The MORABIT Approach to Runtime Component Testing

Dima Suliman,
Barbara Paech,
Lars Borner
University of Heidelberg
Institute of Computer Science
Germany-69120 Heidelberg
{suliman,paech,borner}@
informatik.uni-heidelberg.de

Colin Atkinson,
Daniel Brenner
Mannheim University
Institute of Computer Science
Germany - 68161 Mannheim
atkinson@informatik.uni-
mannheim.de
dbrenner@uni-mannheim.de

Matthias Merdes,
Rainer Malaka
EML Research gGmbH
Germany-69118 Heidelberg
matthias.merdes@eml-r.villa-
bosch.de
rainer.malaka@eml-d.villa-
bosch.de

Abstract

Runtime testing is important for improving the quality of software systems. This fact holds true especially for systems which cannot be completely assembled at development time, such as mobile or ad-hoc systems. The concepts of Built-In-test (BIT) can be used to cope with runtime testing, but to our knowledge there does not exist an implemented infrastructure for BIT. The MORABIT project realizes such an infrastructure and extends the BIT concepts to allow for a smooth integration of the testing process and the original business functionality execution. In this paper the requirements on the infrastructure and our solution are presented.

1. Introduction

Software systems that cannot be assembled completely at development time, such as mobile systems or ad-hoc systems, emerge through composition of components at runtime. Since the participating components are not known in advance, the system cannot be tested fully at development time or at deployment time. In order to ensure that the components act correctly within the steadily changing environment they have to be tested at runtime. The Built-In Test approach (BIT) [9] has been developed to enable runtime testing. As we discussed in [7] it satisfies many important criteria for testing mobile systems. In our project MORABIT (Mobile Resource Adaptive Built-In Test) we enhance BIT in several ways. We support different ways of allocating the responsibilities of the testing process to the components. Furthermore, we monitor and control test resources in order to minimize the effects of the testing on the business logic execution. And, we have implemented a prototypical infrastructure (component container) realizing these features. To this end, we define in detail how a component can trigger tests and react to test results.

In the next section we review the general idea of BIT and related literature. In section 3 we sketch a mobile auction scenario which serves as the running example for this paper, and give examples of runtime failures which should be detected with our approach. In section 4 we provide the most important concepts of the MORABIT approach. In section 5 we discuss the current status of the implementation. Section 6 concludes the paper.

2. Built In Test

Many testing approaches can be referred to as Built-In test. An overview of the best known Built-In testing approaches is presented in [1]. In the MORABIT project [5] we build on the BIT approach proposed by the component+ project [3], since it suggests many useful concepts that can be applied to runtime testing. The main idea in this approach is to enhance the component with testable interface(s). Such an interface provides methods to perform tests on the component. To support this, each BIT component possesses two modes:

- Normal mode: the component performs the original business functionality
- Maintenance or test mode: in this mode a tester (component) can set or check internal logical states of the component through the testable interface.

Note that interior variables are still invisible to the outside. They can be manipulated implicitly through manipulation of logical states that encapsulate these physical variables, but their values resp. names are still hidden. In other words the tester does not know how the logical states are implemented by the component.

When transferring the idea of testing to runtime, a fundamental question is who is responsible for what. The responsibilities encompass providing the test cases, starting and reacting to the test. As shown in Table 1 there are several options for who and what:

Table 1: Runtime test responsibilities

Who / what	**Test cases by CUT**	**Test cases by client or specific tester components**
Test requested by CUT	Self-test [9][10]	
Test requested by client		Contract test [4]
Test req. by specific tester components	Centralized test [8]	

(**who**) Either a component under test (CUT) tests it-self or the CUT is tested by another component. This other component could be the client or a third (infrastructure) component without business functionality, but with specific functionality to perform tests on other components.

(**what**) The test cases can be brought by the CUT or another component.

Three cases have been so far considered in the literature: Wang et. al. [9], [10] use self-test, in order to check whether the component acts correctly within a new environment. A simple case study for this approach is presented in [11]: A COTS component of binary search was enhanced with self-test mechanism. In contrast, Gross [4] uses contract test in order to ensure that server components fulfill the expectations of their clients.

In [8], an architecture for BIT within a runtime system was presented. The BIT facilities are used by a separate handler and tester component: The *tester component* can configure and execute the test modules that are provided by the CUT. The test results are processed by the *handler component* which takes responsibility for a certain quality property of the system e.g. avoiding resource dead lock (centralized test). This architecture presupposes that all defects can be detected and handled by test cases supplied by the developers of the CUT. But the developer of the component cannot foresee all possible use scenarios of the component and sometimes the test reaction affects only part of a system. In other words, a client component often needs to test its server and react to the tests f*rom its point of view*. If the test cases for the server functionality are brought by the server only, they represent merely the server aspect, and thus they cannot test whether the server component acts as the client component in the environment expects. In MORABIT we therefore support all six possible responsibility distributions for the testing process. Note that this discussion is independent of whether the test cases are associated with a component at development time or generated at runtime. In the following we call all tests requested by the CUT *self-tests* and all tests requested by a client or a specific component *contract-tests*.

To our knowledge there does not exist an implementation of the BIT concepts so far. So the mentioned approaches do not define in detail how to perform the test concurrently with the business logic. In particular it must be ensured through *test isolation* that the original business functionality is not compromised and through *resource monitoring* that system performance is not compromised. In the following we sketch how this is achieved by the MORABIT approach.

3. Auction Scenario and Runtime Failures

In this section we sketch a simple auction system which we have realized and tested to evaluate our infrastructure.

In the auction scenario the participants communicate with the auction house via mobile devices e.g., laptop or PDA. Unlike fully electronic auction applications like ebay, the users of this system need to actually be present at a physical auction. The system supports the auction by allowing users to offer and bid for items, use facilities such as e-mail etc., and conduct payment transactions electronically. Participants come and go, and thus the components with which the auction house component communicates are steadily changing. After registration participants can join auctions. Then they can bid for items in the auction. At the end of an auction the auction house notifies the winner and asks him or her to authorize payment. After successful money transfer, the winner gets the auction item. The currency of the winner account may differ from the currency of the auction house. Altogether the software system consists of an auction house component, several participant components, at least one bank and currency converter component. So which kind of problems can be detected through runtime testing for such a system?

Typical test concerns for BIT are described in [8], [9], [10]. In [9] *static inconsistencies* due to design, implementation and maintenance errors are distinguished from *dynamic inconsistencies* due to resource errors, code corruption (e.g. through a virus), environment incompatibilities, configuration errors and bugs. Only dynamic inconsistencies are the focus of BIT such as resource errors or configuration errors. In dynamically built systems that cannot be fully assembled at development time, dynamic inconsistencies errors can arise also due to *incompatibilities between components*. A component might not deliver its expected service, because there is a misunderstanding with the client component. These errors are the focus of contract testing. The misunderstandings concern the contract between the components in terms of inputs, outputs, states, exceptions and quality metrics.

In the following we give some examples for such misunderstandings within the auction scenario:

- **Input order misfit:** At the end of an auction and payment authorization, the auction house has to invoke the method "transfer money" of the bank with the arguments: source account, target account, and money amount. If the auction house calls the method "transfer money" with wrong argument order (target account, source account, money amount), the transfer will not be successful. Since both arguments, source and target account, are of the same type, this will not be detected by the bank before the transfer is executed.
- **Input interpretation misfit:** The auction house could offer a service for automatic bidding, where an amount is added to a bid whenever it is no longer the highest. The input to such a method could be the upper limit for the bidding. The participant however, might provide the minimal amount for the bidding.
- **Output interpretation misfit:** The currency converter may convert with less accuracy than the bank or the auction house expect.
- **Control state misfit:** A participant tries to bid in an auction before s/he has joined the auction.
- **Quality misfit:** The auction house has a much longer response time than expected from the participant.

In the following we discuss how the MORABIT approach copes with these dynamic inconsistencies.

4. The MORABIT Approach

In this section we present the requirements on and solution provided by our approach. In section 4.1 we give an overview of the main use cases and domain data for developers and testers of MORABIT components. In section 4.2 we describe how to define and execute test requests with the help of the MORABIT infrastructure. Section 4.3 exemplifies how some of the inconsistencies given in the previous section are handled by the infrastructure.

4.1 MORABIT Use Cases and Domain Data

As shown in Fig. 1, MORABIT supports the component developer and test designer in the preparation of MORABIT components, and the test administrator and a tester component in the test execution. The tester component is any component performing a self-test or a contract-test. A human test administrator is needed within the MORABIT runtime environment, in order to (re)configure the infrastructure e.g., setting the resource aware strategy to be adopted. The test administrator can also test components within the MORABIT environment. This is basically the same use case as the one performed by a component testing another component. After the execution of a test the reaction to the test result has to be performed.

In Fig. 2 part of the domain data model corresponding to the *test component* use case is shown. The main entities are: A **MORABIT component** is a software component that offers an *extended interface* and that is associated with *test requests*. An extended interface consists of a *service interface* and a **testable interface**. The service interface comprises the business functionality of the component. The testable interface provides methods to support testing, in particular methods to set and get details of the logical state.

The test designer provides self-test cases or contract test cases for the component in terms of a *test suite*. A test suite is part of a **test request** of the component. The test request is the main information exchanged between the infrastructure and a component. At development-time the test designer prepares test requests to check for inconsistencies. The test request in addition to the test suite defines *test timing policy, test reaction policy and resources needed*. The policies are explained in section 4.2. The resources are needed to allow resource-aware scheduling of tests. This is not treated in this paper.

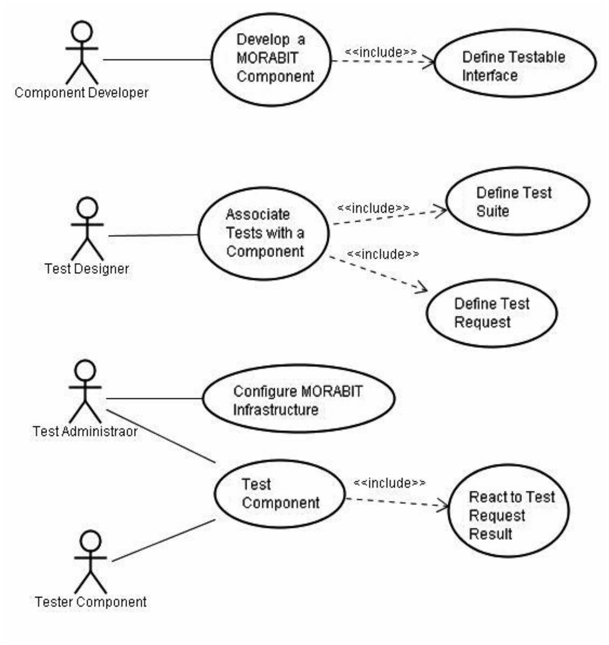

Figure 1: Use cases

A *test request response* is the infrastructure's answer to a sent test request (e.g. test is performed now, test is not possible): The *result of a test request* is the number of test cases in the test suite which were performed successfully. Here we distinguish between confidence (how many test cases have been performed) and reliability (how many of the executed test cases were successful). This is useful to also allow reliability calculations based on the tests. All executed tests should be centrally captured in a *test log*.

In the next section we sketch our solution how to handle test requests.

4.2 MORABIT Infrastructure Concepts

Besides resource-awareness (which is not treated here) the main feature of the MORABIT infrastructure is the handling of test requests without compromising the business functionality of the components.

4.2.1. Test Request Definition. As mentioned in section 4.1 the test request encompasses all test information handed over to the infrastructure. The test suite consists of the set of test cases to be executed.

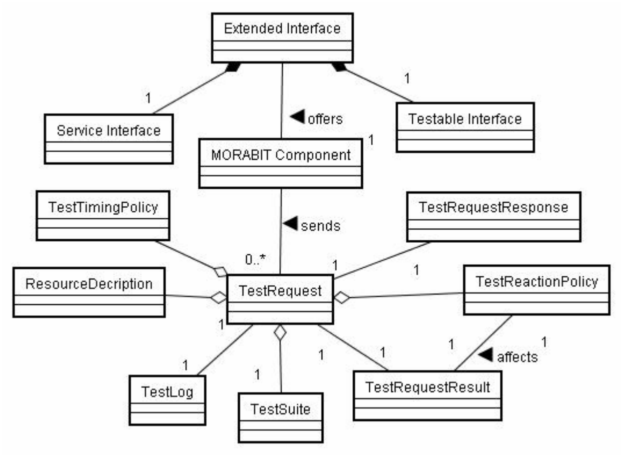

Figure 2: Domain data model

The **test timing policy** describes when the infrastructure is requested to perform the test. There are different test timing policies:

- **idle time tests**: When a component is idle, the component might want to run self or contract tests .
- **server acquisition time tests**: Before acquiring a component as a server, the client might like to execute tests so that the two components are only connected, if the test was successful.
- **call time tests**: When a client component calls a method of a server component, it might like to execute tests.
- **periodic tests**: As soon as a connection to a server is established, the client might like to run tests periodically.
- **topology change time tests**: If the topology changes (e.g., components are replaced by others with the same service interface), a component may re-execute test cases.

Clearly, not all start times are suitable for each test. Self-test should be started at idle time, periodically or triggered through environmental change, while contract test should typically be started at server acquisition or call time.

A **test reaction policy** is needed to react to negative test results**:** The test reaction may be *component driven* i.e., it is carried out by the component that issues the test request. As a component driven test reaction the component may switch to a different way of providing its own services e.g., use a different algorithm that does not need the CUT. The test reaction may also be carried out by the infrastructure (*infrastructure driven reaction*). An example for an infrastructure driven reaction is "try next", e.g. at server acquisition time after a failed test the infrastructure tries to find another server for the requested interface. Infrastructure driven reactions are to some extent similar to the handlers mentioned in [8]. However, handlers in [8] are classified according to specific quality (e.g. ensuring deadlock freeness), while infrastructure driven reaction are classified according to the actions which are performed (e.g., try next).

4.2.2 Test Request Execution and Isolation. For each active component within the MORABIT environment the infrastructure checks, whether it possesses test requests. The infrastructure executes the test requests according to their test timing policy, and the actual resource status. One major problem here is that test cases should not interfere with the business functionality of the components (we call this *test isolation*). Suppose a component B tests component A, and during the testing process component A receives a regular business service request from another component C. How should A react to this request?
There are the following options:

1. Component A responds to the request *just as if A were not tested.* If A manipulates resp. uses data that is already manipulated by the testing process, A may give a wrong answer to C or to B or both. Therefore the developer has to indicate whether A is *sensitive to testing*.
2. Component A is *blocked* during the whole testing process. This may affect the performance of the whole system, since all components that need the tested component A have to wait until the testing process is finished. This option has to be avoided.
3. Component A *aborts* the testing process and the original state of A (before the beginning of the test) is restored. This option will affect the performance of the components that test component A, because they have to wait until the requested test can be completed. Clearly, this option is not satisfying.
4. Component A is *cloned* by the infrastructure before the test starts. The testing process is performed with the clone and the service process is performed with the original component. This option can be very expensive e.g., when, the number of needed clones becomes very large.
5. The testable interface of component A provides methods that allow *test sessions.* These methods ensure that test data and real data are not mixed during the testing process. This could be done also through cloning (driven by the component itself) or by providing specific methods. This option puts additional burden on the component developer.

Since there is no optimal strategy for test isolation, we propose to use a mixture. The infrastructure checks whether the component A is not sensitive to testing or provides test sessions. If yes, the service is performed (option 1 or option 5). If not, the infrastructure tries to clone the component (option 4). If this is not possible, the testing process is interrupted (option 3).

4.3 Examples for Test Request

Here we demonstrate briefly two test request examples that detect misunderstandings mentioned in section 3. In general, the definition of appropriate test cases for runtime is not a simple task.

First we treat the input order inconsistency: The auction wants to test the transfer method of the bank. This is a case where the CUT (the bank) needs to provide specific methods in its testable interface (test sessions). The auction house cannot transfer money between two *real* accounts, in order to test whether it shares the same understanding of the method “transfer money” with the bank component. If the bank provides methods for test sessions e.g., “create test account” in its testable interface, the test case can create two test accounts, invoke the method “transfer money”, and finally check the result without changing the *real* data of the CUT. Since the auction house does not know the winner’s bank in advance, its test time policy could be call time. In the case of a negative test result, the auction house stops the bank transfer and informs the winner (component-driven reaction).

Now we treat the output interpretation inconsistency. Here the problem is the definition of the test oracle. E.g., if the bank component wants to test the currency converter, the bank component does not know the result, since the exchange rate is always changing. Thus, the test case should be able to set the exchange rate of the currency converter component, before calling the method “convert”. That means the currency converter is sensitive to testing. If the currency converter does not offer a test session, the infrastructure has to clone the currency converter component. Start time is server acquisition time, since the bank component has to establish a connection to a reliable currency converter from the beginning. If the test fails, the infrastructure could test another currency converter from the available currency converter components within the environment. Thus, “try next” is an appropriate test reaction policy (infrastructure driven). Finally, the infrastructure hands a reference to the currency converter that fulfils the test requests.

5. Implementation

In this section we present some technical details about the prototypical implementation of the MORABIT infrastructure. In MORABIT we have defined a very light-weight component model which minimizes the coupling of MORABIT components to the infrastructure API. This is in accordance with current trends in industrial component technologies such as the Spring framework or even the new EJB 3.x standard and is achieved by using an empty marker interface to denote MORABIT components. Each component is defined and introduced to the infrastructure by a configuration file which specifies its fully qualified class name enabling the infrastructure to dynamically instantiate components. At runtime components can lookup other components offering a specific service by contacting the infrastructure via a standard service locator interface. The prototypical infrastructure is implemented in Java, and the components must also be implemented in Java. The concepts are, however, programming language independent. The test code is not glued to the functional code of the component. That means, tests can be disabled or enhanced after the release of the component.

We have implemented the auction scenario sketched in section 3 as MORABIT components. The scenario is described in more detail in [2]. The current infrastructure implementation does not support distributed components so far. Its foremost aim is to show that the testing concepts are viable. We believe that typical concepts for distribution can be added on top (e.g. load balancing).

We have also implemented first tools to support the use cases described in section 4.

6. Conclusion

Testing at runtime is a non-trivial task. It plays an important role for component based system which cannot be assembled fully at development time. In this paper we presented the first results of the project MORABIT. The current MORABIT infrastructure supports testing at runtime through many useful features such as test isolation, test scheduling, and resource monitoring. The developed infrastructure can also be applied for testing at deployment time. The flexible structure of the test request allows adding or discarding test cases within a test suite without the need to change the source code of the component. Clearly, the test reaction policies for testing at deployment time might be different from those at runtime, since in contrast to testing at runtime qualified staffs are usually available. After the deployment of components within a static system some tests may be removed since nothing will change.

Future work concerns detailed guidelines that describe how to develop and enhance MORABIT-components. First ideas are sketched in [6]. In particular we investigate guidelines that describe how to derive *runtime relevant* test cases systematically considering the inconsistencies that may appear at runtime.

Currently we are working on tools that alleviate the definition of test requests e.g. through an Eclipse Plug-In. We are extending our application example to evaluate whether we need to enhance the test reaction or test timing policies.

Acknowledgements

This work is funded by the Landesstiftung Baden-Württemberg in the project MORABIT. Rainer Malaka and Matthias Merdes thank the Klaus Tschira Stiftung for its support.

References

[1] S. Beydeda, Research in Testing COTS Components – Built-in Testing Approaches, International Conference on Computer Systems and Application, Cairo, 2005.

[2] D. Brenner, A Case Study for Resource Adaptive Built-in Test Components. Diploma Thesis, University of Mannheim, 2004

[3] Component + project, http://www.component-plus.org

[4] H.G. Gross, Component –Based Software Testing with UML, Springer, Germany, 2004.

[5] MORABIT project site http://www.morabit.org.

[6] B. Paech, A. Atkinson, R. Malaka, L. Borner, D. Brenner, M. Merdes, D. Suliman and D. Dorsch, MORABIT delivarable M2, 2006

[7] D. Suliman, B. Paech, and L. Borner Testing Mobile Component Based Systems. Proceedings of the NET.OBJECTDAYS, pp 213-223, 2005/

[8] J.Vincent, G. King, P. Lay, J. Kinghorn: Principles of Built-In-Test for Runtime-Testability in Component-Based Software Systems. Software Quality Journal, Springer Science +Business Media B.V., Formerly Kluwer Academic Publishers B.V, (2002) 115-133.

[9] Y. Wang, G. King, D. Patel, S. Patel, A. Dorling: On Coping with Real-Time Software Dynamic Inconsistency by Built-in Tests. Annals of Software Engineering, Oxford (1999) 283-296

[10] Y. Wang, G. King, and H. Wickburg, "A method for built-in tests in component-based software maintenance", IEEE Computer Science Press, 1999.

[11] Y. Wang and G. King "A European COTS Architecture with Built-in Tests," *Lecture Notes in Computer Science* vol. 2425 / 2002, pp. 336-347, 2002