Built-in testing of embedded software systems

Irena Pavlova[1] and Aleksandar Dimov[2]

Abstract – **Embedded software systems encompass a broad range of devices, which may be mixed hardware/software system dedicated for a specific application. Usually embedded software systems are part of and manipulate and control a larger, physical system. There are a lot of unresolved issues in building reusable, reliable and predictable systems of that class. One such issue is software testing, which currently is done in an ad-hoc manner. In this paper we propose an approach for testing of embedded software systems, based on the so-called Built-In-Testing (BIT). BIT is a concept, where given software module or system has capabilities to perform testing itself.**

Keywords **– Embedded systems, Component Bases Software Development (CBSE), Built-In-Testing (BIT).**

I. Introduction

Embedded systems are often distributed real-time systems comprising electronics and software. Such systems are increasingly penetrating every aspect of our lives and work, from telecommunication systems, transport, energy and utilities, health, finance, education, tourism and environment. The embedded systems industry is competing with decreasing time to market and increasing product differentiation.

Both lead to increasing dependence on software required to be flexible enough for rapid reuse, extension and adaptation of system functions. It is often difficult to test and verify embedded systems because of the intrinsic "embedded dimension". This is an effect of that the software has to be designed on a platform different from the platform on which the application is intended to be deployed and targeted.

Embedded systems are also often mission-critical and needs to be extensively verified and testing is one of the major challenges. Compared to standard PC software embedded software is harder to observe, test, and debug.

The contribution of this paper is towards interesting and needed research areas within BIT for component-based embedded systems. With the above aim we propose a reference model for Quality of Service (QoS) BIT testing. The main target at this stage is non-functional requirements like timeliness and performance. Within this context we focus on the following main concerns:

• Increase software quality, in terms of functional and non-functional properties

• Shorten development times, in terms of the development process and specifically test reuse.

[1] Irena Pavlova is with the Faculty of Mathematics and Informatics, Sofia University, James Boucher 5, Sofia, Bulgaria E-mail: irena_pavlova@fmi.uni-sofia.bg.

[2] Aleksandar Dimov is with the Faculty of Mathematics and Informatics, Sofia University, James Boucher 5, Sofia, Bulgaria E-mail: aldi@fmi.uni-sofia.bg.

The remainder of the paper is structured as follows: In section 2 CBSE for Embedded systems is presented. Section 3 makes an overview of BIT technology. Section 4 discusses the application of BIT for components in embedded systems. Section 5 presents a reference model for QoS BIT testing of component based embedded systems. Finally, section 6 concludes the paper.

II. CBSE for Embedded Component based Systems

Assembling new software systems from existing components is an attractive alternative to traditional software engineering practices which promises well defined software architectures, reduced developments costs as well as reuse [4]. However, these benefits will only occur if separately developed components can be made to work effectively together with reasonable effort [8]. However, lengthy and costly verification and acceptance testing may impact negatively the independent component development and system integration.

This way application of new processes, approaches and instruments for supporting effective integration and reducing manual system verification effort in the context of embedded software systems is needed. This may be done by equipping components with the ability to check their execution environments at runtime. Built-in-test (BIT) is such an instrument, providing a model for elaboration of detailed tests while developing the component.

One of the major driving-forces behind component-based development is reuse; however, in many companies reuse has not been very successful even though component-based development has been introduced in the software lifecycle. It is often required to restructure the organization to reflect the component based process, i.e., divide component development from system development. Another major obstacle for reuse is efficient administration (e.g., version and configuration management) with growing component repositories [5].

Most embedded systems have requirements not present in other systems, e.g., timeliness, low footprint, low energy consumption, etc.

Such non-functional requirements need to be verified and validated, adding another dimension of testing to the system. Hence, it is essential to satisfy not only the functional behavior, but also extra-functional properties such as, e.g., timing and dependability attributes. These systems characteristics usually implies that embedded systems are statically configured, i.e., the components used and their interconnections are decided at design or configuration time. Here, the binding is static, as opposed to the dynamic binding used in most desktop component technologies.

Furthermore, embedded systems are resource constrained in the sense that the per-unit cost is a main optimization criterion, i.e., the use of computer and computing resources should be kept at a minimum. Also, due to the high variability of many embedded systems, it is common within the embedded systems industry to use product-line architectures. Because of this, reuse of architectures, components, quality assessments and tests are very attractive for reducing development costs.

III. Built-in-Testing overview

Testing is a disciplined process that consists of evaluating the application (including its components) behavior, performance, and robustness – usually against expected criteria. One of the main criteria, although usually implicit, is to be as defect-free as possible.

Expected behavior, performance, and robustness should therefore be both formally described and measurable.

Compatibility of components is one of the greatest issues. It is not of much use to specify a component as part of large and complex software system if it will not deliver what has been promised. One of the key ways of addressing this issue is to build components that are self-testing, to ensure that they meet the specifications for that part of the total application.

BIT [10] proposes to build test-mechanisms into components and systems during design and coding, so that the successive testing and maintenance processes can be simplified. The most interesting feature of the BIT is that tests can be inherited and reused in the same way as that of code in the conventional COTS components [7].

Built-in testing of software components can be done in a large number of ways. The Component+ project [1] developed a methodology for integrating BIT components into COTS software, using methods that are a significant extension of the object-oriented technology. The design principles of BIT for software components embrace two major perspectives: Contract testing - to verify a contract between two components from both parties point of view. Quality of service testing - to verify that the operating environment of a software component continues to give the right service, that the interaction between all components works and that residual faults in a component prevent proper function of the component or the system.

The BIT architecture is based on the following elements:

- **BIT-component**: component that provides a number of built-in test services and test interfaces, as shown on Fig. 1.

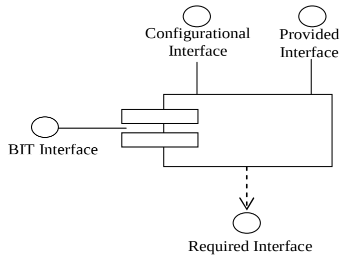

Fig. 1. BIT Component

- **Testers**: components that use the test services of BIT-components to determine whether a system-level error condition exists.
- **Handlers**: components that handle errors detected by BIT components or test components.
- **System constructor**: a conceptual element, nominally responsible for the instantiation of (high level) BIT-components, testers, and handlers, and their interconnection. Note that both BIT components and testers can detect error conditions. The BIT component can detect internal (i.e. component-level) errors, whilst the testers detect external or system-level errors arising from incorrect component interaction.

IV. Built-in-Testing in the context of component-based embedded systems

With traditional development approaches, the bulk of the integration work is performed in the development environment, giving engineers an opportunity to pre-check compatibility of system various parts, and to ensure that the overall deployed application is working correctly. In contrast, late integration implied by component assembly means that there is little opportunity to verify the correct operation of applications before deployment time.

Although component developers may adopt rigorous test methodologies, with non-trivial software components it is impossible to be certain that there are no residual defects in the code - formal proof or 100% test coverage are not viable options in most practical cases. Compilers and configuration tools can help to some extent by verifying the syntactic compatibility of interconnected components, but they cannot check that individual components are functioning correctly (i.e. that they are semantically correct), or that they have been assembled together into meaningful configurations (i.e. systems). As a result, components that may have behaved correctly in the sanitary condition of the development-time testing environment, may not behave so well when deployed in a system where they have to compete with other (third party) components for resources such as memory, processor cycles and peripherals.

Sophisticated verification methods are used to increase the level of assurance of critical software, particularly that of safety-critical and mission-critical software. Embedded software verification is a systems engineering discipline that evaluates software in a systems context [9].

In order to bring the effectiveness of verification to bear within a reuse-based software development process it must be incorporated within the domain engineering process. Failure to incorporate verification within domain engineering will result in higher development and maintenance costs due to losing the opportunity to discover problems in early stages of development. The component Verification, Validation and Certification Working Group at WISR 8 found four general considerations that should be used in determining the level of verification of reusable components [6]:

- Span of application - the number of components of systems that depend on the component

- Criticality – potential impact due to a fault in the component
- Marketability – degree to which a component would be more likely to be reused by a third party
- Lifetime – duration of component usage.

Although encapsulation and information hiding are central principles for facilitating the design and development of component based software systems, their very nature also complicate the task of testing. This is because some of the information that is necessary for comprehensive testing of objects and components is by definition hidden to entities outside a component (e.g. the test software).

Many software components are state machines and the state information is hidden. Encapsulation and information hiding thus give rise to a couple of fundamental problems inherited in conventional software components technologies, which have yet to be addressed in CBSE:

- Low testability.
- Low maintainability for CBSE actors.
- No support for run-time testing.

These problems hold even when components are supplied in their complete form, i.e. with the source code. Software is seldom so well documented that a user unacquainted with a component can verify it in an easy way. Test software delivered with the component increases the testability.

A piece of software with encapsulated state information is testable if we can:

- Set it into a given state before a test.
- Stimulate it with given test data.
- Read the response and the resulting state.
- Compare the actual outcome of the test with expected outcome.[1]

To make such software component testable it should be able to get access to the encapsulated state information of the component before a test is invoked. This holds for tests of behaviour. For other kinds of testing the test software must have access to other internals of the component. Hence part of the testing software has to be built-in.

Besides the testing challenges of standard functional testing of component-based systems, embedded systems have a range of extra-functional properties that also need to be verified. Some of the important attributes for embedded systems that define quality, besides correct functional behavior, are [2, 3]:

- **Real-time properties** – violation of time requirements, despite correct functional behavior, violates the system behaviour.
- **Dependability** – the ability of the system to deliver a service that can be trusted.
- **Resource consumption** – Many embedded systems have strong requirements on low and controlled consumption of different resources.

Besides quality aspects, an important issue for the embedded systems segment is time-to-market. Component-based development has shown to be an efficient paradigm for increasing productivity and lowering development time and costs. However, component based development for embedded systems has not been as successful as for, e.g., desktop systems, especially not considering reuse.

One of the major reasons to this is the lack of support for configuration and version management.

Thus, the perhaps most important aspects for reducing development time and time-to-market are:

- **Reuse** is a basic concept in CBSE that decreases development time and time-to-market.
- **Software configuration management** – important for embedded systems in the context of reusability.
- **Verification** - To find errors in the code, and hopefully at an early stage indisputably shortens the time for testing, redesign etc.

V. Built-in QoS Testing Model for Component-Based Embedded Systems

Within a real (as opposed to a test) system, a component competes with other components for resources such as memory, processor cycles and peripherals. Consequently, its performance may be affected by the system in which it is integrated. This is particularly critical in real-time systems where a component may have deadlines to meet or a certain throughput to achieve. Adequate system performance should be designed into the system, but this requires components to be characterized in terms of the resources they require as well as their functional and dynamic behavior. This is not usually done, so system performance has to be measured during development and deployment. Therefore, a requirement exists for QoS testing to support verification of components dynamic behavior.

Timing and performance testing is an indisputable part of each testing effort. The strict requirements towards embedded systems as well as the utilization of external resources (components) increase the importance of testing the timing as well as the performance of the components when integrated into assemblies.

The reference architectural model we propose for Timing BIT QoS testing is illustrated on Fig 2. The Timing tester is intended to measure the time spend for data access for a particular component scenario. It is important the test for every component to be performed in a single transaction. Single transactions are used also with the purpose any time dependencies to be avoided. Time measurement will start before the starting of the transaction and will end after the end of the transaction.

Fault situations are handled by a *Handler* component. For example such situation may occur if incorrect or impossible attempt to access the data in the database is made.

The Component under test should support *IBITTiming* interface, which is used to perform a timing test on a single component. This interface allows subscribing or unsubscribing for time event for the particular component.

Timing tester provides *IBITTimingNotify* Interface that is used for notification for time events and requires *IBITError* interface. This interface is used by the Timing Handler, and provides only one function, which requests the handler to process the thrown Timing exception. The processing of the exceptions includes logging the exception in file and other user defined functions. The *IBITErrorNotify* interface is required on the Handler for reporting errors.

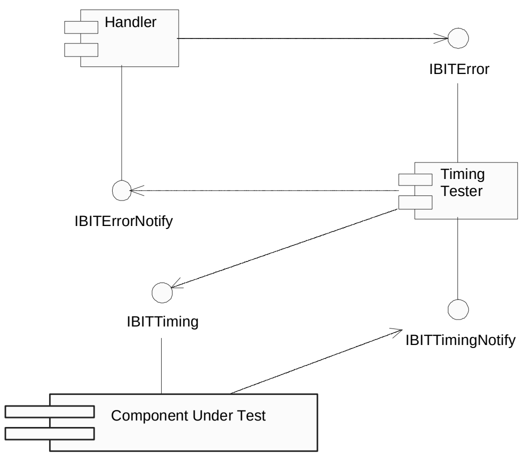

Fig. 2. BIT Timing Testing Reference Architecture

For realizing performance testing that is presented on Fig. 3, a Performance tester is developed "on the top" of the Timing tester. The main idea is to use several time tests on different components. This tester uses a single transaction for every set of time tests. Fault situations are handled by a relevant Handler component. This component collects information about time spent for a particular scenario involving several different components.

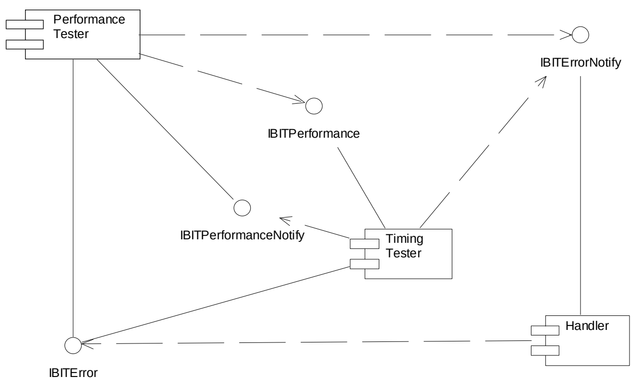

Fig. 3. BIT Performance Testing Reference Architecture

The Timing tester component must support IBITPerformance interface. This interface is used when performing the testing. The IBITPerformance interface allows for subscribing to a particular set of components, which will be monitored during the performance test execution. Performance tester provides IBITPerformanceNotify Interface that is used for notification of events. Using this interface performance tester will receive information about time spent for a method or information that there is an error during the execution of the method.

VI. CONCLUSION

Real-time aspect is one of the major differences between embedded software and PC or internet software. Embedded software systems often interact and controls physical processes with real-time requirements. Timing is often of first priority in testing efforts. To use BIT in embedded real-time systems it is important to understand the relation between the two.

Testing worst-case response-time is not trivial. Typically, tests assess within what time the embedded system reacts (creates an output considering an input). The assessed time forms an end-to-end latency for the response. Internally, the system typically involves transactions of several execution threads that must cooperate to create the correct output. These threads can in turn experience interference from other parallel activities in the system. Thus, the goal for the test is to measure the time from a certain change in the input until a certain output is produced under maximum disturbance from other parallel activities. These techniques not only have to set up a worst-case scenario (which is challenging in itself), but also have to measure the system non-intrusively, i.e., make sure that the measurement does not affect the test (probe-effects).

ACKNOWLEDGEMENT

The work presented in this paper was partially supported by the National Science Fund, part of the Ministry of Education and Science in Bulgaria under the ADEESS (MU01-143) and TASSA (ДО-02-182/16-12-2008) projects.

REFERENCES

[1] Component+ EU FP5 Project, 2006.
[2] I. Crnkovic. Component-based approach for embedded systems. In Ninth International Workshop on Component-Oriented Programming, Oslo, June 2004.
[3] I. Crnkovic. Component-based software engineering for embedded systems. In International Conference on Software engineering, ICSE'05, St. Luis, USA, May 2005. ACM.
[4] I. Crnkovic and M. Larsson. Building Reliable Component-Based Software Systems. ISBN 1-58053-327-2. Artech House, 2002.
[5] I. Crnkovic, S. Larsson, and M. Chaudron. Component-based development process and component lifecycle. In 27th International Conference Information Technology Interfaces (ITI), Cavtat, Croatia, June 2005. IEEE.
[6] S. H. Edwards and B. W. Wiede. Software engineering notes, 22,5,17-31. WISR8L 8th Annual workshop on SW Reuse, 1997.
[7] K. J. Fernandez, V. H. Raja, and M. Morley. A system level testing modeling mechanism in a reengineering environment. Journal of Conceptual Modeling, issue 18, 2001.
[8] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In Proceedings of the Seventeenth International Conference on Software Engineering, April 1995.
[9] D. Wallace and R. Fujii. Software verification and validation: an overview. IEEE Software, 6(3):10–17, May 1989.
[10] Y.Wang, G. King, D. Patel, S. Patel, and A. Dorling. On coping with real-time software dynamic inconsistency by built-in tests. Annals of Software Engineering, 7(1):283–296, Oxford, 1999