. Islam, Behrooz Sangchoolie, Fatemeh Ayatolahi, Daniel Skarin, Jonny Vinter, Fredrik Törner, Andreas Käck, Mattias Nyberg, Emilia Villani, Johan Haraldsson, Patrik Isaksson, and Johan Karlsson. „Towards Benchmarking of Functional Safety in the Automotive Industry.“ In: *Dependable Computing*. Ed. by Marco Vieira and JoãoCarlos Cunha. Vol. 7869. Lecture Notes in Computer Science. Springer, 2013, pp. 111–125.

[JSM07] Andréas Johansson, Neeraj Suri, and Brendan Murphy. „On the Selection of Error Model(s) For OS Robustness Evaluation.“ In: *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 502–511.

[KCR+10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayosi Kohno, Stepen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. „Experimental Security Analysis of a Modern Automobile.“ In: *IEEE Symposium on Security and Privacy (S&P)*, 2010, pp. 447–462. DOI: 10.1109/SP.2010.34.

[KKA95] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. „FERRARI: A Flexible Software-Based Fault and Error Injection System.“ In: *Computers, IEEE Transactions on*, vol. 44, no. 2 (Feb. 1995), pp. 248–260. DOI: 10.1109/12.364536.

[Kni02] John C. Knight. „Safety Critical Systems: Challenges and Directions.“ In: *Proceedings of the 24rd International Conference on Software Engineering (ICSE)*, 2002, pp. 547–550.

[Koo14] Phil Koopman. *A Case Study of Toyota Unintended Acceleration and Software Safety*. Presentation. Sept. 2014. URL: http://betterembsw.blogspot.de/2014/09/a-case-study-of-toyota-unintended.html.

[KP05] Raimund Kirner and Peter Puschner. „Classification of WCET Analysis Techniques.“ In: *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2005.

[LFK09a] Caroline Lu, Jean-Charles Fabre, and Marc-Olivier Killijian. „An approach for improving Fault-Tolerance in Automotive Modular Embedded Software.“ In: *Proceedings of the 17th International Conference on Real-Time and Network Systems (RTNS)*, 2009, pp. 132–147.

[LFK09b] Caroline Lu, Jean-Charles Fabre, and Marc-Olivier Killijian. „Robustness of modular multi-layered software in the automotive domain: a wrapping-based approach.“ In: *Proceedings of the 14th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2009, pp. 1–8. DOI: 10.1109/ETFA.2009.5347121.

[LL73] Chung Laung Liu and James W. Layland. „Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.“ In: *Journal of the ACM*, vol. 20, no. 1 (Jan. 1973), pp. 46–61.

[LNF10] Patrick E. Lanigan, Priya Narasimhan, and Thomas E. Fuhrman. „Experiences with a CANoe-based Fault Injection Framework for AUTOSAR.“ In: *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 569–574.

[LSD89] John Lehoczky, Lui Sha, and Ye Ding. „The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior.“ In: *Proceedings of Real Time Systems Symposium (RTSS)*, 1989.

[MB09] Gary Morgan and Andrew Borg. *Multi-core Automotive ECUs: Software and Hardware Implications*. Tech. rep. ETAS GmbH, 2009.

[MCV00] Henrique Madeira, Diamantino Costa, and Marco Vieira. „On the Emulation of Software Faults by Software Fault Injection.“ In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2000, pp. 417–426. DOI: 10.1109/ICDSN.2000.857571.

[MNS+10] Aurélien Monot, Nicolas Navet, Françoise Simonot, and Bernard Bavoux. „Multicore scheduling in automotive ECUs.“ In: *Proceedings of Embedded Real Time Software and Systems (ERTS²)*, 2010.

[NCD+13] Roberto Natella, Domenico Cotroneo, Joao A. Durães, and Henrique S. Madeira. „On Fault Representativeness of Software Fault Injection.“ In: *Software Engineering, IEEE Transactions on*, vol. 39, no. 1 (Jan. 2013), pp. 80–96.

[NLR09] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. „On the Scheduling of Mixed-Criticality Real-Time Task Sets.“ In: *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[NMR+02] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. „CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs.“ In: *Compiler Construction*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 213–228. ISBN: 978-3-540-43369-9. DOI: 10.1007/3-540-45937-5_16.

[Opt] OptXware Ltd. *Embedded Architect*. URL: http://www.optxware.com/en/embedded/embedded-architect-platform/.

[PFK+13] Ludovic Pintard, Jean-Charles Fabre, Karama Kanoun, Michel Leeman, and Matthieu Roy. „Fault Injection in the Automotive Standard ISO 26262: An Initial Approach.“ In: *Dependable Computing*. Ed. by Marco Vieira and João Carlos Cunha. Vol. 7869. Lecture Notes in Computer Science. Springer, 2013, pp. 126–133.

[PFL+14] Ludovic Pintard, Jean-Charles Fabre, Michel Leeman, Karama Kanoun, and Matthieu Roy. „From Safety Analyses to Experimental Validation of Automotive Embedded Systems.“ In: *Proceedings of the 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2014, pp. 125–134.

[PMT+15] Thorsten Piper, Paul Manns, Michael Tretter, and Stefan Winter. *AUTOGRINDER: GRINDER adaptation to AUTOSAR*. 2015. URL: https://github.com/DEEDS-TUD/AUTOGRINDER.

[PPP+13] Lars Patzina, Sven Patzina, Thorsten Piper, and Paul Manns. „Model-Based Generation of Run-Time Monitors for AUTOSAR.“ In: *Modelling Foundations and Applications*. Ed. by Pieter Van Gorp, Tom Ritter, and Louis M. Rose. Vol. 7949. Lecture Notes in Computer Science. **Winner of the Best Paper Award**. Springer Berlin Heidelberg, 2013, pp. 70–85. ISBN: 978-3-642-39012-8. DOI: 10.1007/978-3-642-39013-5_6.

[PWM+12] Thorsten Piper, Stefan Winter, Paul Manns, and Neeraj Suri. „Instrumenting AUTOSAR for Dependability Assessment: A Guidance Framework.“ In: *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1–12.

[PWS+15a] Thorsten Piper, Stefan Winter, Oliver Schwahn, Suman Bidarahalli, and Neeraj Suri. „Mitigating Timing Error Propagation in Mixed-Criticality Automotive Systems.“ In: *Proceedings of the 18th IEEE International Symposium On Real-time Computing (ISORC)*, 2015.

[PWS+15b] Thorsten Piper, Stefan Winter, Neeraj Suri, and Thomas E. Fuhrman. „On the Effective Use of Fault Injection for the Assessment of AUTOSAR Safety Mechanisms.“ In: *Proceedings of the 11th European Dependable Computing Conference (EDCC)*, 2015.

[RM14] Dominik Reinhardt and Gary Morgan. „An Embedded Hypervisor for Safety-Relevant Automotive E/E-Systems.“ In: *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2014.

[RMM+10] Ishtiaq Rouf, Rob Miller, Hossen Mustafa, Travis Taylor, Sangho Oh, Wenyuan Xu, Marco Gruteser, Wade Trappe, and Ivan Seskar. „Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study.“ In: *Proceedings of the 19th USENIX Security Symposium*. Washington, DC, USA, 2010.

[Rus99] John Rushby. *Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance*. NASA Contractor Report CR-1999-209347. NASA Langley Research Center, June 1999.

[SBC+13] Nuno Silva, Ricardo Barbosa, João Carlos Cunha, and Marco Vieira. „A View on the Past and Future of Fault Injection.“ In: *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–2.

[SBK10] Daniel Skarin, Raul Barbosa, and Johan Karlsson. „GOOFI-2: A Tool for Experimental Dependability Assessment.“ In: *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 557–562.

[SHK+12] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. „FAIL*: Towards a Versatile Fault-Injection Experiment Framework.“ In: *Proceedings of Architecture of Computing Systems (ARCS) Workshops*, 2012.

[SK08] Daniel Skarin and Johan Karlsson. „Software Implemented Detection and Recovery of Soft Errors in a Brake-by-Wire System.“ In: *Proceedings of the 7th European Dependable Computing Conference (EDCC)*, 2008, pp. 145–154. DOI: 10.1109/EDCC-7.2008.24.

[SLR86] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. „Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.“ In: *Proceedings of the 7th IEEE Real-Time Systems Symposium (RTSS)*, 1986.

[SPS12] As'ad Salkham, Antonio Pecchia, and Nuno Silva. „Assessing AUTOSAR Systems Using Fault Injection.“ In: *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2012, pp. 11–12.

[SPS13] As'ad Salkham, Antonio Pecchia, and Nuno Silva. „Design of a CDD-Based Fault Injection Framework for AUTOSAR Systems.“ In: *Proceedings of the International Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR)*, 2013.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. „Priority Inheritance Protocols: An Approach to Real-Time Synchronization.“ In: *Computers, IEEE Transactions on*, vol. 39, no. 9 (Sept. 1990), pp. 1175–1185.

[SVE+10] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren. „MODIFI: A MODel-Implemented Fault Injection Tool.“ In: *Computer Safety, Reliability, and Security*. Ed. by Erwin Schoitsch. Vol. 6351. Lecture Notes in Computer Science. Springer, 2010, pp. 210–222. ISBN: 978-3-642-15650-2.

[SVS+88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. „FIAT – Fault Injection Based Automated Testing Environment.“ In: *Digest of Papers of the 18th International*

Symposium on Fault-Tolerant Computing (FTCS), June 1988, pp. 102–107. DOI: 10.1109/FTCS.1988.5306.

[TP13] Anna Thomas and Karthik Pattabiraman. „LLFI: An Intermediate Code Level Fault Injector For Soft Computing Applications." In: *Proceedings of Silicon Errors in Logic System Effects (SELSE)*, 2013.

[VAV+14] Benjamin Vedder, Thomas Arts, Jonny Vinter, and Magnus Jonsson. „Combining Fault-Injection with Property-Based Testing." In: *Proceedings of the International Workshop on Engineering Simulations for Cyber-Physical Systems (ES4CPS)*, 2014, pp. 1–8.

[Ves07] Steve Vestal. „Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance." In: *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, 2007.

[Voa98] Jeffrey M. Voas. „Certifying Off-the-Shelf Software Components." In: *IEEE Computer*, vol. 31, no. 6 (1998), pp. 53–59. DOI: 10.1109/2.683008.

[WEK10] Armin Wasicek, Christian El-Salloum, and Hermann Kopetz. „A System-on-a-Chip Platform for Mixed-Criticality Applications." In: *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2010.

[Whe] David A. Wheeler. *SLOCCount: Tools for counting physical Source Lines of Code*. URL: http://www.dwheeler.com/sloccount/.

[WPS+15] Stefan Winter, Thorsten Piper, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. „GRINDER: On Reusability of Fault Injection Tools." In: *Proceedings of the IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2015.

[WSS+11] Stefan Winter, Constantin Sârbu, Neeraj Suri, and Brendan Murphy. „The Impact of Fault Models on Software Robustness Evaluations." In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 51–60.

[WTS+13] Stefan Winter, Michael Tretter, Benjamin Sattler, and Neeraj Suri. „simFI: From Single to Simultaneous Software Fault Injections." In: *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[ZBS+14] Michael Zimmer, David Broman, Christopher Shaver, and Edward A. Lee. „FlexPRET: A Processor Platform for Mixed-Criticality Systems.“ In: *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*, 2014.