2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing

Application-Driven Evaluation of AUTOSAR Basic Software on Modern ECUs

Norbert Englisch, Felix Hänchen, Frank Ullmann, Alejandro Masrur and Wolfram Hardt
Department of Computer Science
TU Chemnitz, Germany

***Abstract*—When integrating AUTOSAR software on an automotive ECU, errors may occur due to the large number of modules involved and/or improper timing. These errors manifest at the application level complicating the test and verification process. Since AUTOSAR has a layered architecture, it is often cumbersome to identify sources of errors. In this paper, to help integrating software on an ECU, we propose a technique to verify functionality and timing of generated AUTOSAR modules in a semi-automated manner. Our technique consists in defining test cases based on the interface descriptions of AUTOSAR modules and application software. This allows reliably identifying AUTOSAR modules affected by functional and/or timing errors and simplifies the test and verification process. We illustrate the benefits by our technique by means of a case study performed on the real hardware.**

I. Introduction

Traditionally, in the automotive domain, application software was developed for a specific platform consisting of a specific operating system (OS) and ECU (Electronic Control Unit). The introduction of AUTOSAR (AUTomotive Open System ARchitecture) [1] has led to a shift in the design and development of automotive software, which has now become platform-independent. Application software based on AUTOSAR is developed independently of the underlying software and hardware architecture which increases portability and flexibility.

Basically, AUTOSAR divides automotive software into three layers (see Fig. 1): the application software, the runtime environment (RTE) and the basic software (BSW) layers [2]. The software developer focuses on implementing application software components (SWCs), whereas the RTE and all necessary BSW components such as OS, communication stack, etc. are *generated* and *configured* according to what services are being required by the application.

Available tools usually undertake most of the generation and configuration overhead for RTE and BSW. However, due to the large number of modules/components and interactions between them, timing and/or functional errors (e.g., due to improper configuration) may still occur at this level. These manifest at the application layer and complicate the integration of software onto the ECU. Moreover, it is difficult to test all possible combinations of configuration parameters from the different AUTOSAR modules without considering application-specific details. As a result, to integrate AUTOSAR software on an ECU, i.e., including the BSW and the RTE, it becomes necessary to consider application-dependent configuration parameters.

In industry, XCP (Universal Measurement and Calibration Protocol) is commonly used to test and verify automotive

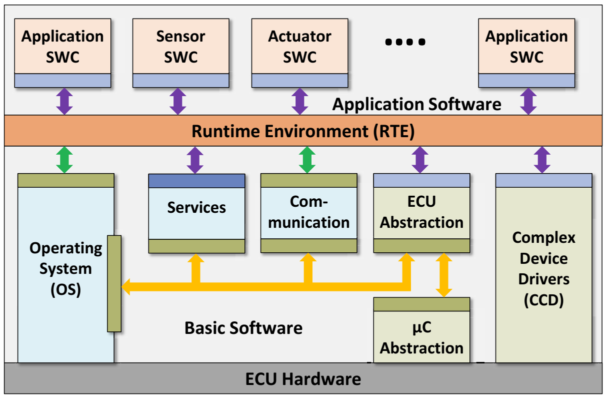


Fig. 1. Overview about components of AUTOSAR ECU

software [3]. With XCP, an external calibration system communicates with the target ECU over some communication medium (e.g., CAN, FlexRay, Ethernet, etc.). On its part, the ECU is flashed with a software module, viz., the XCP driver, which interprets commands from the calibration system. The XCP driver retrieves data on demand directly from the ECU's memory. The developer can then analyze this data towards testing and verifying the ECU.

However, for XCP-based testing, complex tools such as CANape [4] are necessary. These are often difficult to use and require the developer to go through expensive training. Moreover, XCP is not available for all AUTOSAR versions currently being used – only versions from 4.0 onwards are supported [5] – and cannot be easily automated resulting in a non-negligible overhead for the developer. In particular, if first only software need to be integrated onto the ECU. In this paper, we propose a technique that allows automating functional and timing tests of automotive ECUs towards a smooth integration of software upon it. In contrast to XCP, our technique does not require special training and can be applied to all available AUTOSAR versions.

Contributions: We propose a *semi-automated* technique for testing AUTOSAR software modules in the integration phase on the target platform or ECU. In contrast to XCP where the XCP driver has to be aware of hardware features, our technique is platform-independent and portable, since it only makes use of the standard interfaces specified by AUTOSAR. In addition, this method can be implemented using standard AUTOSAR development tools, which are anyway required for application

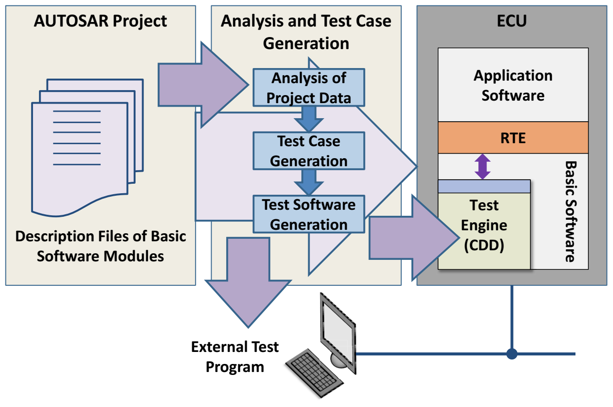


Fig. 2. Overview of the proposed concept for testing automotive software at the integration phase

development and, hence, already available to the developer. In contrast, XCP requires additional licenses to those usually needed for developing AUTOSAR software. The proposed technique, which is illustrated in Fig. 2, can be summarized in the following manner:

- Our technique consists in generating test cases based on the interface descriptions of the application and AUTOSAR BSW modules involved. The user can specify whether all or individual *signals*, i.e., variables exchanged between software modules, should be tested.
- These test cases are performed by our test engine which is flashed in the form of a *complex device driver* (CDD) module – see Fig. 2 – onto the ECU to be tested. The test engine is generated automatically once test cases have been created and interacts with the AUTOSAR software on the ECU by using platform-independent function calls.
- A configuration file is further generated for an external test program which then runs automatically on a laptop or computer connected to the ECU – different communication protocols can be used such as CAN, FlexRay, etc. The external program then communicates with our test engine and controls the execution of tests on the ECU, i.e., starts/stops tests, retrieves test data, etc.

Once the user has decided which signals to test, test cases are automatically generated. These allow testing functionality and timing of software. Functionality is tested by feeding in specific values to software modules and comparing their outputs with the expected outputs. If an output does not match what it is expected to be the outcome, the test engine signalizes an error making reference to the tested module.

In a similar manner, if a software module does not trigger a functional error, its timing behavior can further be *measured*. To this end, our test engine starts a timer at the point of feeding in an input and stops it when the output is delivered. This procedure is repeated a number of times with different input values so as to obtain an average timing of the tested module.

Of course, this is only an estimate and cannot be used to provide any timing guarantees. However, our experience indicates that developers usually need to approximately know how long software modules take for processing data in order to be able to design application software. These timing estimates are particularly important when considering that timing behavior is not specified in the current AUTOSAR versions such as 3.x and 4.0 – timing properties are currently being specified from version 4.0.2 onwards [6].

In addition, since there are multiple tool vendors offering different implementations of AUTOSAR software modules, it is often the case that timing properties strongly differ from one to the other. As a consequence, the proposed technique further helps software designers and developers to select a proper module implementation for specific application requirements.

The remainder of the paper is organized as follows: Section II gives an overview of related work and approaches. Section III briefly discusses the problems related to testing automotive software, whereas Section IV introduces and discusses our proposed test approach. Section V illustrates its applicability with a case study. Finally, we discuss use cases for the proposed method in Section VI, and summarize the results of this paper in Section VII.

II. Related Work and Approaches

A black-box testing of AUTOSAR software is difficult since it often cannot be clearly determined where errors occur. Errors can happen in different software layers (i.e., basic or application software) or even different software modules. To restrict the number of potential sources of error, all software layers need to be tested separately.

In practice, testing AUTOSAR software strongly focuses on the application layer, since this contains the functionality implemented by the user. In line with this, different approaches have been proposed for testing application software [7], [8]. However, these approaches do not help at the software integration phase, since they do not take AUTOSAR basic software into account.

On the other hand, the RTE and the BSW layers are usually checked against the AUTOSAR specification [9], but not against the application. As a result, it might happen that the different software modules pass their respective conformance tests, but errors originate when they are integrated onto the ECU. For this reason, it is necessary to check the BSW in an application-aware context.

As mentioned above, in industry, XCP is used with this purpose [3]. This is a standard for measuring and calibrating ECUs. In order to use XCP in AUTOSAR, it is necessary to include the XCP driver into the basic software. In addition, all software modules to be tested need to undergo an extra XCP-related configuration which is a cumbersome procedure for the developer. In contrast to this, our approach is applicable to all available AUTOSAR versions. Moreover, there is no extra configuration required for software modules to be tested which remain unmodified.

Timing analysis techniques for AUTOSAR have been proposed before. In [10], the authors propose using SystemC in the

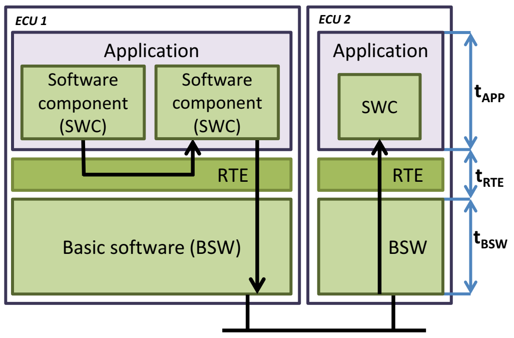


Fig. 3. Inter- and intra-ECU communication

AUTOSAR development process to define the timing behavior of applications. In [11], the end-to-end timing of AUTOSAR software is analyzed based on high-level models of the system. In [12] and [13], a timing-augmented description language is used in context of AUTOSAR software development. Later, in [14], an XCP-based approach was presented to measure the timing properties of AUTOSAR software. Our work is in line with the latter approach. We also measure timing properties of the AUTOSAR software; however, in contrast to [14], we do not use XCP and hence have less configuration overhead. In addition, the proposed technique is automated to a great extent allowing us to speed up the software integration process on an automotive ECU.

Almost all modules of the BSW can be covered with our approach. The only exception are CDD modules – see Fig. 1. These software modules are usually highly hardware-dependent and are not subject to interface restrictions of AUTOSAR (unless they have connections to the application layer through the RTE). To test these modules, an approach has been proposed which consists in virtual integration, i.e., simulation [15]. This approach allows testing a CDD module on a standard desktop computer by *emulating* the hardware drivers it is connected to [15]. In this way, a CDD module can be integrated into a *virtual integration platform* and tested as if it would be running on the target ECU.

III. Testing AUTOSAR Software

Clearly, developers can debug and test application software efficiently. However, even if all SWCs have a correct functional and timing behavior, errors may still happen when integrating multiple SWCs onto an ECU. To test automotive software once integrated on the ECU, we distinguish between intra- and inter-ECU communication.

According to AUTOSAR, SWCs can transparently communicate with other SWCs on the same ECU or on other ECUs, i.e., SWCs are unaware of the location of other SWCs. The RTE is then in charge of *routing* data from one SWC to the other. If an SWC sends data to another SWC on the same ECU (i.e., intra-ECU communication), this is directly handled by the RTE without intervention of the BSW – see in Fig. 3. On the other hand, if the SWC communicates with an SWC on another ECU (i.e., inter-ECU communication), the RTE passes data to the BSW layer as illustrated by the downward arrow in Fig. 3.

The intra-ECU communication case can be easily handled from the application layer. Some techniques have already been proposed for this [8], [7]. For example, timing properties can be easily measured from the application layer, since the BSW layer does not intervene. As a result, in the remaining part of the paper, we mainly focus on inter-ECU communication case and, in particular, on testing the interactions between the application and the BSW layer in that case.

IV. Proposed Approach

Fig. 4 illustrates the semi-automated workflow of our approach. This is divided into three phases: (i) analysis of project data, (ii) test case generation, and (iii) test software generation. In the first phase, all necessary data is extracted from the AUTOSAR project containing the application software. This contains information about which BSW modules are required by the application, the configuration of such modules, etc.

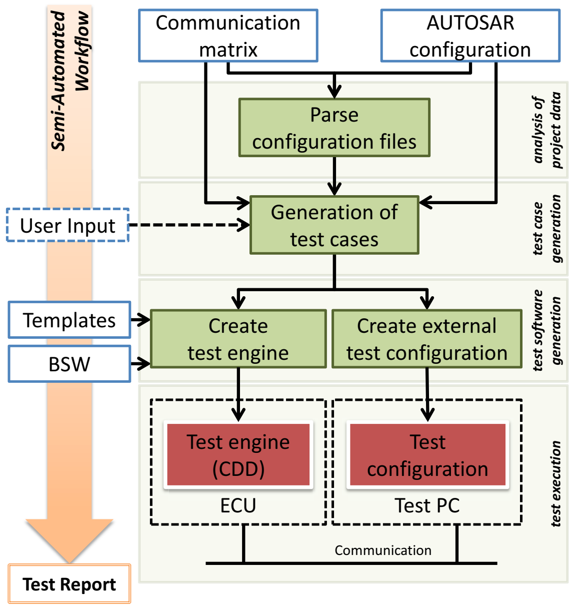


Fig. 4. Workflow of the proposed approach

In the second phase, we generate test cases based on the information extracted from the AUTOSAR project and the interface information of the BSW modules involved. The test cases are created such that they check the AUTOSAR stack successively. For example, if a BSW module is a cascaded arrangement of three submodules, our test cases will first stimulate the first submodule alone, then the first and the second one, and finally the first, the second and the third submodule. If the output of stimulating the first submodule is correct and an error occurs after stimulating the first and the second submodule, then the second submodule is clearly affected by errors being reported to the developer. Timing properties are measured only for those modules which have passed the functional tests.

In third phase, our test engine is generated using predefined templates in C-code, which is then integrated into the AUTOSAR BSW as a complex device driver (CDD) module. The test engine contains a program with the sequence of tests that are to be performed. This sequence of tests needs to be synchronized with the external test program, which controls the test process and evaluates results. Results are then reported to the user. Besides other possible formats, test results can be provided as a tabular HTML page or as a CSV file. In the following, we describe these three phases in more detail.

A. Analysis of Project Data

In this phase, data related to BSW modules is extracted from the AUTOSAR configuration files. These files are available in XML format and can be easily evaluated by an XML parser. Thereby the parser extracts information like length in bits, name and data type of all signals, which is used by the application software. All necessary XML tags, which contain this information, can be found in the AUTOSAR specification of the specific module.

AUTOSAR software is composed of different *stacks* such as for communication, i.e., CAN, FlexRay, etc. If one or more such stacks are used by the application, each of them needs to be analyzed with its corresponding modules. An *object* is then created for each module of the BSW stack, which models its internal structure. An object contains the module properties which are defined in the AUTOSAR specification, i.e., interfaces, data types, data ranges, etc. These objects provide a quick and easy access to data structures for the automated test case generation that follows. Before objects are forwarded to the test case generator, a plausibility check needs to be performed, which then verifies whether data types are correct and whether data values are in the corresponding valid ranges. Moreover, the plausibility check verifies that AUTOSAR modules have a consistent version.

B. Test Case Generation

The testing is performed in a black-box fashion. This means that the internal structure of the individual software modules is not relevant for us. BSW modules are stimulated with a given signal and the result of that stimulation is checked for its correctness [16]. To this end, the expected result must be computed for a given stimulus and stored together with the test case. The stimulus and the expected result are generated automatically. The input value/stimulus is extracted from the interface configuration of a basic software module, whereas the corresponding expected result value is calculated using input value and functional description of a module. Functional descriptions of modules need to be obtained from the AUTOSAR specifications – to demonstrate the presented approach we have done this for a subset of the AUTOSAR modules, but this can be extended to all of them.

The test case generator proceeds iteratively for each BSW stack. Once these stacks and their corresponding modules are identified, test cases for each of the BSW modules will be defined. As discussed above, test cases are generated in a successive manner. That is, the transportation of signals through the different modules of a BSW stack is tested successively. If the signal does not arrive at a given module, this is reported to the developer.

The use of a suitable test strategy is very important because an exhaustive test of all possible values is seldom possible. To test the BSW modules, the equivalence class method and the extreme value analysis are implemented [16]. The equivalence class method consists in defining different sets of input values. All the values in one such set have an identical effect on the test module. As a result, one representative value is chosen as a stimulus from each of the sets as stimulus. On the other hand, the extreme value method consists in choosing the maximum or the minimum data value as a stimulus that is in the possible range of input values. Our framework can further be extended to integrate user-defined test cases.

Algorithm 1 shows the procedure to generate test cases for a BSW module. In the first line, necessary updates of the communication matrix are defined. This function adds (if non-existent) a control message to the communication matrix, which controls the test engine on the target ECU and forwards test results to the external test program (running on the test PC). In the second line all testable functions of this module are returned. At this step, it is analyzed whether a function of a specific module to be tested has a measurable output or not – only in this case we can measure and compare it to the expected value. This function accesses a look-up table including a list of *testable* functions for each BSW module, i.e., functions with inputs and outputs that can be tested with our method. Currently, this look-up table is manually created based on the AUTOSAR specification. Its contents can also be generated by parsing the source code – checking cross-references between BSW modules. However, this is not implemented in the current version of the proposed technique.

A for-loop then iterates over all testable functions, where each time we analyze the signals that are sent and received by these functions. More specifically, for each signal, *getSignalValues()* is called – see line 7, which automatically generates a set of test data based on the configuration of the AUTOSAR module (e.g., data type and range of its ports, etc.).

The test data generated by *getSignalValues()* follows the extreme value method (i.e., the where minimum and the maximum value of a signal are checked) and the equivalence class method. By the equivalence class method, the range of a given data type is divided into a configurable number of intervals. A representative value is then selected for each interval and used as test value. This procedure can be extended by other methods the user may require.

In the next step, for each BSW module, expected output values are successively computed for each test data value obtained before. Expected output values are obtained by means of a functional model of each module, which describes its behavior. At the current state, we have implemented such functional models with Matlab Simulink. Clearly, the Simulink models need to be created manually for each BSW module based on the AUTOSAR specification. However, once these models are obtained, they allow automating the computation of expected output values from given test data values.

After the test data and expected output values have been obtained, test cases can be constructed that are later executed on the ECU. This requires generating control sequences between our test engine and the external test program. Afterwards, the corresponding code should be generated for the test engine, which specifies which input values a specific signal of a BSW module should be stimulated with and which output values should be expected. Finally, the generated test engine has to be

Algorithm 1 Analysis, test case generation, and test software generation for one BSW module

```
Require: Set of AUTOSAR project files and functional description of module
 1: Extend CommunicationMatrix by Control Messages
 2: getTestableFunctionsOfModule()
 3: for all testable functions of module do
 4:     for all RxSignals/TxSignals of testable function do
 5:         getSignalValues()
 6:         for all SignalValues do
 7:             getExpectedValue()
 8:             Generate TestCases
 9:             Generate ControlSequences for Test Program
10:             Generate ControlSequences for Test Engine
11:         end for
12:     end for
13: end for
14: Integrate Test Engine into basic software
```

integrated with the BSW on the ECU. This implies configuring the operating system on the ECU to regularly schedule the test engine, etc. This step is performed at the end of the algorithm.

C. Generation of Test Software

Once test cases have been obtained, the test engine to be flashed onto the ECU and the external test program need to be generated. In order to do this, the information about the stack objects (i.e., the output of the previous project data analysis) and the test cases are required.

The C-code for the test engine is then created and integrated as a CDD module into the BSW layer. This is then compiled together with RTE and the application software. To generate the test engine, a library consisting of a set of C-code templates is used, like displayed in Fig. 5. Examples for these C-code templates are if-then clauses, while-loops or function headers. These templates are filled with data from the test cases and the stack objects. The generated code stimulates BSW modules with application-specific values and checks the result – the output of the BSW module. The generation of the control/configuration file for the external test program is performed in a similar manner, i.e., using templates. A template here features a control sequence's skeleton which is filled in with test and communication data. This configuration might have the format of an XML file like in our case, but also other formats are possible. It contains configuration parameters for the communication between ECU and external test program, and specifies how test results should be treated and presented to the user. Clearly, due to the close interaction between test engine on the ECU and external test program, the synchronization between them should be paid particular attention.

D. Timing Analysis

In the previous sections, we discussed the test case generation for testing functionality of software modules. In the current section, we analyze the technique for measuring timing of software components in AUTOSAR.

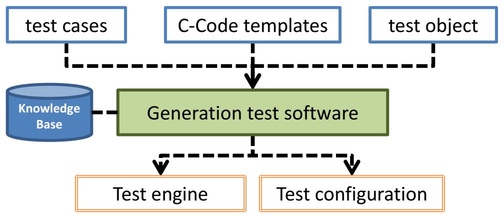


Fig. 5. Workflow of test module generation

Since AUTOSAR defines three main software layers, the timing of the whole ECU can be defined by the following equation – illustrated in Fig. 3.

$$t = t_{APP} + t_{RTE} + t_{BSW}. \qquad (1)$$

In words, the execution time of automotive software can be divided into three: t_{APP}, t_{RTE} and t_{BSW}. t_{APP} stands for the execution time of the application software, t_{RTE} is the execution time of the RTE and t_{BSW} is the time elapsed in the BSW layer. The timing properties t_{APP} and t_{RTE} can be easily obtained in an intra-ECU communication setting where $t_{BSW} = 0$ – see again Fig. 3. These two times will expectedly remain the same in an inter-ECU setting. In this latter case, t_{BSW} will be non-zero and needs to be determined. The time t_{BSW} is in turn composed of the times elapsed in the BSW stack, which again consist of a number of software modules.

$$t_{BSW} = t_{Mod1} + t_{Mod2} + \cdots + t_{Modn}. \qquad (2)$$

In the case, the application is sending a message over CAN, $t_{BSW} = t_{CANSt}$, i.e., the time consumed by the CAN stack.

$$t_{CANSt} = t_{CAN} + t_{CANif} + t_{PduR} + t_{Com}. \qquad (3)$$

Similarly to the functional tests, timing is measured in a successive manner. For example, in the case of the CAN stack, the time t_{CAN} is measured by stimulating the CAN module (i.e., the CAN driver) with different input values and computing an average execution time. Second, t_{CANif} is computed and so on until the whole chain of execution times is determined.

E. Test Procedure

For each test case, a control sequence is created. The control sequence serves the communication between test engine and external test program as shown in Fig. 6.

The external test program initiates the test sequence by sending a start command to the test engine on the ECU and waiting for acknowledgment. This way, the test engine can initialize all necessary parameters for the test. After receiving the acknowledgment, the external test program sends test data to the test engine and waits for the result. For example, this could be the checking whether a module of the CAN stack or even the complete CAN stack forwards a set of signals in the right way. The waiting for the results is connected to a timeout (set by at test case generation). If the timeout is passed, this current test step is considered as failed.

Once all test steps have been finished, the external test program signalizes the end of the test sequence and awaits the test engine's acknowledgment to finish the test procedure. After receiving all test results, the external test execution program visualizes the test results as a report.

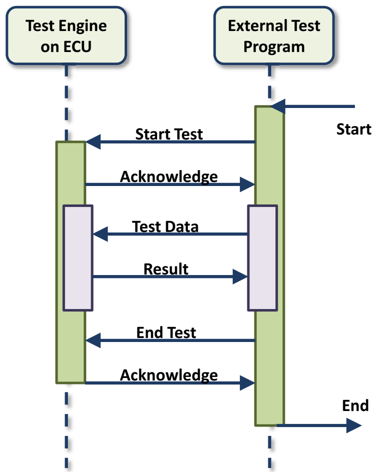


Fig. 6. Test procedure between external test program and test engine (CDD module)

F. *AUTOSAR tool chain*

Important selection criteria in choosing tools for implementing our test method are tool interoperability and the highest possible degree of automation. For the automated creation of software components and the CDD module needed by our approach, the tool SystemDesk from the company dSpace is used [17] as displayed in Fig. 7. SystemDesk is controlled by a Python Script. The resulting configuration and CDD module is used to pre-configure tresos Studio from the company Elektrobit [18]. We use this tool to configure the Basic Software EB AutoCore from Elektrobit and generating the Runtime Environment. The tresos Studio offers no automation capability; however, user interventions can be reduced to a minimum by pre-configuration. For the currently used target platform, the STM SPC560 [19], a compiler from the company Wind River [20] is used. The need for the concept of an external test program is covered by the tool ECU-TEST from the company TraceTronic [21]. This tool sends the control messages to the test engine, waits for the answer and evaluates the results. Note that our concept might be implemented with another AUTOSAR tool chain as well.

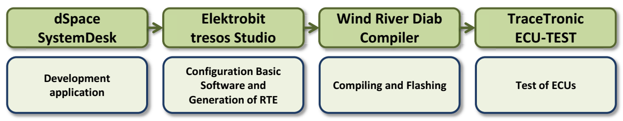
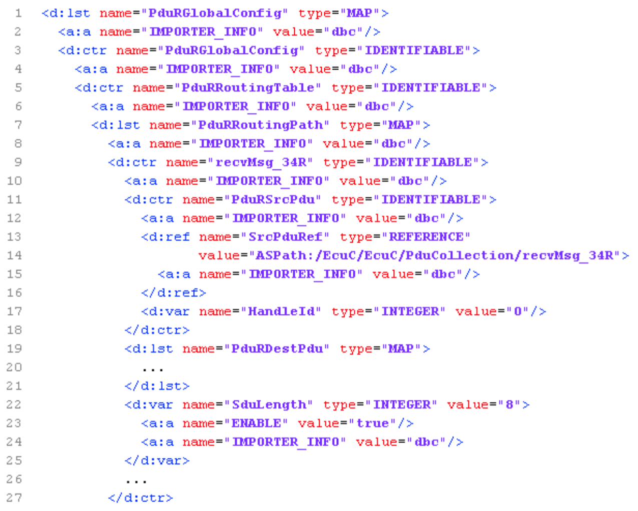


Fig. 7. Tool chain for AUTOSAR 3.1

V. Case Study

In this section, we present a case study to illustrate our test approach in the context of integrating an application software component that sends messages over CAN. The target ECU is an STM SPC560 [19], which is typically used for body and comfort applications in the automotive domain.

In Fig. 2 an overview about the whole concept is shown. The figure displays the steps from the analysis of the AUTOSAR project data, the generation of test cases to the generation of test software. The execution of the test is illustrated by an ECU with the integrated test engine. The ECU is connected with the external test execution program via CAN bus. In the following sections the applied test methods for functionality and timing are explained, and the resulting test cases are shown.



```
 1  <d:lst name="PduRGlobalConfig" type="MAP">
 2    <a:a name="IMPORTER_INFO" value="dbc"/>
 3    <d:ctr name="PduRGlobalConfig" type="IDENTIFIABLE">
 4      <a:a name="IMPORTER_INFO" value="dbc"/>
 5      <d:ctr name="PduRRoutingTable" type="IDENTIFIABLE">
 6        <a:a name="IMPORTER_INFO" value="dbc"/>
 7        <d:lst name="PduRRoutingPath" type="MAP">
 8          <a:a name="IMPORTER_INFO" value="dbc"/>
 9          <d:ctr name="recvMsg_34R" type="IDENTIFIABLE">
10            <a:a name="IMPORTER_INFO" value="dbc"/>
11            <d:ctr name="PduRSrcPdu" type="IDENTIFIABLE">
12              <a:a name="IMPORTER_INFO" value="dbc"/>
13              <d:ref name="SrcPduRef" type="REFERENCE"
14                     value="ASPath:/EcuC/EcuC/PduCollection/recvMsg_34R">
15                <a:a name="IMPORTER_INFO" value="dbc"/>
16              </d:ref>
17              <d:var name="HandleId" type="INTEGER" value="0"/>
18            </d:ctr>
19            <d:lst name="PduRDestPdu" type="MAP">
20              ...
21            </d:lst>
22            <d:var name="SduLength" type="INTEGER" value="8">
23              <a:a name="ENABLE" value="true"/>
24              <a:a name="IMPORTER_INFO" value="dbc"/>
25            </d:var>
26            ...
27          </d:ctr>
```

Fig. 8. Example of an AUTOSAR XML configuration file

A. *Analyzing AUTOSAR configuration files*

Fig. 8 shows the configuration file of the PduR (Packet Data Unit Router) module, which is one of the modules in the CAN stack. In the case of PduR, the parser looks for the *PduRGlobalConfig* container, shown in line 1 of Fig. 8. This container has all information needed by the API function call used to test functionality of the PduR module.

Pdur_CanIfRxIndication() is the API function call used to test PduR. Note that this function is specified by the AUTOSAR standard and allows us to test software in a hardware-independent manner. Usually this API function is called by the CanIf module, when a CAN message is received. It needs two parameters: the handle ID of the message and a pointer to the payload. The idea is to call this function from our test engine by setting the payload pointer to test data.

To extract the information required by the API function call, the parser searches for the corresponding message container in the XML file. In Fig. 8 line 9, such a message container is shown. It includes a sub-container called *PduRSrcPdu*, which has a child called *HandleID*. In this, the parser finds the required handle ID to test PduR.

Now, to get the necessary data length information, the XML file can be searched again. Besides the *PduRSrcPdu* container, there is a container called *SduLength*, which holds the data length information. After finding all necessary parameters, the procedure to test PduR will be generated – again using the API function call *Pdur_CanIfRxIndication()* where the payload pointer is set to the generated test data.

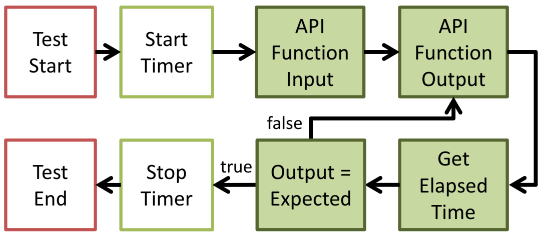


Fig. 9. Flowchart of timing analysis

B. Generating Test Data

Test data is generated on the basis of the configuration of the Com module. Information about the signal (length, type, etc.) is needed to create the test cases. During the test, when the *Pdur_CanIfRxIndication()* function returns, the result can be retrieved by the *Com_ReceiveSignal()* function call. This function has two parameters again. The first parameter is the ID of the signal to be received, and the second parameter is a pointer to a buffer, where the received data is stored. The signal ID can be found in the Com configuration within the signal container under the name *ComHandleId*.

In addition, the expected output value should be computed and stored together with the test case. In case of PduR, the expected output is equal to the input, i.e., the data sent by *Pdur_CanIfRxIndication()*.

C. Timing Analysis

To determine execution times, the AUTOSAR Gpt module is used. Gpt provides functions to start/stop timers and to retrieve elapsed times. Before each test case, a timer is started by the *Gpt_StartTimer()* function. Fig. 9 illustrates the general procedure.

Test data is then passed to the tested software module by an AUTOSAR API function call as discussed above. After the output is received from the module, the elapsed time is taken by the *Gpt_GetTimeElapsed()* function.

It is not always possible to pick the results of a module directly above or below it by making use of AUTOSAR API function calls, i.e., for some modules AUTOSAR defines no function calls that would allow this. In such cases, all modules that are passed through by signals must be tested as a cluster. In case of any communication stack, for example, the only possibility to test signals is above the Com module with help of the *Com_ReceiveSignal()* function call. If we want to measure the timing behavior of PduR, we first need to stimulate the cluster composed of PduR and Com to obtain $t_{Com} + t_{PduR}$. Then we need to stimulate Com alone to obtain t_{Com}. The time consumed at PduR is then obtained by subtracting t_{Com} from $t_{Com} + t_{PduR}$ – see Fig. 10.

D. CAN Communication Stack

The basic task of the CAN stack is the transmission and reception of messages over the CAN bus. An application can use the CAN bus to communicate with other devices.

The most important information to test software using CAN is the direction of messages (whether data is sent or

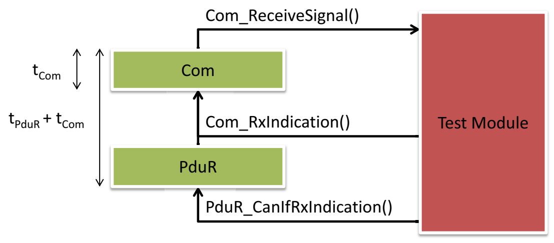


Fig. 10. Timing analysis of AUTOSAR modules

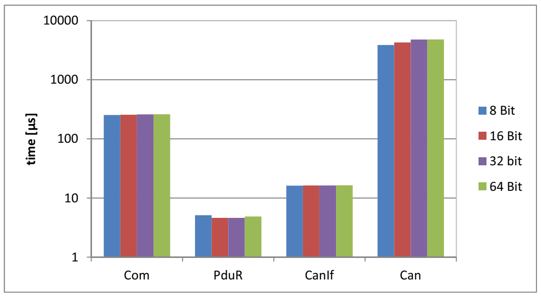


Fig. 11. Timing of CAN stack modules

received), the data type and the length in bits. According to this information test cases are generated which consider among others the limits of this data type (maximum value, minimum value), random values, consecutive values, etc.

The last stage of the proposed approach is the generation of the test engine's source code and the generation of the configuration for the external test program.

For example, let us consider the test sequence for a "Signal_1" which was extracted from the project files and can contain values between 0 and 255. First, the test sequence is initialized from the external test program, afterwards the test engine runs 3 tests (minimum, maximum and middle value of "Signal_1") and sends the test results (test successful or failed) to the external test program. This sequence is consecutively run for every stack module involved. Finally after all test cases have been executed, the external program stops the test procedure and collects all test results. These can now be used for a detailed analysis. If an error occurs, the test results allow identifying the exact module in which the error originated.

Fig. 11 shows the results of the timing analysis performed for the CAN stack on the real ECU. All modules of the CAN stack were measured for different payload lengths. In particular, for 32 and 64 bits, there is almost no timing difference on the CAN stack for the considered setting. Fig. 12 shows the timing results for the Com module alone. As it can be noticed, the Com module shows a timing variation of around $10\mu s$ when increasing the payload from 8 to 64 bits.

Fig. 13 shows the timing behavior of a wrongly configured project which sends a signal to the bus. The reason for this can be a configuration error in PduR or CanIf. A wrong configuration can make PduR route the signal to the wrong place (i.e., to the wrong bus in the system). In addition, since Com uses PduR to send signals, an error at the PduR module also leads to a timeout (t_{to}) at the Com module.

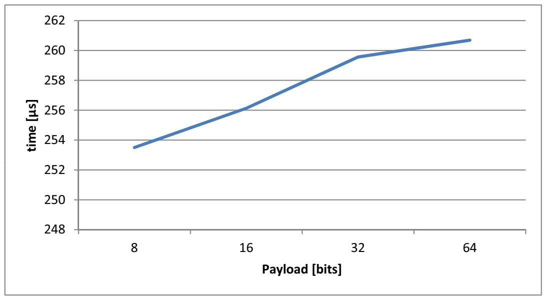


Fig. 12. Timing of the Com module

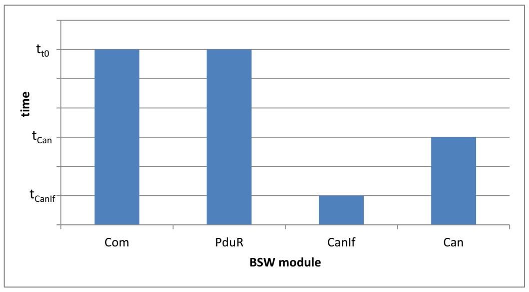


Fig. 13. Detection of configuration error in PduR

VI. Use Cases

Application-specific testing of the AUTOSAR BSW needs to be part of the development process. This way, BSW modules that are, for example, substituted by new ones can be tested towards finding configuration errors. It is further possible to benchmark BSW modules from different providers so as to find the best of such modules for a specific application.

In addition, our method supports integrating software and testing ECUs in projects where some BSW modules or application software components are bought from different providers. Each such individual component is tested and works correctly alone, but configuration problems might occur when components are integrated. In this case, the presented concept can be used to localize problems in the BSW configuration. Another use case is the migration of software between different versions of AUTOSAR or migration from one ECU to another.

VII. Concluding Remarks

In this paper we proposed a technique for testing functional and timing properties of AUTOSAR software. Our technique consists in automatically creating test cases from AUTOSAR configuration files and aims to help integrating AUTOSAR software on an ECU. Once test cases are defined taking the interfaces between the application and the AUTOSAR modules into account, a test engine in form of a complex device driver (CDD) is automatically generated and flashed onto the ECU to be tested. This then communicates with an external test program to coordinate the test sequence. Our technique does not rely on XCP and, hence, reduces the implementation and configuration overhead. In contrast to XCP, which is only available from AUTOSAR 4.0 onwards, our approach can be used for all available AUTOSAR versions. Moreover, since our approach is strictly based on AUTOSAR function calls and, hence, fully compliant with AUTOSAR, it is independent of hardware features which increases its portability.

Acknowledgments

We would like to thank reviewers for their valuable comments and suggestions towards improving this paper.

References

[1] "The official AUTOSAR website," http://www.autosar.org, Jun. 2015.

[2] "AUTOSAR: Layered software architecture," http://www.autosar.org/fileadmin/files/releases/4-0/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf, Jun. 2015.

[3] "Vector Informatik GmbH: Overview XCP," https://vector.com/portal/medien/solutions_for/xcp/Vector_XCP_Basics_EN.pdf, Jun. 2015.

[4] "Vector Informatik GmbH: Product information CANape," http://vector.com/portal/medien/cmc/info/CANape_ProductInformation_EN.pdf, Jun. 2015.

[5] "Vector Informatik GmbH: Analyzing AUTOSAR ECU software with XCP," http://vector.com/portal/medien/cmc/press/Vector/AUTOSAR_Monitoring_HanserAutomotive_SH_201111_PressArticle_EN.pdf, Jun. 2015.

[6] "AUTOSAR: Specification of timing extensions," http://www.autosar.org/fileadmin/files/releases/4-0/methodology-templates/templates/standard/AUTOSAR_TPS_TimingExtensions.pdf, Jun. 2015.

[7] G. Park, D. Ku, S. Lee, W. Won, and W. Jung, "Test methods of the AUTOSAR application software components," in *International Joint Conference (ICCAS-SICE)*, 2009.

[8] H. Moon, G. Kim, Y. Kim, S. Shin, K. Kim, and S. Im, "Automation test method for automotive embedded software based on AUTOSAR," in *International Conference on Software Engineering Advances (ICSEA)*, 2009.

[9] "AUTOSAR: AUTOSAR BSW & RTE conformance test specification part 1: Background," http://www.autosar.org/fileadmin/files/releases/4-0/conformance-testing/AUTOSAR_PD_BSWCTSpecBackground.pdf, Jun. 2015.

[10] M. Krause, O. Bringmann, A. Hergenhan, G. Tabanoglu, and W. Rosentiel, "Timing simulation of interconnected AUTOSAR software-components," in *Design, Automation & Test in Europe (DATE)*, 2007.

[11] K. Lakshmanan, G. Bhatia, and R. Rajkumar, "Integrated end-to-end timing analysis of networked AUTOSAR-compliant systems," in *Design, Automation & Test in Europe (DATE)*, 2010.

[12] K. Klobedanz, C. Kuznik, A. Thuy, and W. Mueller, "Timing modeling and analysis for AUTOSAR-based software development - a case study," in *Design, Automation & Test in Europe (DATE)*, 2010.

[13] M. Peraldi-Frati, H. Blom, D. Karlsson, and S. Kuntz, "Timing modeling with AUTOSAR," in *Design, Automation & Test in Europe (DATE)*, 2012.

[14] P. Caliebe, C. Lauer, and R. German, "Flexible integration testing of automotive ecus by combining AUTOSAR and XCP," in *Computer Applications and Industrial Electronics (ICCAIE)*, 2011.

[15] M. Deicke, W. Hardt, and M. Martinus, "Virtual validation of ECU software with hardware dependent components using an abstraction layer," in *Simulation und Test für die Automobilelektronik*, 2012.

[16] A. Spillner, T. Linz, and H. Schaefer, *Software testing foundations ; a study guide for the Certified Tester Exam - foundation level, ISTQB compliant*. Santa Barbara, USA: Rocky Nook Inc., 2011.

[17] "dSpace SystemDesk product page," http://www.dspace.com/en/inc/home/products/sw/system_architecture_software/systemdesk.cfm, Jun. 2015.

[18] "Elektrobit tresos studio product page," https://automotive.elektrobit.com/products/ecu/eb-tresos/studio/, Jun. 2015.

[19] "ST Microelectronics documentation - SPC560P50L5," http://www.st.com/web/en/catalog/tools/FM116/SC959/SS1675/PF259567, Jun. 2015.

[20] "Wind River Diab Compiler product page," http://www.windriver.com/products/development_suite/wind_river_compiler/, Jun. 2015.

[21] "TraceTronic ECU-TEST product page," http://www.tracetronic.com/products/ecu-test.html, Jun. 2015.