Instrumenting AUTOSAR for Dependability Assessment: A Guidance Framework

Thorsten Piper, Stefan Winter, Paul Manns and Neeraj Suri
Technische Universität Darmstadt, Germany
{piper,sw,manns,suri}@cs.tu-darmstadt.de

***Abstract*—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 paper 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.

***Keywords*-AUTOSAR; instrumentation; interface wrappers; fault injection; run-time monitoring**

I. Introduction

AUTOSAR (AUTomotive Open System ARchitecture) [9] 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 of different suppliers.

Automobiles are safety-critical systems 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" [4], manufacturers comply to industry-wide safety standards and functional safety specifications, such as IEC 61508 [12] and in particular the recently released ISO 26262 ([13], [5]), 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, 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 [21]. 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 ([14], [20], [10]), 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 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.

Paper Contributions

Facing these challenges, we aim to 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

[1]Throughout this paper, 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.

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 [22], 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 the paper as follows. We introduce the system model in Section II and review related work in Section III. In Section IV we investigate the development of a systematic, automated process for instrumentation. We evaluate our approach in Section V and discuss its characteristics and limitations in Section VI.

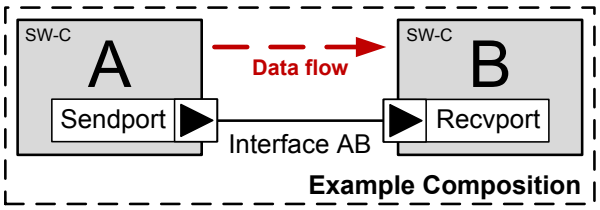

Fig. 1. Model of two software components (SW-Cs) communicating via a sender-receiver interface.

II. The 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 ([6],[18]). This section provides the foundations to understand key aspects of the AUTOSAR development process/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 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 communicate 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

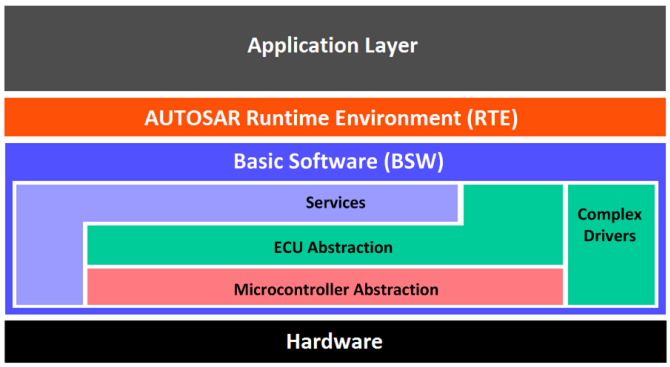

Fig. 2. The AUTOSAR layered software architecture [1].

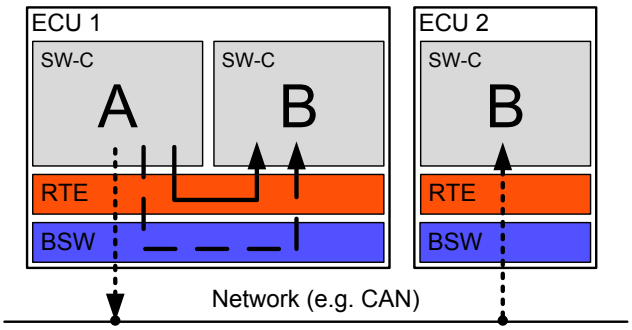

Fig. 3. Possible data flow paths of two communicating SW-Cs at the implementation level.

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 as 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. 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 sublayers and modules, and provides the hardware abstraction.

Recalling the example model of Figure 1, the communication between components A and B can result in three distinct communication paths in the implementation, as shown in Figure 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 generation 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 IV. 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.

III. Related Work

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 [16], 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/stop, 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 [17], 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. [15] 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 [16] 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. Our publication is the first to address these open issues.

IV. 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.

A. 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 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 1, with the intent to provide an illustrative example and give the reader a glimpse at the overall process.

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

Listing 1. Component prototype specification (extract from the ARXML of the model in Figure 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 [3], 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 1 specifies the port name (*SendPort*) while the DataElementPrototype can be obtained by de-referencing

the interface reference in lines 9-10. Listing 2 shows the interface specification. The name of the DataElementPrototype (*DataPrototype*) is located in line 4.

```
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. Interface specification (extract from the ARXML of the model in Figure 1).

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 that of the actual generated code, as in Listing 3.

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

Listing 3. Communication primitive generated from the ARXML specification in Listings 1 and 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.

B. Opportunities for Instrumentation

The AUTOSAR standard document "Requirements on RTE Software" [2] 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*". 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 VI. 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 [8]. 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.

C. 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.

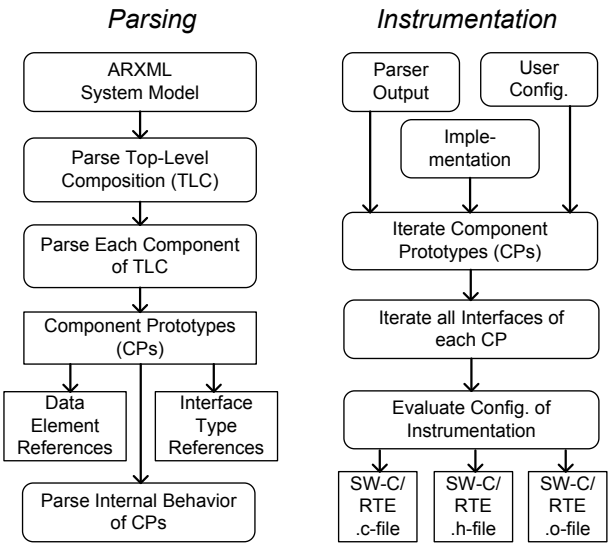

Fig. 4. Automating instrumentation: Basic workflow of the model parsing and instrumentation phases.

The overall workflow is divided into three phases: parsing, configuration and instrumentation, as depicted in Figure 4. 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 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* [3] 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. During 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 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.

V. 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* [7] and *OptXware Embedded Architect (EA) V1.0.0.201103031241* [19], 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 IV.

A. The Experimentation Setup

The system on which we implement the proof of concept is a simplified anti-lock braking system, as shown in Figure 5. 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 V-B. 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*

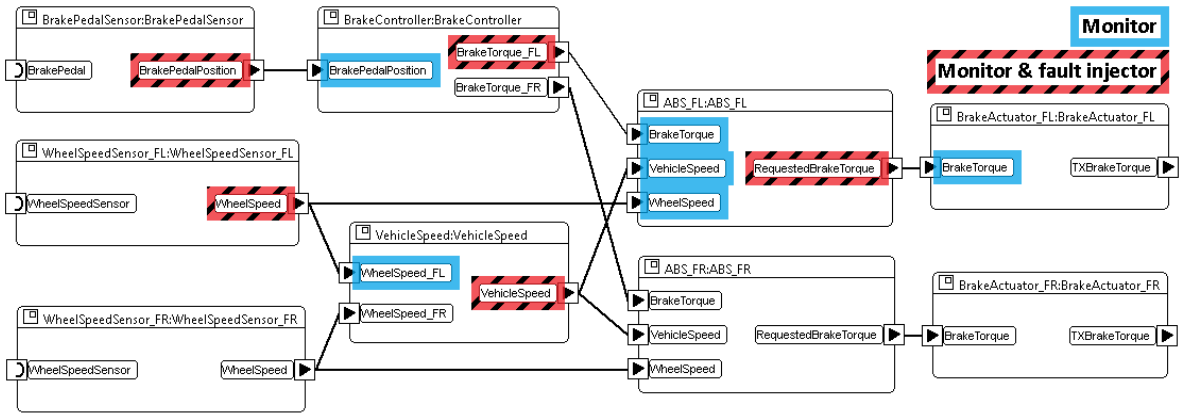

Fig. 5. Model of an anti-lock braking system (ABS), instrumented with monitors and fault injectors at selected interfaces.

.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.

B. 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 m/s^2 in the optimal case of non-blocking wheels and -6 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.

C. 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 [11] 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 300ms after the simulator has initiated a full application of the brake. As all interfaces in our example system transmit 16-bit values, each injection

TABLE I
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

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 I. 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 1940ms 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 evaluator the desired instrumentation choices as based on the access and implementation/execution criteria of Section V-D and Section VI.

D. 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.

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 [23], a set of tools for counting physical source lines of code (SLOC). Table II and Table III 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 IV, 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.

TABLE II
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 III
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

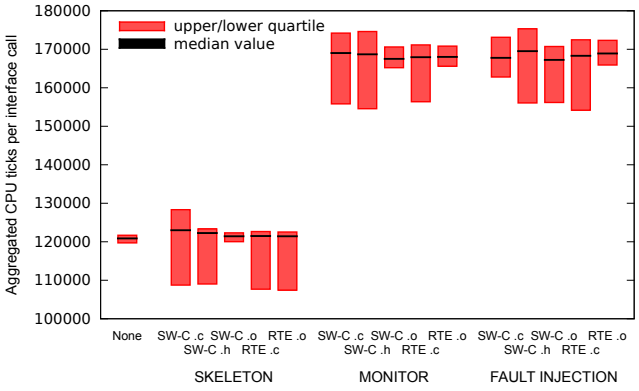

Fig. 6. ETAS INTECRIO: Relative comparison of the execution time of instrumentation methods, grouped by implemented functionality.

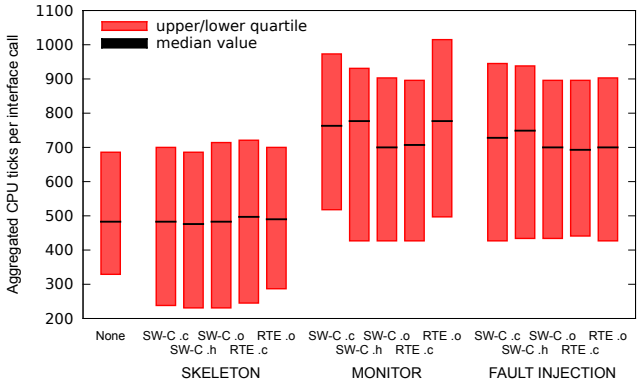

Fig. 7. OptXware EA: Relative comparison of the execution time of instrumentation methods, grouped by implemented functionality.

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) 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. Figures 6 and 7 depict boxplots of the accumulated runtime of all instrumented interface 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.

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 [8]. 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 IV.

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)*.

VI. 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.

A. 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 V.

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 IV-B, 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 IV-C, 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 each instrumentation method's requirements 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

TABLE IV
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 V
RELATIVE COMPARISON OF INSTRUMENTATION METHOD AND LOCATION FOR DIFFERENT QUALITY ATTRIBUTES.

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

Legend: ★ poor, ★ ★ ★ good

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 V, 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.

B. 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.

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 due to 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 one being *runnable wrappers*, which can be implemented on the SW-C and RTE side. The second one being 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 their 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. Primarily the experimental results show that both approaches are suitable for systematic and automatic instrumentation of AUTOSAR systems using shared memory communication.

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.

VII. Conclusion

In this paper, 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 (black-box and white-box) that are prevalent in AUTOSAR systems, we enable the instrumentation at the source code (implementation, interface specification) and binary object levels. 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 the user freedom of choice 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 discuss 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.

Acknowledgment

We thank ETAS Group and OptxWare Research & Development Ltd. for the provision of their tools and for their support. This work was supported by CASED (www.cased.de).

References

[1] AUTOSAR GbR, "Technical Overview," Document ID 067, 2008.
[2] AUTOSAR GbR, "Requirements on RTE Software," Document ID 083, 2009.
[3] AUTOSAR GbR, "Specifiation of RTE," Document ID 084, 2010.
[4] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, pp. 11–33, January 2004.
[5] M. Born, J. Favaro, and O. Kath, "Application of ISO DIS 26262 in practice," in *Proc. of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, 2010, pp. 3–6.
[6] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Proc. of the 26th Symposium on Fault-Tolerant Computing (FTCS)*, 1996, pp. 304–313.
[7] ETAS Group GmbH, "INTECRIO," http://www.etas.com/en/products/intecrio.php.
[8] GNU Binutils, http://www.gnu.org/software/binutils/.
[9] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. 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.
[10] T. Hoppe, S. Kiltz, and J. Dittmann, "Security threats to automotive CAN networks–Practical examples and selected short-term countermeasures," *Reliability Engineering & System Safety*, vol. 96, no. 1, pp. 11–25, 2011.
[11] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, pp. 75–82, April 1997.
[12] International Electrotechnical Commission, "IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems," 2010.
[13] International Organization for Standardization, "ISO/FDIS 26262: Road vehicles – Functional safety," 2011.
[14] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental Security Analysis of a Modern Automobile," in *IEEE Symposium on Security and Privacy*, 2010, pp. 447–462.
[15] P. Lanigan, P. Narasimhan, and T. Fuhrman, "Experiences with a CANoe-based fault injection framework for AUTOSAR," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 569–574.
[16] C. Lu, J.-C. Fabre, and M.-O. Killijian, "An approach for improving Fault-Tolerance in Automotive Modular Embedded Software," in *Proc. of the 17th International Conference on Real-Time and Network Systems (RTNS)*, 2009.
[17] C. Lu, J.-C. Fabre, and M.-O. Killijian, "Robustness of modular multi-layered software in the automotive domain: a wrapping-based approach," in *Proc. of the 14th IEEE International Conference on Emerging Technologies & Factory Automation*, 2009, pp. 1102–1109.
[18] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2000, pp. 417 –426.
[19] OptXware Ltd., "Embedded Architect," http://www.optxware.com/en/embedded/embedded-architect-platform.
[20] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar, "Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study," in *Proc. of the 19th USENIX Security Symposium*, Aug. 2010.
[21] D. Skarin and J. Karlsson, "Software Implemented Detection and Recovery of Soft Errors in a Brake-by-Wire System," in *Seventh European Dependable Computing Conference (EDCC)*, 2008, pp. 145–154.
[22] J. Voas, "Certifying off-the-shelf software components," *Computer*, vol. 31, no. 6, pp. 53–59, 1998.
[23] D. A. Wheeler, "SLOCCount," http://www.dwheeler.com/sloccount/.