A Lightweight Framework for Testing Safety-critical Component-based Systems on Embedded Targets

Nermin Kajtazovic, Andrea Höller, Tobias Rauter, and Christian Kreiner

Institute for Technical Informatics, Graz University of Technology, Inffeldgasse 16, Graz, Austria
{nermin.kajtazovic,tobias.rauter,andrea.hoeller,christian.kreiner}@tugraz.at

Abstract. Rigorous development and quality assurance are inherent parts in the engineering of safety-critical systems. Many standards that address the development and certification of these systems provide a collection of various types of tests that have to be conducted to achieve the desired level of quality. Further, they recommend to perform most of these tests on the target embedded system, rather than on development hosts for example. For specific architectures, such as those used in component-based systems, this requirement is often difficult to achieve, mostly because of lack of available test frameworks that can support such specific architectures.
In this paper, we propose a framework for testing component-based systems on their embedded targets. The test framework allows to deploy software components in their binary form onto such targets. Further, it allows to build compositions out of deployed components so that complete applications can be tested. The compositions are build using the techniques for webservice composition, since interfaces to deployed software components are exposed as webservices. With this lightweight framework, it is possible to conduct some relevant tests required by the safety standards in early development phase, because it only requires software components to be implemented to test complete applications.

Keywords: safety-critical embedded systems; component-based systems; software testing

1 Introduction

Safety-critical systems can cause serious consequences such as harm on humans or equipment and environmental damages, if they malfunction. A rigorous development and quality assurance are therefore required to reduce the risk of such malfunctioning. To this end, standards for functional safety such as IEC 61508 and ISO26262 provide guidelines and methods on how to reduce the risk of failures and how to evaluate the quality of systems [8]. One set of these methods are

tests that have to be performed in order to provide an evidence about the operational profile of the system i.e., to show the conformance with the functional, safety, and other non-functional requirements. These tests are usually aligned with the well-known V development model, which comprises tests on different levels, i.e., from tests on module/unit level, integration on software and system level, to the final systems validation. One important aspect of this test chain within a V-model is that the evidence provided in reports has to conform to the real context in which the system shall operate, i.e., the safety standards require to perform tests on the real target hardware and considering the real environmental conditions [2], [8]. In many cases, this is difficult to realize, because of a variety of used target processors, used systems and software architecture and also because of a lack of specific test platforms for embedded systems. Especially, for component-based systems, which have separated development for the (software) components and for the system[1], this can be a tedious task. Many features have to be prepared in order to reach the point where software integration can be tested. For example, communication mechanisms, middleware for coordination of components and adequate interfaces to the development host are required to just test the integration between components, i.e., their composition.

In this paper, we describe a lightweight framework to perform tests of software components and their compositions on an embedded target. The distinguishing advantage of our approach is that only software components have to be provided to perform such tests. This allows developers to perform certain types of tests on complete component-based applications in the early development phases, i.e., before any middleware service for the coordination and communication of software components is developed. The framework utilizes the technique for webservice composition in order to build applications out of such components. To allow for building compositions, software components are deployed within an embedded target as standalone webservices. One of the major contributions in this paper is the mapping between specific component technology and webservices. In the end of paper, we discuss the applicability of the framework to different component technologies used in the industry.

Section 2 provides a brief overview of studies related to testing component-based systems on embedded targets. Section 3 summarizes a motivation behind the work. The proposed framework is described in Section 4, and its concrete implementation and used tools are described in Section 5. A brief discussion and concluding remarks are given in Sections 6 and 7 respectively.

2 Related Work

Now we turn to a brief overview of related studies. We outline here some relevant articles that describe test frameworks for component-based embedded systems.

Currently, the introduced problem of testing software components and their compositions on embedded targets is very important topic for automotive systems. In the last decade, several frameworks have been developed to support

[1] In the context of Component-based Software Engineering (CBSE) [1]

rapid prototyping, simulation and testing of automotive systems which implement a complex component-based architecture – AUTOSAR [3], [6], [9]. Among these frameworks, the DaVinci Component Tester [3] is the one with most features to test complete component-based systems. It is an emulated environment, with unit testing facilities for atomic and composite AUTOSAR software components. An emulated runtime environment (RTE) of the framework implements basic communication and coordination services for software components so that all necessary interaction scenarios between those components can be emulated. For the purpose of testing, the components have to be compiled for the target where the RTE is operating. Usually, the development host is used to execute RTE, rather than an embedded target. Similar to DaVinci Component Tester, the framework in [6] provides test support for compositions. In contrast to previous work, fault injection tests are applied here to evaluate the reliability of AUTOSAR systems. Finally, the Artop ARUnit framework [9] provides AUTOSAR RTE services to perform unit testing for single software components.

In general, described frameworks perform the testing on development host only. Therefore, for systems qualification, safety-relevant software components and related compositions need to be tested again on their real embedded target. As mentioned in the previous section, to realize this deployment and a support for building compositions such as RTE of the DaVinci Component Tester for embedded targets may require much effort. One option to overcome this issue is to use the standardized techniques for the deployment and composition, such as those provided by service-oriented architectures (SOA) for example. Currently, there are many approaches that use SOA to expose embedded devices with the purpose of testing [7] or integrating devices to implement certain business processes [4]. Another advantage of using SOA for this purposes is that many mature tools and methods exist that provide various generation facilities or test frameworks. Similar to work in [4], we use SOA to build compositions, but instead of exposing device functions as webservices, we expose the deployed software components. Thus, for every software component, there is a single container, a server, which provides a webservice interface to its function.

3 Motivation

Mastering complexity of today's embedded systems is one of the major challenges in safety engineering. In addition to rapid increase of software complexity, many application fields are confronted with the issues coming from the concurrent engineering, where different organizations are contributing to systems development. In the automotive industry for example, many system parts are delivered by the suppliers, including devices, OS services and libraries, while the automotive companies, i.e. manufacturers, are focusing on software applications. Developing such applications is challenging for manufacturers, because for testing purposes they need a system in which all required parts from suppliers are integrated.

Providing a test support for the application-level software without a need to consider supplier's parts would leverage rapid development for the manufactur-

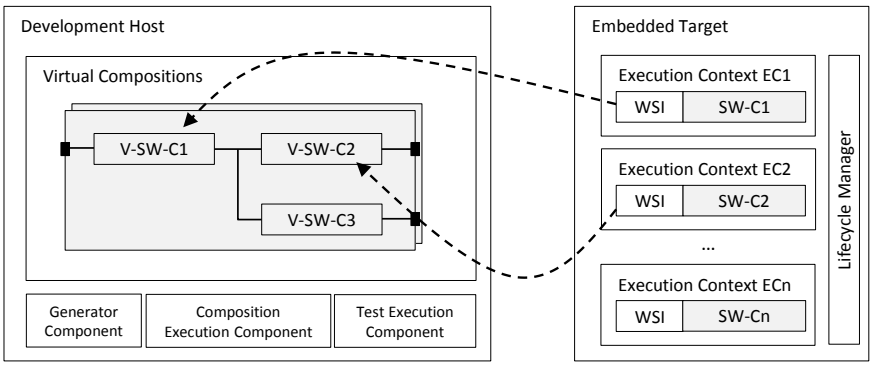

Fig. 1. Test framework architecture: a Development Host – modelling component-based system, definition and execution of tests (left) and an Embedded Target – execution of software components (right)

ers, and would allow them to achieve many test objectives earlier. In summary, the manufacturers would be able to (i) qualify the application-level software components according to regulations of safety standards, and (ii) to early conduct the functional tests, which can be used later in the verification and validation part of the V-model to complement the remaining test activities.

4 Proposed Framework

In this section, we introduce the proposed test framework. We first give an overview of its main components, and then we describe how the test process is conducted using the framework.

Fig. 1 shows the simplified architecture of the framework. Basically, the framework consists of the two main components: the Development Host and the Embedded Target, which has to be used in the operation of the safety-critical system. This Embedded Target provides services to deploy software components and to manage their lifecycle during the tests. The Lifecycle Manager component shown in Fig. 1 is responsible for these purposes. One of the main characteristics of the framework is that software components are executed in isolation and do not interfere with each other within the Embedded Target. That means, the interaction between those components is not possible within the Embedded Target. Thus, software components can just execute their functions, based on input data provided by the Execution Context component, which also collects the results from that component after the execution. The Execution Context corresponds to a simple middleware or a container that implements the lifecycle management for a single software component. To communicate with the rest of the framework, it exposes the interfaces of its software component as a webservice (the Webservice Interface WSI in figure).

On the other side, the Development Host comprises a collection of various tools in order to (i) to map a particular component technology on webservices, i.e., to build the Execution Context (see Section 5 for more details), (ii) to build a component-based application by composing software components running on the Execution Context, and (iii) to conduct the test process on those applications.

4.1 Framework Components

Virtual Composition corresponds to a modelled component-based application. In its simplest form, it corresponds to a concrete software component, which runs on the Embedded Target. Therefore, for every software component, there is a corresponding Virtual Composition which runs on the Development Host. From the technical viewpoint, a Virtual Composition is a webservice instance which points to a concrete software component on the Embedded Target. In a more complex form, the Virtual Composition can comprise multiple software components, i.e., a composite of multiple Virtual Compositions, so that complete component-based applications can be modelled. For this purpose, a technique for webservice composition is used (see Section 5 for more details).

Composition Execution Component. This component of the framework executes Virtual Compositions. Based on their descriptions, it simulates the application behavior while executing the involved components on the Embedded Target.

Test Execution Component is a bundle consisting of the test cases, stubs and drivers to conduct the complete test on modelled Virtual Compositions. It executes Virtual Compositions against provided test cases and reports the test status.

Execution Context. The Execution Context is a standalone component container that provides the lifecycle management for a single component. Furthermore, it exposes the component interfaces through webservices (WSI) in order to allow to build Virtual Compositions. It also provides the state isolation functionality for stateful components, which is relevant if a particular component is used multiple times by the Virtual Composition (see Section 4.2). Fig. 2 shows the architecture of Execution Context. In short, a software component is wrapped by the webservice, which is generated and linked with the standalone SOAP[2] server that hosts the service. The code on the left in figure shows the concrete method body of the webservice and its link with the concrete software component. We explain this link in Section 5 in more detail.

Generator Component is a part of the deployment process. It comprises a collection of tools (i) to generate the required artifacts for the modelled Virtual

[2] Simple Object Access Protocol: application-layer protocol for webservices.

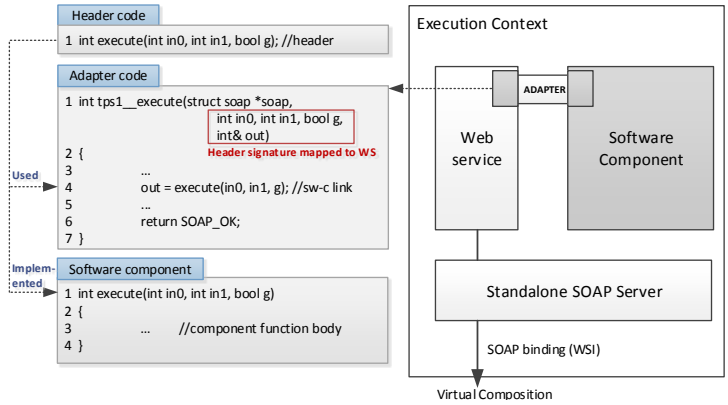

Fig. 2. Execution Context: artifacts used for the mapping between CBSE and SOA (left) and the architecture (right)

Compositions and to deploy them onto the Development Host, and (ii) to generate the Execution Context, i.e., to map CBSE on SOA, and to deploy it onto the Embedded Target.

Lifecycle Management manages the lifecycle of all Execution Contexts within the Embedded Target. It provides services for the deployment, removal and roll-back[3] of software components.

4.2 State Isolation

The presence of the stateful components is an issue if multiple instances within a single Virtual Composition exist and if several Virtual Compositions share the software components simultaneously. It is therefore necessary to protect such software components from transition into an inconsistent state, i.e., the state influenced by one Virtual Composition shall not be used or compromised by another Virtual Composition. This feature is a part of the Execution Context. It isolates the state by identifying the Virtual Composition that owns the currently active request. For this purpose, the context-dependent state is queued within the Execution Context for each stateful component. Based on the incoming request, the corresponding state is accordingly restored.

4.3 Test Workflow

After software components are developed, they are compiled for the Embedded Target and deployed there, using the Generator Component. To perform the tests on software components or on their compositions, a component-based system is modelled first, by defining Virtual Compositions and by deploying them onto the

[3] Required to reset the component state for new test iteration.

Development Host. Finally, test cases are defined and executed on the modelled Virtual Compositions, using the Test Execution Component.

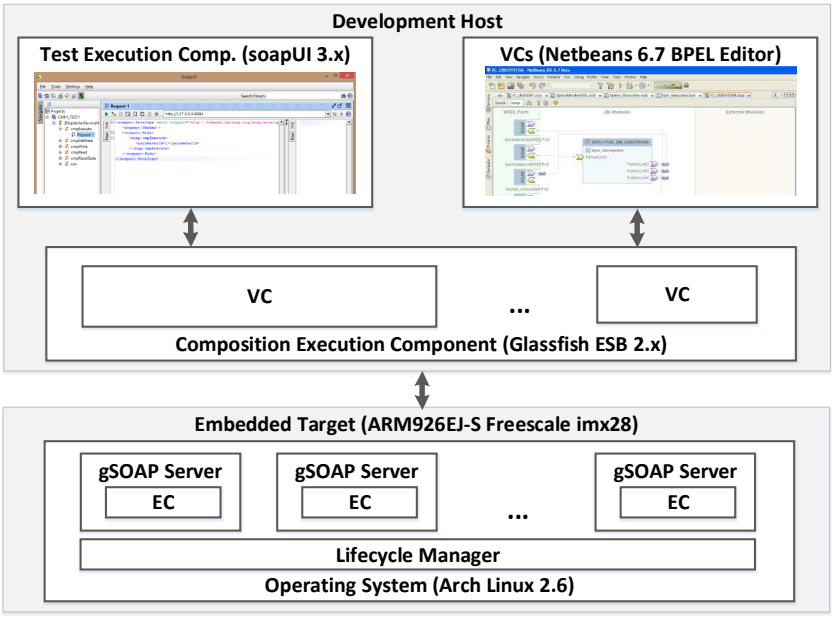

Fig. 3. Implementation of the framework and used tools and configurations (EC - Execution Context, VC - Virtual Composition)

In the following section, we introduce the concrete implementations and tools used to realize the mentioned components (see Fig. 3).

5 Implementation

In order to apply the framework in the domain of embedded systems, we use the gSOAP[4] webservice library, which offers the SOAP stack dedicated to resource-constrained systems. The gSOAP webservices host the Execution Context and the Lifecycle Manager directly on the Embedded Target. For every software component, there is a dedicated gSOAP server that hosts that particular component. All gSOAP servers are running as Unix processes within an Arch Linux operating system. In our test setup shown in Fig. 3, we deploy software components on an ARM9 target (Freescale imx28 with 454MHz, 128MB of RAM).

For modeling and executing the Virtual Compositions, we use an XML-based WS-BPEL (WS Business Process Execution Language) [5]. This technology is widely applied in enterprise applications to seamlessly integrate webservices or

[4] gSOAP Homepage: `http://gsoap2.sourceforge.net/`

legacy applications wrapped by webservices into a business process. Thus, every Virtual Composition is represented as a BPEL process and therefore consists of a workflow, which describes the sequence on how the software components have to be executed. In addition to the workflow, every Virtual Composition describes the integration between software components as a structure, i.e., in a composite diagram (see Fig. 3, Netbeans 6.7 BPEL editor).

From the front-end viewpoint, we use the soapUI Tool[5] to define the test cases and to drive the test process. This tool allows to perform various test strategies on webservices, such as functional tests, load tests and security tests. It also enables the automated test execution for given test suites. In our context, it plays the role of the Test Execution Component. We manually specify the test cases, define a test suite and execute it on the deployed Virtual Compositions.

Another tool which is used as part of the front-end is the Netbeans BPEL Editor. We used it to graphically define Virtual Compositions in terms of the structure and the workflow and to generate the necessary artifacts for the deployment, such as WSDLs and assembly descriptions for Virtual Compositions. In order to deploy Virtual Compositions as webservices on the Development Hosts, we use the Sun Glassfish Enterprise Service Bus (ESB).

5.1 Webservice Interfaces for Software Components

As illustrated in Fig. 3 webservice interfaces of the Execution Context are hosted by the gSOAP servers. Except of the SOAP stack, the gSOAP library consists of a generator toolchain, which allows to build webservice stubs and skeletons from the WSDL specifications. We use the *wsdl2h* tool to generate the header files that are in turn used to link the object code of software components with the Execution Context. The generation of the Execution Context is supported by the *soapcpp2* skeleton compiler (see Section 5.2). This is one of the most challenging parts of the framework, because here, a mapping from the used component technology to webservices is performed. An excerpt of this mapping is depicted in Fig. 2. Here, a software component is represented using just a single C/C++ method. This method is used by the Adapter, which routes the data from the Development Host to the component, and returns the results from that component. To establish this link, we define a header file, which is implemented by the software component, and which is used by the Adapter to find a proper symbol after linking the adapter with the component. Both the adapter code and the headers are generated based on interface description of a software component. In the following, we describe the process of generating artifacts used to build the Execution Context.

5.2 Deployment Process

The essential part of the deployment process in which the Execution Context is generated is depicted in Fig. 4. The process starts by submitting the software

[5] soapUI Homepage: `http://www.soapui.org/` – in this work used for specifying functional tests only.

component in form of the object code and its interface description to the Lifecycle Manager on the Embedded Target. The Lifecycle Manager in turn starts the deployment by generating the necessary skeleton code based on component interface description, i.e., it generates a header file that describes the component interface for the linking with the Execution Context, and the required libraries for the SOAP stack. In the next step, the Execution Context is generated. At this point, all artifacts for the deployment are ready. They are, in the final step, compiled and linked to a single image of the Execution Context, which is then bootstrapped by the Lifecycle Manager. The Execution Context in turn takes the control over the component lifecycle and publishes its WSI. After this last step is completed, the software component is ready for tests.

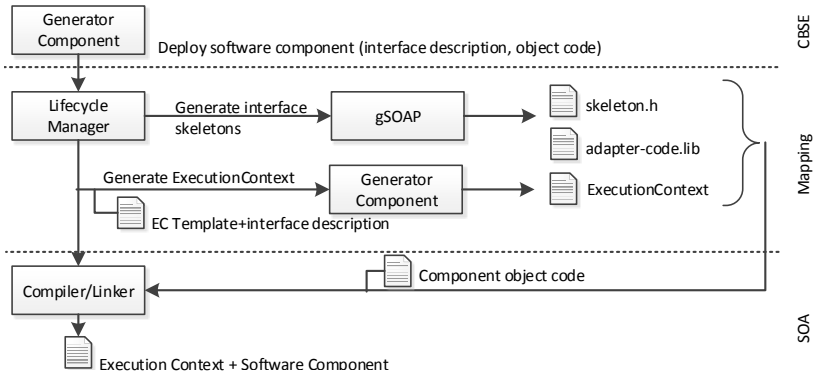

Fig. 4. Generation process for the Execution Context

6 Discussion

We showed in this paper how component-based systems can be tested on embedded targets. With the introduced framework, the functional tests can be performed on a level of software components and their compositions. Although the framework allows testing just on a functional level (compared to introduced frameworks where also middleware is part of a test), it allows to conduct the early qualification of software components on embedded targets and to test potential component-based applications.

We also described the link between SOA and CBSE, using plain C/C++ methods as component technology. To apply the framework to other component-based systems, similar adapters to SOA have to be realized. For instance, to test AUTOSAR systems, the adapters for Runnables have to be implemented. Runnables are execution units within AUTOSAR software components, and are triggered by specific events by the AUTOSAR RTE. The events have to be realized with BPEL, and AUTOSAR software component have to be realized as

Virtual Compositions. In contrast to CBSE to SOA mapping introduced here, a composition of AUTOSAR software components would be represented as a Virtual Composition that consists of further Virtual Compositions, each representing a single AUTOSAR software component. On the other side, for data-flow synchronous systems such as Matlab Simulink and IEC61131, the mapping to SOA can be realized as described in this paper. That means, in case of specific execution semantics such as request-response and sender-receiver interaction styles in AUTOSAR, additional layers of Virtual Compositions are required.

7 Conclusion

Testing safety-critical systems in their real context, i.e., on embedded targets and under real environmental conditions, is recommended by safety standards. However, there are many challenging factors to perform this task, such as variety of available target processors, lack of test frameworks for embedded systems and used specific systems architecture.

In this paper, we introduced a framework to test safety-critical component-based systems on embedded targets. With the framework, the functional tests can be performed on a level of software components and their compositions. The distinguishing advantage of our approach is that only software components have to be provided to perform such tests. This allows developers to perform functional tests on component-based applications in the early development phases.

Currently, the framework can host software components with the primitive data types on their interfaces only. As part of the ongoing work, we will provide a support for specific complex data types, to enable to host some existing component-based systems such as AUTOSAR or IEC61131 systems for example.

References

1. Crnkovic, I., Larsson, M.: Building Reliable Component-Based Software Systems. Artech House Publishers, ISBN 1-58053-327-2 (2002)
2. Grünfelder, S.: Software-Test for Embedded Systems. dpunkt.verlag (2013)
3. Informatik, V.: Davinci component tester - user manual. Tech. rep., VI GmbH (2011)
4. Karnouskos, S., Baecker, O., de Souza, L., Spiess, P.: Integration of soa-ready networked embedded devices in enterprise systems via a cross-layered web service infrastructure. In: IEEE ETFA. pp. 293–300 (Sept 2007)
5. Louridas, P.: Orchestrating Web Services with BPEL. IEEE Softw. (Mar 2008)
6. Piper, T., Winter, S., Manns, P., Suri, N.: Instrumenting autosar for dependability assessment: A guidance framework. In: 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 1–12 (June 2012)
7. Rusli, H.M., Ibrahim, S., Puteh, M.: Testing Web Services Composition: A Mapping Study. Communications of the IBIMA 2011(598357) (2011)
8. Smith, D., Simpson, K.: A Straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849. Elsevier Science (2010)
9. Wong, D., Wengler, T., Asmus, R., Rudorfer, M.: Artop: Developing autosar tools in the community. ATZextra worldwide 18(9), 34–36 (2013)