Reducing Verification Effort in Component-Based Software Engineering through Built-In Testing

Daniel Brenner[1], Colin Atkinson[1], Barbara Paech[2],
Rainer Malaka[3], Matthias Merdes[3], and Dima Suliman[2]
1- Institute of Computer Science, University of Mannheim, 68131 Mannheim, Germany,
E-mail: {dbrenner|atkinson}@informatik.uni-mannheim.de
2- Institute of Computer Science, University of Heidelberg, 69120 Heidelberg, Germany,
E-mail: {paech|suliman}@informatik.uni-heidelberg.de
3- EML Research gGmbH, 69118 Heidelberg, Germany,
E-mail: matthias.merdes@eml-r.villa-bosch.de, rainer.malaka@eml-d.villa-bosch.de

Abstract

Today component- and service-based technologies play a central role in many aspects of enterprise computing. However, although the technologies used to define, implement, and assemble components have improved significantly over recent years, techniques for verifying systems created from them have changed very little. The correctness and reliability of component-based systems are still usually checked using the traditional testing techniques that were in use before components and services became widespread, and the associated costs and overheads still remain high. This paper presents an approach which addresses this problem by making the system verification process more component-oriented. Based on the notion of built-in tests – tests that are packaged and distributed with prefabricated, off-the-shelf components – the approach and supporting infrastructure help to automate some of the testing process, thereby significantly reduces system testing effort. After providing an introduction to the principles behind component-based verification, and explaining the main features of the approach, we show by means of a small example how it can reduce system verification effort.

1. Introduction

Because they can significantly reduce the levels of effort involved in building and developing systems [1], components and services today play a central role in most software engineering projects, particularly enterprise computing projects. However, in their currently most widely used form, component and service technologies have little or no impact on the level of effort needed to verify such systems once created. Component-based systems are still typically verified using the same system testing techniques that were used before the notions of components or services became widespread. These techniques are not only very expensive, they are also unable to use knowledge about a system's component structure to pin-point the source of failures because they treat them as monolithic, black boxes. As a result, many of the benefits of component-based development are defeated by the costs involved in verifying the systems using traditional techniques. This problem is not unique to enterprise systems, but is much acuter at the enterprise level because of the shear sizes and number of components involved. Finding better verification techniques is thus a central challenge in enterprise computing.

For small systems under carefully controlled conditions it is theoretically possible to prove or calculate the correctness or reliability of a system given the correctness or reliability of its components [2]. However, such mathematically rigorous techniques rarely if ever scale up to enterprise systems, and are only applicable to relatively static architectures. Systems which are continually changing their configuration, such as those serving dynamically changing sets of users or those composed of ad hoc collections of components, cannot be analyzed using these techniques

Until rigorous methods for analyzing enterprise scale systems become available, dynamic testing techniques will remain the only practical way of gaining some confidence of their fitness for purpose. The challenge is thus to enhance the traditional,

component-agnostic testing techniques available today to make them more component-friendly. One of the most promising ways of doing this is the notion of built-in testing (BIT), first suggested by Wang [3] and later refined in the Component+ project [4]. The basic idea behind this approach is to build into components the ability to test their environments at run-time so that they can perform much of the required system validation work "themselves". Although the Component+ project defined many of the basic ideas behind the approach, however, it did not take the issue of resource-awareness into consideration or fully elaborate how traditional development practices need to be enhanced.

Because testing is a resource intensive activity which by definition involves the execution of a component's normal functional code, it is only practicable if the tests are executed when the load on the system is sufficiently low. In other words, built-in tests should only be executed when they will have a minimal, or at least an acceptable, impact on the performance of the component. This implies the need for an intelligent run-time infrastructure (or component container) which is able to coordinate the built-in testing process and orchestrate the testing of the various components.

To further investigate the feasibility of built-in, run-time testing, particularly in the domain of mobile-accessible enterprise systems where its potential benefits are at their highest, we have built a prototype version of the required run-time testing infrastructure as part of the MORABIT project [5]. In the paper we explain how this infrastructure can be used to partially automate the verification of a component that is a core part of a mobile business system. This approach promises to significantly reduce the effort involved in verifying enterprise system components and thus to lower the overall costs and time associated with enterprise systems engineering.

The remainder of the paper is structured as follows. In section 2 we present the case study which we use to illustrate the approach. In section 3 we discuss the main issues involved in developing a component-oriented approach to system verification and introduce some of our new terminology. Then in section 4 we present a high level overview of the MORABIT approach. Section 5 follows with detailed examples of how this approach might be used in the context of the case study. Section 6 discusses related work and Section 7 concludes.

2. The Auction House Example

The case study we use to illustrate the approach is an Auction House System whose job is to enable auction participants to interact electronically using mobile devices. Unlike fully electronic auction applications like e-bay, the users of this system need to be actually present at a physical auction. The system supports the auctioneer by allowing users to offer and bid for items, use infrastructure facilities such as e-mail and conduct payment transactions electronically.

The overall architecture of the Auction House System is illustrated schematically in Figure 1. Each of the nodes in this figure is a distributed component executing on an independent device. Each of the edges represents a remote interaction. The central component in the system is the Auction House. This is the central server which mediates requests from auction participants - the clients - usually hosted on mobile devices such as PDAs. The other components in the system assist the Auction House in delivering its service. The Activity Logger is responsible for storing a log of all the main activities in an auction, such as auction initiation, the offering of items, the placing of bids, the completion of auctions etc. The Auction and Participant Managers are responsible for storing the important data involved in an auction. The Mail Server is responsible for dispatching mails and the Bank takes care of handling payments.

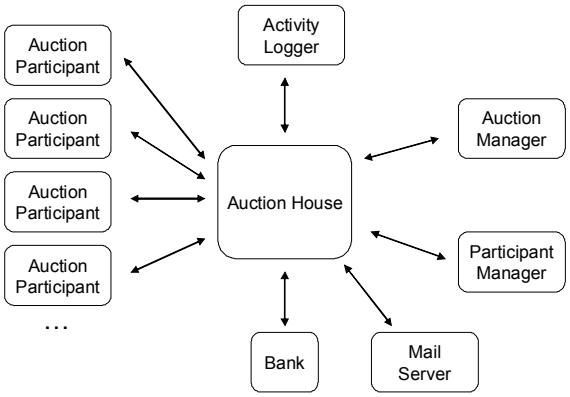

Figure 1. Auction House System architecture

Figure 1 depicts a sketch of a typical configuration of the system. The actual configuration of the system at any given point in time is highly dynamic and changes as new participants join and leave an auction. The Mail Server and Bank component may also change if compatibility or reliability problems are detected. This architecture is suitable for implementation in a variety of component technologies such as .NET, CORBA, EJBs, and Web Services.

Since it plays the central coordinating role in the architecture, the Auction House also plays the main role in determining the dependability of the system. In fact, from the point of view of auction participants the Auction House (AH) *is* the system. Determining the reliability of the AH is a non-trivial tasks because it is itself also uses components whose identity is determined at run-time. Establishing whether the AH is capable of fulfilling its contract is therefore a challenging task which would take a great deal of time and effort if performed in the traditional way. To understand how MORABIT built-in test technology addresses this problem we need to clarify the provided and required interfaces of the component, and the methods which each of these provides.

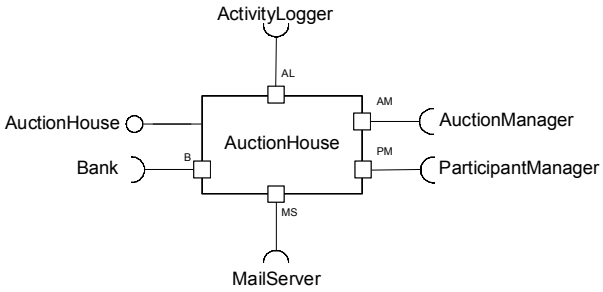

Figure 2. AH Interfaces –Plugs/Sockets Notation

Figure 2 illustrates the different interfaces of the Auction House component (provided and required) using the plugs and sockets notation of the UML, while Figure 3 shows them in a more complete form using the class notation.

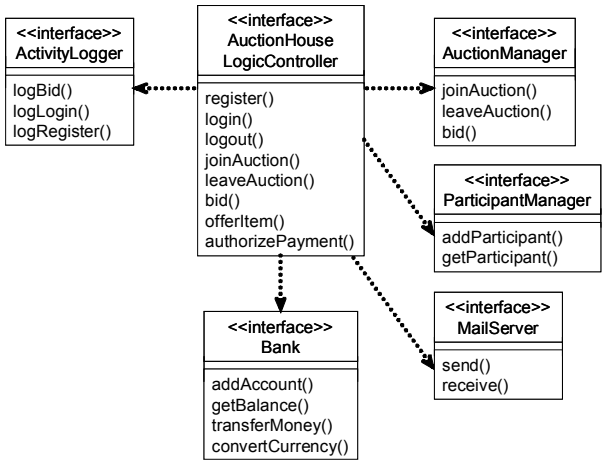

Figure 3. AH Interfaces – Class Notation

The services offered by the AH system as a whole are realized by algorithms (i.e. code) within the AH component together with the services of the other components that it uses through its required interfaces. This, in turn, means that the "correctness" of the service offered by the AH component depends on two things – the "correctness" of the functionality (i.e. code) within the component itself and the "correctness" of the component's servers (i.e. the components delivering the services that it uses through its required interface. Verifying the correctness therefore involves checking of both of these elements. To check the first, it is necessary to know the algorithms used to realize the functionality of the AH component's methods, either in the form of code or in the form of design models. An example is shown in Figure 4 in the form of a collaboration diagram. This collaboration diagram shows how the Auction House components invokes operations of the Participant Manager, Mail Server, and Activity Logger component to implement the Auction House's register() operation.

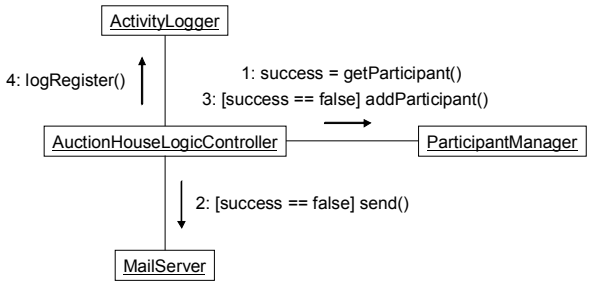

Figure 4. Algorithm for the register operation

Each operations is realized in a similar fashion.

3. Testing Component-Based Systems

In traditional development approaches two basic testing notions are used to check the quality of systems assembled from separate modules - the notions of "integration testing" and "acceptance testing". "Integration testing" is a technique for verification that focuses on the testing of successively larger groupings of modules, leading up to the system as a whole, in the context of the development environment[1]. According to the terminology of Boehm [6], integration testing aims to verify that "we are building the system right" according to some well defined description of what the system should do. "Acceptance testing", in contrast, focuses on validation and essentially involves the testing of a deployed instance of the system in the target execution environment before it is put into service. In Boehm's terminology [6] acceptance testing aims to validate that "we are building the right system" based on the expectations of the customer or users.

When systems are assembled from components at deployment-time and may have their configurations

[1] For the purposes of this paper we regard "system" testing as special case of integration testing. It is the concluding case where the integrated unit under test happens to be the complete system.

changed dynamically, these notions of testing are no longer adequate. In particular, integration testing no longer makes sense in its traditional form because the precise composition of a system is not known at development-time when integration testing is traditionally performed. In the case of the Auction House system the specific set of components making up an instance of the system is chosen when instances of its components are deployed, and these can originate from numerous vendors.

The notion of testing "the system" as an integrated whole at development-time no longer applies in the traditional sense therefore. Testing at development-time is still important, but its role is to test the code that implements a component's provided interface in terms of representative implementations of its required services. In terms of the auction house example, this corresponds to the testing of the AH component (which contains the algorithms implementing the AH's provided interface) using representative implementations of the other components (e.g. Mail Server etc.) which realize the AH's required interfaces. A "representative implementation" of a required component can either be a full working version of the component or a stub which mimics the component for a few chosen test cases. Since these tests are performed at development-time and are exclusively focused on verification against a specification, we simply use the term ***development-time testing*** for this activity.

In the context of component-based development neither the notion of integration testing nor the notion of acceptance testing is fully appropriate in its traditional form. The former is not appropriate because integration can and should no longer be fully performed at development-time as has hitherto been the case. The latter is not appropriate because the testing that is performed at deployment-time should no longer focus just on validation as has traditionally been the case. Instead, the testing activities that are performed at deployment-time also need to include tests to verify the assembly of components against the system's specification. It therefore makes sense to combine the notions of integration and acceptance testing into a single activity known as ***deployment-time testing*** (where, as in our example, a component can be the system in the traditional sense). Such a deployment-time, component testing activity serves the dual roles of validation and verification of the assembled system in its run-time environment.

For systems whose structure remains constant after initial deployment there is clearly no need to revalidate the system once it has been placed in service because any tests that have been performed will not be able to uncover new problems. However, many component-based systems do not have a constant structure. On the contrary, an important benefit of component-based development is that it allows the structure of a system to be changed while it is in service. In our auction house case study, for example, the external components that are used to realize the AH's required interfaces, such as the Mail Server or the Activity Logger etc., may be changed dynamically at any time. If a change is made, then clearly any results of tests performed at deployment-time may no longer be valid.

The notions of development-time and deployment-time testing are therefore not sufficient to cover the full spectrum of testing scenarios in dynamically reconfigurable component-based systems. We need to add the notion of ***service-time testing*** as well. Service-time tests are carried out once a system has entered service and is delivering value to users (i.e. is being used to fulfill its purpose). Deployment-time and service-time testing both take place at "run-time" in the sense that they are applied to a "running" system in its final execution environment.

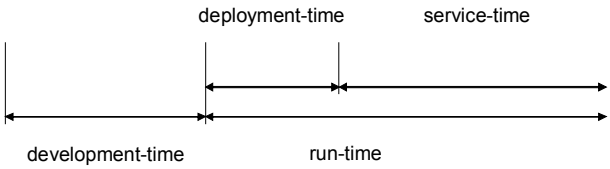

Figure 5. Life-cycle phases

The relationship and role of these different phases in the life-cycle of a component-based system are summarized in Figure 5. At the highest level of abstraction, two different phases exist, the development phase, in which the system is developed and tested using representative service providers in the development environment, and the run-time phase in which an instance of the system is connected to actual service providers and is running in its final execution environment. The run-time phase is divided into two subphases – the *deployment phase* and the *service phase*. In the deployment phase, the system is set up in its initial configuration and starts to run in its execution environment, but it is not yet delivering service to users. This is important because it allows testing activities to be performed under controlled conditions under certain specific sets of assumptions. In the service phase the system has been put into service and is delivering value to users. During service-time the assumptions that held during deployment-time may no longer be valid.

3.1. Built-in Tests

Having established the basic notions of ***when*** testing makes sense in component-based development

and reinterpreted the goal of testing at different stages in a component-based system's life-cycle, we now turn to the question of ***how*** the tests are performed. Since they essentially replace the integration and acceptance testing activities of traditional development approaches (albeit with difference emphases) development- and deployment-time testing can be performed using some suitable mix of traditional integration and acceptance testing techniques. The difference between whether tests focus on validation or verification is manifested in the nature of the test cases rather than in the actual activities or technologies used.

Traditional testing technologies (whether for validation or verification) are primarily manual activities. Although certain aspects can be automated under certain circumstances, the main steps in the testing process are driven by humans. In particular, human engineers are responsible for putting the system into an appropriate state, judging when tests can and should be executed, analyzing the results to identify unexpected behavior and working out how to respond. It is precisely because these activities require such a large amount of human involvement that verification is such an expensive part of traditional development processes. Moreover, the problem is exacerbated when traditional techniques are used to test dynamically assembled component-based systems at deployment-time because, as explained above, much of the verification work that was done as part of integration testing has to be postponed to deployment-time. As a result, the amount of testing that has to be done at deployment-time to reach a given quality confidence level in component-based systems is higher than for traditionally developed systems. This, in turn, significantly reduces the plug and play benefits of component-based development because the cost and time benefits gained though rapid application assembly are largely lost due to the extended validation effort.

The MORABIT built-in testing technology was developed to address this problem and significantly reduce the amount of manual effort needed to attain a given level of confidence in a system's quality. It can be viewed as an enhancement of the "built-in testing" approach developed in the Component+ project [4]. The basic idea behind built-in testing is to equip prefabricated components with the ability to perform their own automated tests of their environments whenever necessary to determine whether they are able to fulfill their own responsibilities. They can also be equipped with the ability to perform an appropriate reaction to any detected problems and/or to report perceived problems to human operators. The term "built-in" does not mean that code for executing and reacting to test is necessarily bound into the normal application code of a component. Rather, it indicates that components purchased from third party vendors are packaged with the necessary run-time test and reaction definitions.

As well as reducing the human effort required to reach a given level of confidence in a component's quality at deployment-time, built-in testing has another significant benefit. It also enables the dependability of component-based systems to be tested at service-time whenever changes may affect the validity of results derived from previously executed tests. For example, whenever an external server of the Auction House component is changed the validity of results derived from hitherto executed tests is no longer guaranteed. In order to maintain the same level of confidence in the Auction House system's reliability it is necessary to execute some, if not all, of the tests that were performed at deployment-time. However, this is easier said than done for two reasons. Unless one takes the component out of service temporarily (which effectively represents a temporary return to the deployment phase), the service-time tests have to be performed and reacted to at a time which

- is meaningful with respect to the component's thread of execution. Test reactions, in particular, need to be synchronized with the component's activities to ensure that any changes are made consistently.
- does not unduly affect the component's ability to deliver its service. Tests are not only highly resource intensive, they also require certain conditions to exist such as private access to server components to be available, or server components to be in specific states etc.

Thus, in order to support service-time as well as deployment-time testing, run-time time testing technology needs to have two additional characteristics: it needs to be context-sensitive and it needs to be resource-aware. These are the novel characteristics of the MORABIT run-time testing technology which are presented in this paper.

3.2. Qualitative versus Quantitative Testing

We have so far discussed the different life-cycle phases which determine when tests are performed and the built-in test notion which determines how they are performed, but we have not yet addressed what it is that the tests are aimed at discovering. Generally speaking tests can be performed with two goals in mind. One goal is to establish in a "black and white" manner whether a component or a system of components satisfies a given set of test cases. The other goal is to measure the reliability of a component

or system of components based on its satisfaction of a set of test cases. We refer to the former as qualitative testing since it yields a binary pass/fail result and the latter as quantitative testing since it delivers numeric measures of the reliability of a component.

Qualitative test suites usually have a much smaller number of test cases than quantitative test suites. In qualitative testing, the test cases are chosen according to carefully developed criteria derived from classic testing coverage principles. In contrast, with traditional defect testing, however, test cases are not chosen solely based on their likelihood of uncovering implementation errors, since it is assumed that all components have undergone extensive development-time testing in which the normal coverage criteria were used to define test cases. Rather, the test cases are chosen with a view to uncovering the most likely causes of "misunderstandings" in the nature of the service to be delivered. Typical examples are the orders of parameters or the sequencing of operations. In essence, therefore, qualitative run-time tests are driven more from the perspective of validation than of verification, since the goal is to check whether one component meets another's expectations rather than its specification. A reaction to qualitative tests is based solely on whether the component fails or passes the tests. If the component passes no action is usually taken, but if it fails, a variety of actions can be taken.

Quantitative test cases are developed to provide a statistically significant sampling of the usage pattern that a component is likely to experience in a particular run-time environment. There are therefore usually many more test cases in quantitative tests than in qualitative tests in order to attain statistical significance. Instead of a binary pass/fail value, two qualitative measures are returned – one the measured reliability of the component derived from the ratio of test case in which the component passed to those in which it failed, the other the statistical level of confidence which can be attached to the first value. The precise way in which these values are calculated depends on the assumptions and sampling model which is used, of which there can be several [7]. All approaches, however, rely on a model of the usage profile of the component. Such profiles describe the distributions of various invocation properties such as relative method invocation frequency, parameter values, and method invocation sequences [8]. The basic goal of the test case selection process is to pick a statistically significant set of test cases which most closely resembles the usage profile of the component, and thus give the best estimate of its reliability.

Both forms of testing rely on an ability to determine whether a particular invocation of a component's service succeeds or fails from the perspective of the invoker. There are three basic ways in which such an invocation can be judged to have failed

- the operation completes, but returns a value that was not the expected one,
- the operation does not complete and returns some indications to the caller that it was unable to do so (e.g. raised an exception),
- the operation does not complete within a required period of time.

In principle, all three forms can be used in both quantitative as well as qualitative testing. However, since forms (2) and (3) do not require an expected result to be determined, they lend themselves to quantitative testing. The creation of expected results for the first form of failure has traditionally been one of the biggest stumbling blocks to quantitative testing (also known as "statistical" or "random" testing) because it is difficult to do automatically [9].

4. MORABIT Infrastructure

The main contribution of the MORABIT project are to define the basic functionality and services that an infrastructure needs to provide to support run-time testing of the kind describe above and to define the basic methodology that component developers and deployers need to follow to use it. The former is briefly summarized in the subsections below.

A prototype version of the MORABIT infrastructure has been developed which demonstrates the basic features of the approach in the context of a system composed of a set of collocated components – that is, within the confines of a single virtual machine. The MORABIT infrastructure is able to influence and monitor life-cycle events such as component creation and migration etc., and is able to intercept inter-component invocations. In the longer term, a distributed version of the infrastructure needs to be developed and integrated into an enterprise middleware infrastructure supporting the distributed interaction of components (e.g. J2EE or .NET).

The so-called MORABIT infrastructure provides the basic run-time services needed to execute context-sensitive, resource-aware tests of components. These services are accessed via an API and an XML-based language which allows various specifications to be defined. This includes component configuration files and test requests. The basic philosophy of the infrastructure's design is to minimize the impact of the built-in testing feature on the normal application code. Therefore, although there is a programmatic API which allows programmers to explicitly control when tests are

executed and what reaction are performed, the goal is to allow run-time testing concerns to be separated to the greatest extent possible from normal application concerns. Thus, the responsibility for executing and reacting to tests is generally passed to the infrastructure via so-called "test request" specifications.

A test request specifies all the important parameters that the infrastructure needs to know in order to effectively perform the tests at run-time. In addition to the description of the test cases to be executed on a component this includes specifications of when (i.e. under what circumstances) a test should be executed. The currently supported test timing options include:

- Lookup-time: when a component first acquires a reference to another component,
- Call-time: when a client component calls a method of a server component,
- Topology-change time: when a component leaves or joins the network,
- Idle-time: when the load of the system drops below some specified threshold,
- Periodic: at fixed time intervals,

It is not only the tests themselves which are executed automatically, in many cases the system's reactions to the test results are enacted by the infrastructure as well. Several infrastructure-level reaction strategies are currently supported including:

- Shut down: in which the infrastructure shuts down one or more components of the system,
- Try next: in which the infrastructure tests the next available server component.

To enable users to best leverage the infrastructure, MORABIT has defined an accompanying method which explains how components should be developed and deployed. Based on mainstream component-development practices, the method covers all aspects of the component life-cycle, from specification and design through to implementation and deployment.

5. Run-Time Testing Examples

In this section we provide an example of the use of MORABIT to support run-time testing and show how it helps reduces the manual effort involved in validating the AH component at deployment-time. The example illustrates where quantitative testing might be used. We will focus on the register() operation which is implemented according to the collaboration diagram in Figure 4. The components that take part in this collaboration are Participant Manager, Mail Server, and Activity Logger. The collaboration diagram illustrates successful execution paths through the operation, but does not indicate the likelihood of a particular path being taken. To do this we used Markov chains to show the path probabilities. Figure 6 shows a Markov chain for the register() operation which illustrates the relative probabilities of the method invocations comprising the algorithm being executed at each at branching point.

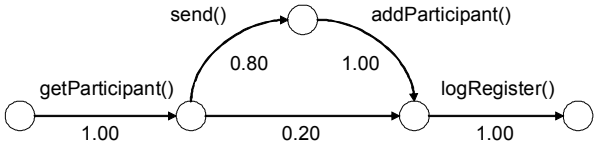

Figure 6. Markov Chain for register() algorithm

Table 1 shows the relative execution frequencies and the failure rates for each operation called from register(). The former can be calculated from the collection of Markov chains for all of Auction House's operations, while the latter are measured by performing quantitative tests of the server components at run-time.

operation	frequency	failure
ParticipantManager.addParticipant()	0.80	0.002
ParticipantManager.getParticipant()	1.00	0.001
MailServer.send()	0.80	0.100
ActivityLogger.logRegister()	1.00	0.005

Table 1. Invocation Frequencies and Failure Rates

The failure rate of 0.002 indicates that out of, say, 1000 calls the addParticipant() operation on averages is likely to fail 2 times. This frequency is called the probability of failure on demand (POFOD). It is important because it represents the probability of a failure being perceived by the user.

To calculate the POFOD of register() the execution likelihoods derived from the Markov chain are multiplied by the failure rates and the products are summed together. Given the data in Table 1, therefore, the probability of a given invocation of the register() method failing is thus -

((1.00)*0.001) + ((1.00*0.80)*0.100) +
((1.00*0.80*1.00)*0.002) + ((1.00)*0.005)
= 0.001 + 0.08 + 0.0016 + 0.005
= 0.0876

Thus, if the POFOD of each operation of a component can be determined by run-time testing the reliability of the component as a whole can be determined. The usage profile can be enhanced by

taking into account the distribution of actual parameter values for the operation parameters and also the probability of specific operation sequences (and thus component states) arising. However, for the sake of simplicity, these are not considered.

Without the MORABIT run-time testing technology, the only way to determine the reliability of the AH component is to measure it by running a statistically significant number of test cases on it at development-time, and measuring the distribution of any observed failures. These test cases must be distributed in accordance with the usage profile in order to get an accurate measure of the probability of the user perceiving an error.

5.1. Deployment-time Test on Lookup

As a first example we show how MORABIT can be used to test a server of the Auction House component when its reference is first acquired. If usage information is available for every operation of the AH component then, assuming the operation invocation distribution defined in the usage profile, it is possible to calculate invocation frequencies for each of the servers of the component. It is then, in turn, possible to use this information to calculate the acceptable POFOD levels for the individual operations of the individual server. For a given POFOD of the AH as a whole, this can be broken down to determine acceptability thresholds for each operation.

Having determined what reliability is required for the server components the next step is to define an appropriate set of test cases to test each server component. The test cases must be distributed according to the usage profile and their number must be high enough to get a statistically significant result. The final thing to do is to decide what reaction to take if the desired reliability cannot be delivered. One option, for example, is to shut down the component

The examples below are shown in the XML syntax used in a MORABIT test request which is passed to the MORABIT infrastructure.

```
<testRequest
   name="Auction House's TR for the MailServer"
   reliability="0.85"
   confidence="0.80"
   testTime="lookupTime"
   testReaction="shutdown">
   <testSuite
      name="for testing compliance"
      typeUnderTest="MailServer">
      <testCase>
         ...
      </testCase>
   </testSuite>
</testRequest>
```

Each component has such a test request which contains all information about the tests to be performed. Here, this test request belongs to the Auction House. Below the name of the test request the required reliability and confidence are listed. These are the values the testing component expects from its used services. The values are followed by an indication of when the test should be executed. "lookupTime" means that the potential new server is tested before the reference to a Mail Server is acquired.

From the result of the test runs, reliability and confidence are calculated. If the specified required values are not met, the specified test reaction will be performed. Here the component will be shutdown, meaning that it will not serve any further service requests. Less drastic action can also be chosen. Since the reason for the failure of the tests cannot be determined it might be possible that the tested service failed because of its required services.The final part in the test request contains the test cases. In our current implementation, the test cases are implemented as JUnit-like Java classes. Tests requests therefore contain the fully-qualified Java class name of each test.

5.2. Service-time Test on Replacement

When all components are deployed and all tests passed, the system starts delivering its service. From then on only service-time testing can be done. As an example of such service-time tests, here we show a test request that defines that the mail server is to be tested whenever the external server is swapped. Everything is basically the same except the test time when the testing should be executed

```
testTime="topologyChangeTime"
```

This indicates that the tests are performed whenever the environment changes, meaning whenever the mail server is replaced.

5.3. Service-time Self-test on Idle

In the final example we address the issues of resource-awareness. The MORABIT infrastructure constantly measures the resources of a device, in order to detect whenever "enough" resources are available to run tests in the background. As well as running tests for its required services, a component can also perform self-tests. A good time to do so this is when the device the component is running on is idle.

```
testTime="idleTime"
```

For self-testing the "type under test" needs to be set to the AH component as a whole rather than one of its servers. The test request thus has to be changed to

typeUnderTest="AuctionHouse"

Idle is defined relative to the device's resources. Therefore, the test request must also define what is considered to be "idle" for the resource in question:

<resource type="CPU"
load="below 10 out of 100" />

This indicates that if the resource "CPU usage" of the device drops below 10% the device is considered to be idle. When the test time is defined to be "idle" the infrastructure thus has a concrete threshold defining when to perform the tests. Besides CPU usage the MORABIT infrastructure is also capable of measuring memory consumption and bandwidth. All three can be combined to better adjust to the changing environment.

This approach has the advantage that unused resources are put to good use when they would otherwise be "wasted". Regular business functionality is not affected because tests are only performed when sufficient resources are available.

6. Related Work

Supporting the run-time verification of component-based systems though built-in testing is still a very specialized topic. While there have been several published approaches to "self" testing via built-in tests [10], [11], [12] these have all focused on enhancing development-time testing activities. To our knowledge the only previous project to have explicitly focused on run-time verification using BIT is the Component+ project [13], [14], on which the MORABIT project builds. The Component+ approach was in turn influenced by the original work of Wang et al [3].

MORABIT enhances Component+ in three important ways. First, it covers the full spectrum of run-time testing possibilities, including the contract testing and QoS testing approaches elaborated in Component+. For example, MORABIT allows components to define self-tests as well as server tests, and allows a larger range of test timing policies such as idle-time and random testing. Second, it allows built-in testing concerns to be separated, to the greatest extent possible, from the normal component development and deployment concerns. Based on the XML test request, the infrastructure can take over all responsible information for executing, evaluating, and reacting to tests. Last but not least, MORABIT takes into account the resources of the computer or the computers on which the tests are to be executed and evaluated. Since service-time tests are executed when applications are servicing requests, built-in tests must be executed under carefully controlled conditions, otherwise they run the risk of undermining the very thing they are intended to improve – the reliability of the system.

The built-in tests themselves draw heavily on accepted practices and theories in software testing [15]. The definition and execution of qualitative built-in test cases is based on the standard theories for defining defect-testing test cases such as equivalence class and boundary value analysis as well as algorithm-based coverage criteria such as path coverage. Similarly, the definition of quantitative test cases is based on widely known statistical testing techniques such as explained in [16] or those developed in the Cleanroom approach [17]. However, in both case subtle but important adaptations were necessary to tailor the approaches to the needs of run-time testing.

Applying these testing capabilities in a meaningful way in the context of components requires appropriate theories for the composition of properties and features. In particular, to reduce the level of traditional testing needed to verify component-based systems it is necessary to be able to derive some conclusions about correctness and/or reliability of systems based on the correctness and/or reliability of its components. This aspect of the MORABIT approach has been influenced by other component reliability work such as that of [2].

Built-in testing approaches of this kind overlap to a certain extent with service-level management and monitoring technologies such as that of Wang et al [18]. However, the overlap is small because built-in testing is focused on checking the conformance of components to their published interface, either in terms of correctness or reliability, and does not place a significant emphasis on fault tolerance. In contrast, the approach of Wang et al is focused on monitoring QoS characteristics of systems and emphasizes the explicit provision of counter measures. The two approaches are essentially complementary therefore.

7. Conclusion

In this paper we have presented the MORABIT approach for run-time testing which can help automate much of the traditional verification effort involved in building software systems from components. We have explained the basic rationale and philosophy behind the approach and demonstrated in terms of a case study how it helps verify the correctness and/or reliability of component-based systems at both deployment- and service-time. Performing automated tests at the component level not only reduces the amount of traditional system testing needed to attain a given level

of confidence in a system's reliability, it also helps to pin-point the location of faults, and thus reduces the effort involved in finding and removing them.

Although this technology is not tied to enterprise systems per se, the problems which it addresses become much acuter the larger and more complex a system becomes. The built-in testing technology developed in MORABIT therefore promises to be of particular benefit in the creation and integration of enterprise computing systems. As mentioned above, the approach essentially plugs a hitherto neglected gap in the spectrum of dependability techniques between classic development-time verification techniques on the one hand and enterprise service-level management and fault tolerance techniques on the other hand.

The traditional notions of verification and validation and their shoe-boxing into development- and run-time activities is no longer appropriate with modern component- and service-based development methods. Instead a more component-oriented notion of verification is needed in which the components of a system are checked according to their ability to fulfill their provided contract in terms of their required contracts. Thus, the verification of a system is essentially performed by validating the pairwise relationships between its components. In the spirit of Boehm we can characterize this new approach as answering the question, for each component – "do I have a connection to the right component" as opposed to "do I have the right connection to a component".

The development of our infrastructure is still at an early stage, but the initial prototype has enabled us to validate the core tenets of the approach. The next step is to enhance the system to work with distributed middlewares and apply it to more case studies.

8. Acknowledgements

We are grateful to the Landesstiftung of the state of Baden-Württemberg for the funding of this work. Rainer Malaka and Matthias Merdes thank the Klaus Tschira Stiftung for its support.

References

[1] C. Szyperski, *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, London, 1998.

[2] R. H. Reussner, I. H. Poernomo, and H. W. Schmidt, "Contracts and Quality Attributes for Software Components", *Proc. 8th Int'l Workshop on Component-Oriented Programming (WCOP'03)*, Darmstadt, Germany, June 2003.

[3] Y. Wang, G. King, and H. Wickburg, "A method for built-in tests in component-based software maintenance", *IEEE International Conference on Software Maintenance and Reengineering (CSMR'99)*, IEEE Computer Science Press, 1999, pp. 186-189.

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

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

[6] B. Boehm, "Verifying and validating software requirements and design specifications", *IEEE Software*, 1984, Vol 1.

[7] A. A. Abdel-Ghaly, P. Y. Chan, and B. Littlewood, "Evaluation of Competing Software Reliability Predictions", *IEEE Trans. Software Eng,* 1986, 12(9).

[8] J. Musa, "Operational profiles in software-reliability engineering", *IEEE Software*, 1993, 10(2), pp. 14–32.

[9] R. Hamlet, "Random Testing", *Encyclopedia of Software Engineering*, Wiley, Second Edition, 2002.

[10] Y. L. Traon, D. Deveaux, and J.-M. Jezequel, "Self-testable components: from pragmatic tests to design-to-testability methodology", *Technology of Object-Oriented Languages and Systems (TOOLS)*, IEEE Computer Society Press, 1999, pp. 96-107.

[11] S. Beydeda, V. Gruhn, "Black- and White-Box Self-testing COTS Components", *Software Engineering and Knowledge Engineering (SEKE)*, Banff, 2004.

[12] L. Mariani, M. Pezze, and D. Willmor, "Generation of Integration Tests for Self-Testing Components", *FORTE Workshop*, 2004, pp. 337 - 350

[13] C. Atkinson, H.-G. Groß, F. Barbier, Chapter 8, "Component Integration through Built-in Contract Testing in Component-Based Software Quality: Methods and Techniques", *Lecture Notes in Computer Science #2693*, Springer, 2003, pp. 159-183.

[14] H.-G. Gross, *Component-Based Software Testing with UML*, Springer, 2004.

[15] B. Beizer, *Software Testing Techniques*, Van Nostrand Rheinold, New York, 1990.

[16] J. D. Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper*, Osborne/McGraw-Hill, 1998.

[17] Deck, M., "Cleanroom and object-oriented software engineering: A unique synergy". *Proceedings of the Eighth Annual Software Technology Conference*, Salt Lake City, USA, April 1996.

[18] G. Wang, A. Chen, C. Wang, C. Fung, and S. A. Uczekaj, "Integrated Quality of Service (QoS) Management in Service-Oriented Enterprise Architectures", *Enterprise Distributed Object Computing (EDOC)*, Monterey, USA, 2004, pp. 21-32