Electronic version of an article published in **Net.ObjectDays (Hrsg) Tagungsband Net.ObjectDays 2005, pp. 213-226**

http://www.old.netobjectdays.org/node05/de/conf/proceedings.html

Testing Mobile Component Based Systems

Dima Suliman[1], Barbara Paech[1], and Lars Borner[1]

[1]University of Heidelberg
Institute of Computer Science
Software Engineering
Im Neuenheimer Feld 326
D-69120 Heidelberg, Germany
{suliman, paech, borner}@informatik.uni-**heidelberg**.de
http://www.informatik.uni-heidelberg.de

Abstract. Using components in building software systems is a popular approach in software engineering. Testing component based systems (CBS), however is still a challenging task. Additional problems arise, if the tested CBS is a mobile system. Mobile systems offer more challenges, because their structure is dynamic: Components may be added or removed at any time. Thus, testing is required at run time. In this paper we present criteria that a testing approach of mobile CBS has to satisfy. We investigate to what extent common CBS testing approaches fulfill these criteria: For each approach we explore how the particular test steps (test cases, test inputs and test oracles) are created or generated. Although many approaches introduced in the literature assume that testing is needed only at deployment time, we found many interesting features that can be applied in testing mobile CBS, but no existing approach can be taken as it is.

Keywords: testing mobile systems, component testing, mobile systems, component based systems

1 Introduction

Building software systems from components can save time and effort. One of the domains that use this approach is the domain of mobile systems. A mobile system consists of mobile devices. On each mobile device many components may be installed. Each component belongs to one mobile device. When testing mobile component based systems (mobile CBS) two questions are important:
1. Do we have a *good connection* to a component?
2. Do we have a connection to a *good component*? In other words: Does the component perform as expected?
This paper deals with the answer to the second question, whether the participating components "share" a mutual understanding of their functionality.
In general, testing of CBS is difficult, because the user (the component buyer) has limited information about the components. For example, in most cases the source

code of the components is not delivered. And even if the source code is available, it is very difficult and very time-consuming for the user to analyze and understand the code, and to create test cases. Testing mobile CBS is more challenging, since mobile systems have special requirements (e.g. the computing and storage capacity of mobile devices are limited, the environment is steadily changing). In this paper we investigate the issue of testing mobile systems, because mobile systems will play an essential role in modern technology. Thus, developing testing approaches for mobile CBS is not only an interesting problem; it is a major demand. This paper is structured as follows: in section 2 the criteria for mobile test approaches are explained, in section 3 popular CBS testing approaches are examined. And in section 4 the features transferable from CBS testing approaches to mobile CBS testing approaches are discussed. Finally a conclusion is presented in section 5.

2 Criteria for test approaches for mobile CBS

A testing approach for mobile CBS has to overcome difficulties that a *general* CBS testing approach does not face. In a *general* CBS the testing approach is usually needed only at deployment time: At deployment time the *system builder* uses the test approach to check whether the components can *work* with each other, and whether they *perform* as expected. If this is the case, the software system can be released. Mobile CBS complicate the situation, because the systems are built dynamically. Thus, the participating components are not known in advance. Strictly spoken there is no deployment time, since at any time components may be added or removed from the system. Thus, testing is required at run time. In a mobile CBS the components are running on mobile devices, and mobile devices have special properties that have to be considered while testing. In particular, their limited resources play an essential role. In this section the challenges that a mobile testing approach has to overcome, are explained. We justify our choice of criteria with a general scenario:
Suppose a component A with functionality f is available on a mobile device Z. Another component B (possibly on another mobile device) needs this functionality. Before functionality f is used, B wants to test, whether f fulfils its expectations – analogously to a human tester who would like to test f before it is used. Now, which challenges must the run time testing approach overcome, and which criteria must be fulfilled?
The first criterion is inherent in the concept of run time testing: In contrast to testing at deployment time, testing at run time needs to run as much as possible without human intervention. Involving a human user of a mobile device (on which the component is running) in the testing approach is not practical. For example, it is unrealistic to assume that the human user of the mobile device (on which component B is running) can create test cases for the needed functionality f of component A, since this requires professional knowledge:

C.1 The test cases of a component should be either:

a. available at run time,

b. or they can be generated at run time.

Other test activities (e.g. test case execution, test results evaluation) should also, as much as possible, be executed automatically without the need of human intervention.

Further criteria can be justified while discussing the different test activities (they are marked by Roman numbers). First the difficulties are explained followed by the implied criteria (the criteria are marked by Arabic numbers):

I - Starting the test:

The tested component A may be involved, for example, in other testing processes: Another component D may also send a testing request at the same time as component B. Thus, at the same time many test requests from many *testers* with different test inputs may be received. Therefore, the following criterion concerning the starting time can be identified:

C.2 The testing approach has to detect the right moment for testing (Scheduling).

II - Execution of the test:

Having found the right starting time for testing, the resource constraints of the test case have to be initialised: There are resource constraints that may indirectly affect the results of the test case e.g. bandwidth. These resource constraints can not be set at run time: When testing occurs at deployment time we have control over almost all resource constraints. In contrast, when testing occurs within a mobile CBS, many resource constraints are determined at runtime. The computing resources of a mobile device are always changing:

C.3 Executing the test case in a mobile system has also to consider steadily changing resource constraints.

At run time testing processes are running and “normal” functionalities may run simultaneously. Thus, the testing approach has to manage:

C.4 Concurrency of testing and normal behaviour on system level: The tested component A may be in use by other components, when the test begins: The functionality of the tested component should not be affected by the test. That means the functionality of all other components in the system that use the functionality f should not be affected by the testing process.

C.5 Concurrency of testing and normal behaviour on component level: A component may offer several functionalities; suppose the component A also offers functionality f ` which operates with data that is also used by the tested functionality f. In order to avoid confusion, the test has to

be set up so that the other functionalities of the component are not affected.

The testing approach should also guarantee system reliability:

C.6 Robustness in respect of testing effects: Neither the tested component is damaged nor an unintended system crash is caused, because of the testing process.

Component A is installed on a mobile device Z. The location of Z may be changed at any moment, so that component A and component B can not communicate anymore. That means, the testing process may be interrupted at any time: Thus, the testing approach has to guarantee:

C.7 Robustness in respect of changes of the testing context.

The testing approach also has to take into account the special properties of mobile devices:

C.8 The testing approach has to be very efficient, that means that the approach should use the available resources very efficiently, e.g. :
- **a. the storage capacity of mobile devices is limited**
- **b. the computation capacity of mobile devices is limited,**
- **c. power consumption and bandwidth is limited.**

III - End of the test:

How to evaluate the test results? What does it mean, if no error is detected? If the test fails, how should the system react to the result? Detected defects can not be repaired at run time, thus they have to be logged. What does this mean for the tester (again the tester is not a person) and the tested component?

C.9 The testing approach also has to provide a mechanism for:
- **a. error logging**
- **b. and error handling.**

Table 1. The common approaches and the mobile testing criteria

	C1		C3	C5	C8			C9	
	Develop.	Deployment/ Run time			a	b	c	Error Handling	Error Logging
Meta Data	-	(X)	-	-	-	-	-	-	-
CBSFG	-	(X)	-	-	-	-	-	-	-
Reflective Wrapper	X	-	-	-	-	-	-	-	-
Retrosp.	(X)	-	-	-	-	-	-	-	(X)
BIT	(X)	(X)	X	?	-	X	X	X	X
STECC	-	(X)	-	-	X	-	-	-	-
Test Bench	-	(X)	-	-	-	-	-	-	X

3 Exploring common approaches for testing CBS

Many CBS testing approaches are presented in the literature. A detailed overview of CBS testing approaches is presented in [1], [2]. In [2] Beydeda and Gruhn distinguish between approaches that tackle the absence of information required for testing and those who try to test the component, although information is not available. In Memon's article "A Process and Role-Based Taxonomy of Techniques to Make Testable COTS Components" in [1] a taxonomy is presented: For each approach the role of the developer and the role of the user are investigated. We present another overview of popular approaches: Each approach is shortly introduced and evaluated against the criteria mentioned in Sec. 2. Our ambition is to detect the criteria that are already supported by the popular CBS testing approaches and can be applied in the context of testing mobile CBS. Table 1 presents an overview of the approaches. Criteria that are not supported by any approach are not listed in Table 1. An X is used to indicate that the approach fulfills or supports the criterion. An (X) indicates that the approach may support this criterion under certain conditions. The testing approaches realize testing steps in various ways. Some approaches fulfill some criteria partially. For example, an approach may support error logging, but not error handling. Thus, we split some criteria into subcriteria, in Table 1, in order to investigate to which extent this criterion is fulfilled: The subcriteria of criterion 1: An X in column Development indicates that the test cases for the approach in this row are created or generated at development time. Whether the test cases are created manually or generated automatically makes no difference for our purpose. In the second column: As mentioned in Sec. 2 there is no deployment time in mobile systems, therefore deployment and run time are merged in this column. An X in this column indicates that the test cases can **automatically** be generated at deployment resp. runtime. An (X) indicates that the test cases can not always be generated (i.e. sometimes a human intervention is needed). The subcriteria of criterion 8 are already explained in Sec. 2. We distinguish with Criterion 9 whether the testing supports error logging and/or error handling. The other columns C3 and C5 do not need subcriteria.

For every approach the features that can be transferred into a mobile CBS testing approach are pointed out. In Sec. 4 we discuss these features.

3.1 Meta data approach

In [12] a framework is introduced to help the developer of the component (in our scenario the developer of component X) to add meta data to the component. In this approach the developer enhances the component with information about the static and dynamic aspects of the component the so-called "meta data". Since different meta data in different contexts are needed, and the user may not need all given meta data, the meta data can be generated on demand, or remotely stored. The developer may deliver as meta data: state machine, pre-conditions, post-conditions, invariants, test inputs and test oracle. The developer can also extend the meta data. The user (in our scenario component B) can query the meta data, and the user can then (automatically or manually) generate test cases depending on the meta data given. The user effort for generating and processing of tests depends on the amount and quality of the delivered meta data. In the worst case the component user is responsible for most test steps:

e.g., test inputs, test oracles. Thus, we have an (X) in Table 1, under Dev/Run time in this row. If the delivered meta data are insufficient, the user (component B) cannot generate the test cases automatically at run time.

3.2 Component-based software flow graph (CBSFG)

Another approach for testing components is presented in the article "Testing Component-Based Systems Using FSMs" in [1]. The user (component B) of component (A) is able to create test cases by using a graphical representation of the component the so-called *component-based software flow graph* (CBSFG). A CBSFG combines information of finite state machines and control/data flow graphs. Thus, it is possible to combine black-box-testing und white-box-testing. First the developer (of component A) models the behaviour of the component by using finite state machines. One of the main focuses of the modeled state machines is the transition between the states. The information contained in the transitions is transformed into Java code by using nested *if-then-else* constructs. Within the *if-then-else* construct it is possible to check if the component is in the right state, and to call the corresponding action, if the guards are fulfilled. In the second step the Java Code for the transitions is built into the source code of the component. After the successful integration, the developer (of A) can create the CBSFG out of the source code and it contains all the information about the control/data flow and about the states and state transitions of the component. Using the CBSFG the user (component B) can create test cases. All steps mentioned above can be automatically executed except the test case generation step. The authors [1] are working on the automation of the test case generation. This is indicated in Table 1 by an (X) in the second column.

3.3 Reflective Wrapper

In this approach [9] specification and verification information is delivered and encapsulated within another component the so-called reflective wrapper component. The execution of the delivered information is carried out through a standardized reflection-based interface. In this component many information may be encapsulated: simple information about the formal specifications, verification history, violation checking services, and even self-testing services. In [9] the authors decided to remove the wrapper component after the development, because of the increased size of the component (A), since every component would then be delivered with one or more reflective wrapper components. The generation of wrappers can be automated, however, if the wrapper component supports all mentioned features, manual modification by the developer is required. For example, in the case of constructing test suites for self-testing services, or violation checking services. Assuming that the reflective wrapper components are not removed after the development phase, we entered an X in the second column, since the test cases are then available at run time (they are delivered from the developer of component A). The storage capacity demand in this approach is not practical for mobile CBS (Criterion 8.a). Since this approach is not intended for run time, there is no mechanism to manage the deected defects automatically.

3.4 Retro-components

Retro-components [11] are components provided with retrospectors. A *retrospector* is a software entity that records all test and execution activities, and makes them accessible to the user (component B). For example, a retrospector provides the user with information about already executed tests, recommended test cases, and coverage. A retrospector can be written by the developer (of component A). A default retrospector may also be automatically generated, if the developer adheres to a certain programming style. The developer (of A) and the buyer (of A) can customize the default retrospector. If the retrospector is kept after deployment, it offers a great possibility for the developer to trace the errors found in the functionality of the component (this indicates the (X) in the last column in Table 1). Thus, in this approach the information may be exchanged in two directions: from the developer (of A) to the user (component B) and vice versa. However, this requires a huge amount of mutual confidence between the buyer of the component and the developer. On the basis of the information stored in the retrospector, the user (B) can detect which test cases are recommended, and he or she can create the tests. Thus, retrospectors provide the user with information about the test cases. The recommended test cases can be manually created by the developer (of X) or the user (B). The test cases in this approach can not be automatically generated by the user (B).

3.5 Built In Test

The main idea in this approach is to extend the component (A) into a BIT component. A BIT component is a component with test interface(s) which increase(s) its testability [14]. The main characteristics of a BIT component are:

1- It has at least one testing interface, to support access to activities.

2- If test cases are available, a BIT component possesses two modes:

- *Normal mode*: In this mode the BIT component acts like any other component.
- *Maintenance mode*: in this mode the user can access test cases. In the maintenance mode the user (B) may validate, set or change internal component states. Such access is not possible in the normal mode; either because the interior states of a component are invisible outside the component or because the manipulation of interior states is forbidden as this may affect the whole system.

A BIT component may possess complete test cases (that's the reason of entry (X) in Table 1). The test cases may be generated [17] (this indicates the (X) in Table 1). In order to avoid the problems of large overhead, the authors [14] suggest that the test cases can be stored in an extra component the so-called "*tester*". They also defined an extra component that handles the detected errors (handler component). The BIT approach may be realized in two ways:

1) Wang et al. [16] suggest that the test cases with their implementations are provided by the developer (of A) for the developed component. The test cases for the component may be contained in the component or produced on demand. With these test cases the component (A) can test itself. The user (B) can execute the delivered test cases through the testing interface.

2) Gross [10] suggests that a BIT component (B) should possess test cases not for its own functionality, but for the functionalities of other components. With this feature the component (B) can test whether the other components (A), respectively the whole environment, fulfill its expectations. The other components in the environment can check via their test cases, whether the component (B) fulfills their expectations. This approach is called “contract testing”.

If a BIT component contains test cases, they are defined at development time. This indicates the X in the first column in Table 1. Test cases may also be generated (on demand) at run time, but only if the required information is delivered by the developers. This indicates the (X) in column 2. Thus, the delivered test cases are static and cannot be changed. The BIT approach is the only approach where we found attempts to use it at run time explicitly. For example, in [13] a BIT testing architecture for run time environment is suggested. Error propagation is managed via the configuration interface (CIF) (thus, we have an X under Error handling, and Error logging). The authors suggest mechanisms to detect failures on the component’s and system’s level. The suggested architecture can also detect resources deadlocks (this indicates the X in Column 8.c). In [13] resource competition problems are solved. No concurrency problems (testing and normal processes operate with the same data) on component level are mentioned, this indicates the question mark under C5 in Table 1.This work is an extension of the work of Wang et al. [15]. The suggested architecture contains many useful features that can be applied by a testing approach for mobile CBS.

3.6 The Self-Testing COTS Components (STECC)

This approach was developed by Beydeda [7], [3]. In this approach the developer augments the component with test tool functionality. Thus, the component (A) can test itself without the need to export its source code. Each component encapsulates an ordinary control flow graph that models the source code information. The white-box oriented test cases are generated and conducted in the STECC framework. If a class specification implementation graph (CSIG) [5] is used instead of a normal control flow graph, white box and black-box oriented test cases can be generated [4], as the CSIG combines the specification and implementation view of a class. In this case the component has to be implemented as a class. Concerning the adequacy criterion specified by the user (B), the framework determines which paths have to be traversed. The test cases are generated via the *binary search-based test case generation* algorithm [6]. Whether the expected results are provided automatically or manually depend on the specification of the component. If the component needs some results from other components to complete its calculation, the user (B) has to embed the component in the context or has to provide stubs to replace the functionality of absent components. The test cases are generated dynamically. Thus, this approach needs considerable computation time. The test cases may be generated nearly full automatically, this indicates the (X) in the second column in Table 1.

3.7 The Component Test Bench

In this approach [8] the developer enhances the component (A) with the specification of test cases. The specified test cases are stored in XML format in the so-called *test-descriptor*. This simplifies the exchanging and parsing of the data. The user (B) can generate the test case code from the specification using the *verify pattern module*. A *verify pattern* module is a stand alone module that generates the test cases implementation using the given specification. The component (A) may be implemented in many programming languages, and it may also supply multiple interfaces. There are many ways to specify test cases; the developer may specify the test cases using: a text editor, an XML editor, a GUI with input boxes for each element or a data flow editor etc.

The specification of test cases may be generated automatically using the symbolic execution approach. Unfortunately there are some difficulties using this method in a number of programming languages. Thus, automatic generation of test cases specification is not always possible. If the specification of the test case is available, it can be generated at run time. This indicates the (X) in the second column. After each execution of the test case code, the test descriptor file is updated with the executed results (this indicates the X in the last column in Table 1). Thus, a history of the calculated results is available, but no error handling mechanism is available.

4 Discussion

All approaches mentioned above require more or less human intervention. Some of them are not applicable at run time. Even in these approaches we found useful features. For example, the retrospector approach offers a mechanism for error logging; this mechanism can be used, in order to log detected error at run time. Enhancing the component with adequate meta data (meta data approach) enables the generation of test cases at run time. We may extend the meta data with the reaction to test results, in order to react on test results at run time. We found, however more interesting features in the following approaches: BIT, reflective wrapper, and Component Test Bench. In the following, we summarize the features we identified:

The BIT approach: The contract testing suggested in [10] can be used to answer our main question in testing mobile CBS: “Do we have a connection to a *good* component?”. Using periodic executing of “self-testing in [16]” can detect failures caused by resource fluctuation or by topological changes (e.g. a component is removed or added) in the environment. A BIT testing architecture for run time environment is suggested in [13]. The authors in [13] assume that the system is static. That means, the system *knows* its components in advance. Although the architecture in [13] is designed for static systems, it contains many useful features that can be applied for mobile CBS. As we see in Table 1 the BIT approach fulfills “most” criteria listed in the table, but not all criteria mentioned in Sec. 2: The BIT approach needs a big storage for the test cases. This problem can be circumvented by generating the test cases on demand, or storing them in a remote component [14].

Since this approach fulfils most criteria, it is evident to use it as a base for an approach for testing mobile CBS.

The reflective wrapper approach: The authors [9] suggest removing the reflective wrappers after development and deployment because of the storage cost. If the reflective wrapper is kept after deployment, the approach is still not applicable for testing mobile CBS, but it offers interesting features: verification history can be used for logging the detected defects; self testing can be periodically executed to check the functionality within a changing environment.

The Test Bench approach: This approach offers a logging mechanism, and tools for encoding test cases in a portable format (XML). The used specification of test cases can be used and enhanced: For example, it can be enhanced with reactions to test results.

As mentioned in Sec. 2 mobile systems have special needs. An approach for testing such systems has to accomplish many aspects. Many of the required criteria are available, but not in one testing approach. Sometimes one approach (BIT) has more than one attempt (in [10] and [16]) and each attempt fulfills a part of the criteria in Sec.2. We believe that an approach that is built on the BIT approach in combination with the identified features by other approaches in a resource customized form will satisfy all the requirements for testing mobile CBS mentioned in Sec. 2.

5 Conclusion

In this paper we presented essential criteria that a testing approach for mobile CBS has to satisfy considering the:

- limited resources of mobile devices
- scheduling problems
- robustness requirements
- problems with error handling.

Most of these criteria are needed, because the tests are executed at run time in a steadily changing environment. For the most popular approaches we investigated to what extent they fulfil these criteria. Unfortunately neither of the investigated approaches fulfils all the required criteria. But we have to remember that these approaches have not been developed for steadily changing systems, and most of them assume that testing is needed only at deployment time. This applies to software systems with static structure, but not for dynamically built systems like mobile CBS. We found many features that can be applied in testing mobile CBS. We believe, it is possible to develop a testing approach for mobile CBS which satisfies all criteria mentioned in Sec. 2. In our project MORABIT (**M**obile **R**esource **A**daptive **B**uilt **I**n **T**est) we are working with our partners (Mannheim University and the EML Research gGmbH) on a testing approach built on the features already realized by other approaches. As the name says, the BIT approach plays a central role in our project, as it satisfies most criteria (see Table 1). But we are also working on the integration of the features found in other approaches.

Acknowledgements

This work is funded by the Landesstiftung Baden-Württemberg in the project MORABIT.
We want to thank our partners from the EML Research gGmbH, and Mannheim University for their discussion and support. We also want to thank Carsten Binnig, Timea Illes, Andrea Hermann, and Jürgen Rückert for their comments and advice.

References

1. Beydeda, S, Gruhn, V. (eds.): Testing Commercial-off-the-Shelf Components and Systems. Springer Verlag, Berlin Heidelberg New York (2004)
2. Beydeda, S., Gruhn, V.: State of the Art in Testing Components. IEEE Computer Society Press, (2003) 146-153
3. Beydeda, S., Gruhn, V.: The Self-Testing COTS Components (STECC) Strategy – a new form of improving component testability. ACTA Press, USA (2003) 222-227
4. Beydeda, S., Gruhn, V.: Black – and White-Box Self-testing COTS Components. IEEE Computer Society Press, Canada (2004)
5. Beydeda, S., Gruhn, V., Stachorski, M.: A graphical representation of classes for integrated black- and white-box testing. IEEE Computer Society Press, Italy (2001) 706-715
6. Beydeda, S, Gruhn, V.: BINTEST – Binary Search-based Test Case Generation. IEEE Computer Society Press, USA (2003) 28-33
7. Beydeda, S.: The Self-Testing COTS Components (STECC) Method. PhD thesis, Universität Leipzig (2003)
8. Bundell,G.A., Lee, G., Morris, J., Parker, K., Lam, P.: A software verification tool. IEEE Computer Society Press, (2000) 137-146
9. Edwards, S.H.: Toward reflective metadata wrappers for formally specified software components. OOPSLA workshop, USA (2001)
10. Gross, H.G.: Component –Based Software Testing with UML. Springer Verlag, Berlin Heidelberg New York (2004)
11. Liu, C., Richardson, D.: Software components with restrospectors. International workshop on the Role of Software Architecture, Italy (1998) 63-68
12. Orso, A., Harrold, M.J., Rosenbluhm, D.: Component metadata for software engineering tasks. LNC, Springer Verlag, USA (2000) 129-144
13. Vincent, J., King, G., Lay, P., Kinghorn, J.: Principles of Built-In-Test for Run-Time-Testability in Component-Based Software Systems. Software Quality Journal, Springer Science +Business Media B.V., Formerly Kluwer Academic Publishers B.V, (2002) 115-133
14. Vincent, J.: BUILT IN TEST VADE MECUM –PART I A common BIT Architecture. Component+ Project Report, http://www.component-plus.org/sidor/reports.htm, (2002)
15. Wang, Y., King, G., Patel, D., Patel, S., Dorling, A.: On Coping with Real-Time Software Dynamic Inconsistency by Built-in Tests. Annals of Software Engineering, Oxford (1999) 283-296
16. Wang, Y., King, G., Wickburg, H.: A method for built-in tests in component-based software maintenance. IEEE Computer Science Press, Netherlands (1999) 186-189
17. Wang, Y., King, G.: A European COTS Architecture with Built-in Tests. LNCS, Springer Verlag, Germany (2002) 336-347