Electronic version of an article published in **Haasis, K.; Heinzl, A.; Klumpp, D. (Hrsg) Aktuelle Trends in der Softwareforschung. Tagungsband zum doIT Software-Forschungstag am 13. Juli 2006, pp. 35 - 46**

Copyright © [2006] dpunkt.verlag

http://www.dpunkt.de/

Laufzeittest mobiler und komponentenorientierter Software

D. Suliman[1], L. Borner[1], M. Merdes[2], D. Brenner[3]
(1) Universtität Heidelberg, Lehrstuhl Software Systeme, (2) EML Research gGmbH, (3) Universität Mannheim, Lehrstuhl für Softwaretechnik

Abstract

Die Komplexität von Software-Systemen wird durch Komponenten-orientierung und Mobilität deutlich erhöht. Damit steigt auch das Risiko von Fehlern. Das Ziel des Forschungsprojekts MORABIT ist es, dieses Fehlerrisiko zu verringern und somit die Qualität von komponentenorientierter Software besonders in mobilen Systemen zu erhöhen. Dies wird durch die Ausführung von Tests zur Laufzeit unter Berücksichtigung der in mobilen Systemen beschränkten Ressourcen erreicht. Diese Ausführung wird unterstützt durch eine im Forschungsprojekt MORABIT entwickelte Infrastruktur. Der Beitrag gibt eine Übersicht über die grundlegenden Konzepte in MORABIT.

1 Einleitung

Fehlerhafte Programme verursachen viel Ärger, große Kosten und immer wieder erheblichen Schaden. Dieses Fehlerrisiko wird durch die Komplexität komponentenorientierter, mobiler Systeme erhöht, da die Komponenten von verschiedenen Herstellern geliefert und erst zur Entwicklungszeit oder – bei Mobilität – zur Laufzeit zusammengesetzt werden.

In dem von der Landesstiftung Baden-Württemberg finanzierten Forschungsprojekt MORABIT (Mobile Resource Adaptive Built-In Tests) entwickeln die Projektpartner Universität Heidelberg, EML Research gGmbH und die Universität Mannheim eine Methode, um dieses Fehlerrisiko durch Tests zur Laufzeit besonders in mobilen Systemen zu verringern. Als Basis für diese Methode wird die Testtechnik Built-In Test (BIT) [Com99] verwendet. BIT erfüllt bereits einige der Anforderungen für das Testen von Systemen zur Laufzeit [Sul05], ist aber bisher noch nicht realisiert und auch nicht auf mobile Systeme angepasst. In MORABIT wurden Konzepte zur Anreicherung der Komponenten mit Laufzeittestfällen erforscht. Die daraus entstehenden Leitlinien ermöglichen es sowohl dem Komponentenentwickler als auch dem Komponentenbenutzer, in einer flexiblen, aber systematischen Art and Weise eigene Testfälle hinzuzufügen Die Ausführung der Testfälle wird dann von der im Projekt entwickelten Infrastruktur gesteuert. Diese berücksichtigt dabei auch die Ressourcen, die auf den mobilen Endgeräten nur in beschränktem Umfang zur Verfügung stehen.

In diesem Beitrag werden die bisherigen Ergebnisse von MORABIT und die bereits realisierte Infrastruktur vorgestellt. Dabei wird auf mögliche Probleme, die in der Kommunikation zwischen Komponenten auftreten können, näher eingegangen. Hierfür beschreiben wir in Kapitel 2 ein Szenario und identifizieren mögliche Fehlerquellen in der Komponenteninteraktion. In Kapitel 3 erklären wir die Grundkonzepte des MORABIT Ansatzes, beschäftigen uns mit den Laufzeittestfällen und der Infrastruktur sowie den Prinzipien der Ressourcenverwaltung. In Kapitel 4 werden die Erkenntnisse kurz zusammengefasst.

2 Auktionsszenario

Um die entwickelten Methoden und Konzepte für das Testen von mobilen Systemen zur Laufzeit zu evaluieren, verwenden wir ein Auktionsszenario. Dies könnte zum Beispiel der Fischmarkt im Hamburg der Zukunft sein. AuktionsteilnehmerInnen können dort mit Hilfe von Handys oder PDAs[1] Gebote abgeben, um die gewünschte Menge Fisch zu erwerben. Sobald sie den Markt betreten, werden ihre mobilen Geräte automatisch in das existierende lokale Netzwerk des Fischmarkts mit sämtlichen BieterInnen und VerkäuferInnen eingegliedert. Dies geschieht im Hintergrund und für die TeilnehmerInnen vollständig transparent. Anschließend kann jedeR TeilnehmerIn um den Preis für die gewünschten Waren mitbieten. Hierzu werden die Gebote mit Hilfe des mobilen Endgerätes abgegeben. Erhält jemand den Zuschlag, wird der vereinbarte Betrag von seinem/ihrem Bank-Konto abgebucht. Sämtliche Geldtransaktionen laufen über eine Bank, die sich ebenfalls im Netzwerk befindet.

Das Bieten für und das Anbieten von Waren sowie die Überweisung wird durch Softwarekomponenten realisiert. Diese Softwarekomponenten befinden sich auf den einzelnen Geräten (z.B. Handy oder PDA) und führen dort ihre geforderte Funktionalität aus. Die Komponenten müssen miteinander kommunizieren, um ihren Zweck zu erfüllen. In Abbildung 1 ist das Netzwerk mit seinen enthaltenen Komponenten für einen Fischmarkt zu erkennen. Es zeigt die TeilnehmerInnen mit ihren PDAs (auf der rechten Seite), die verschiedenen Anbieter (repräsentiert durch das Auktionshaus in der Mitte) und weitere Komponenten, die für das erfolgreiche Abschließen des Handels notwendig sind (z.B. die Bank auf der linken Seite).

Die mobilen Softwarekomponenten des „Fischmarkt“-Systems wurden nicht, wie in einem „traditionellen“ Software-System üblich, von einem Hersteller entwickelt und vertrieben, sondern von vielen verschiedenen. Im „schlimmsten Fall“ bedeutet das, dass alle Komponenten im Netzwerk von verschiedenen Softwareentwicklungsfirmen erstellt wurden. Somit muss jegliche Interaktion zwischen den Softwarekomponenten in diesem Szenario als potenzielle Fehlerquelle eingestuft werden. Die Komponenten könnten zwar syntaktisch übereinstimmen, d.h. dass ihre Schnittstellen zueinander kompatibel sind, aber das semantische Verständnis dieser Schnittstellen könnte trotzdem verschieden sein. Ein solches „Missverständnis“ könnte zum Beispiel durch

[1]. Personal Digital Assistant

die Verwendung von unterschiedlichen Interpretationen der Mengenangaben entstehen. Während eine Komponente die empfangene Menge als Kilogramm interpretiert, könnte eine andere Komponente die Menge in Gramm oder gar in Tonnen übernehmen. Die Folgen würden zu einem Verlust von Geld oder Waren führen, entweder für den/die BieterIn oder den/die AnbieterIn. Ein weiteres mögliches Problem könnte in der Kommunikation zwischen Auktionshaus und der Bank entstehen. So könnte die Bankkomponente die Parameter in folgender Reihenfolge „{Empfängerkontonummer, Senderkontonummer}“ erwarten, aber die Auktionshauskomponente liefert sie in umgekehrter Reihenfolge. In einem solchen Fall wäre die Transaktion erfolgreich durchgeführt worden, aber in verkehrter Richtung.

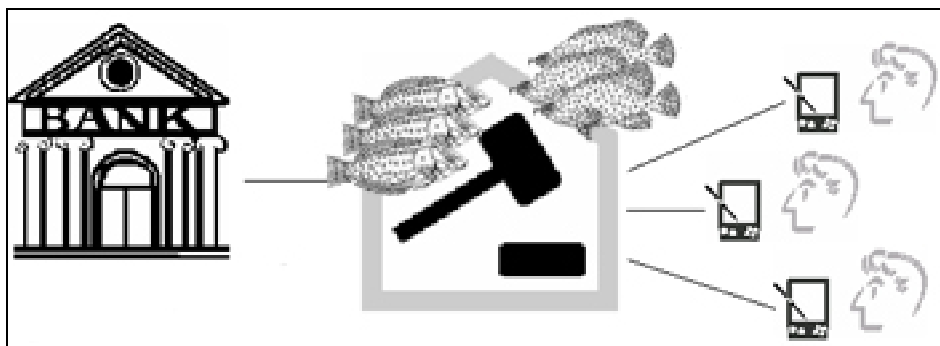

Abb. 1.: Mobiles Auktionsszenario eines Fischmarkts

Nachfolgend werden kurz die von uns identifizierten Arten von Fehlerquellen in mobilen, komponentenbasierten Softwaresystemen vorgestellt und anhand des Beispielszenarios erläutert.

2.1 Fehlerquellen in mobilen komponentenbasierten Systemen

Adhoc entstehende Systeme, wie im obigen Beispielszenario beschrieben, lassen sich nur schwer zur Entwicklungszeit testen. Um die fehlenden Tests zur Laufzeit „nachholen“ zu können, muss sich ein Tester der verschiedenen Fehlerquellen innerhalb eines solchen Systems bewusst sein. Nachfolgend sollen diese verschiedenen Fehlerquellen in der Komponenteninteraktion aufgezeigt werden.

Eine mögliche Fehlerquelle ist das Vertauschen von Parametern, wie oben bereits in dem Beispiel mit den vertauschten Kontonummern gezeigt. Daran schließt sich direkt eine weitere Fehlerquelle an: die fehlerhafte Interpretation der Parameter. Dies kann mit dem oberen Beispiel der falsch verwendeten Gewichtsangaben erläutert werden. Beide Komponenten arbeiten mit Zahlen. Jedoch werden die Zahlen jeweils unterschiedlich interpretiert. Die Vertauschung und die unterschiedlichen Interpretationen können sowohl bei Eingabeparametern als auch bei Ausgabeparametern einer Komponente auftreten.

Im Laufe der Kommunikation zwischen zwei oder mehreren Komponenten kann es unter Umständen auch zu einem Fehler führen, wenn eine Funktionalität aufgerufen wird, die in dem vorhandenen Datenzustand oder im vorgeschriebenen Protokoll nicht erlaubt ist. Das Problem könnte darin bestehen, dass die beteiligten Komponenten ein unterschiedliches Verständnis der verwendeten Protokolle haben, oder eine Komponente falsche Annahmen über die Zustände der verwendeten Komponente macht. Auf solche falschen Aufrufe muss die aufgerufene Komponente entsprechend reagieren. Aber auch hier kann es zu Fehlern kommen. Die Ausnahmebehandlung von nicht erlaubten oder nicht möglichen Aktionen muss von der ausführenden Komponente durchgeführt werden.

Eine weitere Fehlerquelle in der Komponenteninteraktion können auch zu lange Antwortzeiten sein. Unter Umständen benötigt eine Komponente die geforderte Antwort sehr schnell. Aber aufgrund von Überlastung der verwendeten Komponente oder des Netzwerks bekommt sie ihre Antwort zu spät oder gar nicht.

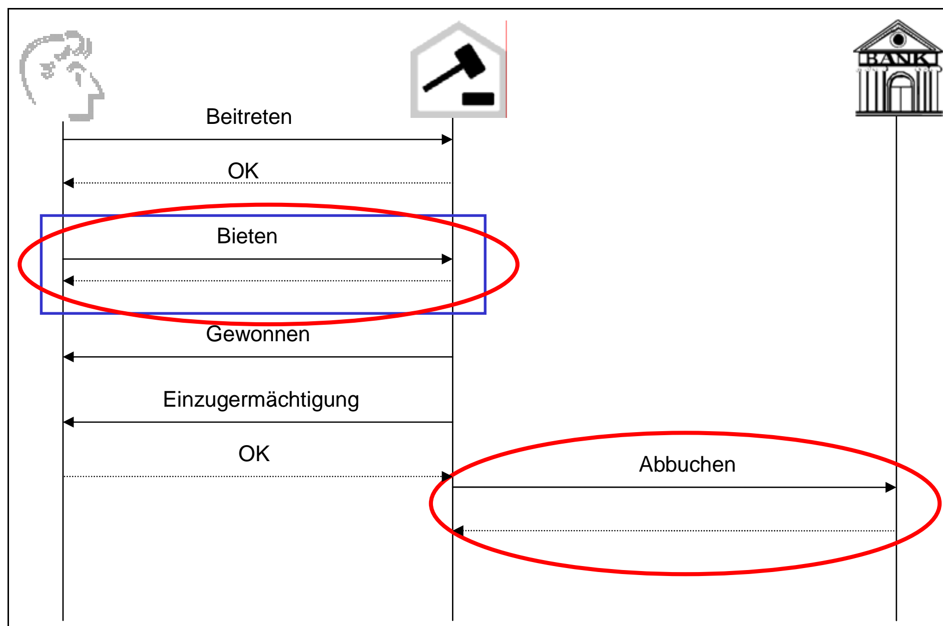

Abb. 2.: Kritische Interaktionen im Auktionsszenario

In Abbildung 2 sind verschiedene Interaktionen zwischen den Komponenten des Auktionsszenarios abgebildet. Sie beschreibt eine typische Sequenz von Interaktionen einer Komponente auf dem Fischmarkt. In einem ersten Schritt tritt die Bieterkomponente der Auktion bei. War dieser Vorgang erfolgreich, kann der/die BesitzerIn der Komponente anschließend an der Auktion teilnehmen. Er/Sie kann Gebote abgeben, wird unter Umständen überboten, erhöht sein/ihr Gebot usw. Am

Ende wird ihm/ihr mitgeteilt, dass er/sie die Auktion gewonnen hat, die Einzugsermächtigung wird eingefordert und abschließend der geforderte Betrag vom Konto des/der GewinnerIn abgebucht. Die kritischen Interaktionen wurden in der Abbildung durch Ovale gekennzeichnet. Zum einen muss die Interaktion, die das Bieten realisiert überprüft werden. Bei Fehlern in diesen Interaktionen kann es zum Verlieren der Auktion (aus Sicht des Kunden) oder zur Nichtvollendung der Auktion (aus Sicht des Verkäufers) kommen. Daher muss überprüft werden, ob die richtige Menge an Geld geboten worden ist und ob das Gebot unverfälscht und rechtzeitig bei der Auktion angekommen ist. Weiterhin muss sichergestellt werden, dass alle Gebote bei der Auktion eintreffen. Eine Interaktion, die zum Verlust von Geld führen kann, ist die Abbuchung. Hierbei können die oben beschriebenen Fehlerquellen, wie das Vertauschen von Parametern oder die Nichtbeachtung von Datenzuständen auftreten.

Viele der oben beschriebenen Fehler können durch Laufzeittests aufgedeckt werden. Somit kann die Zuverlässigkeit der Komponenten und des gesamten Systems erhöht werden.

3 Die MORABIT Idee

In diesem Kapitel werden wir kurz die Grundkonzepte des MORABIT Ansatzes vorstellen. Im ersten Abschnitt stellen wir die zu Grunde liegende Built-In Testtechnik vor und beschreiben die daraus resultierende MORABIT Testtechnik. Anschließend gehen wir näher auf die Laufzeittestfälle und die Ressourcenbeschränktheit für mobile Systeme ein und wie sie in MORABIT unterstützt werden.

3.1 Die Built-In Testtechnik

Im BIT Ansatz besitzt jede Komponente zusätzlich zu ihren normalen funktionalen Schnittstellen eine Schnittstelle zur Unterstützung von Tests, die so genannte „Testschnittstelle“ (Testable Interface). Mit Hilfe dieser Schnittstelle kann der Tester (ein Mensch oder eine andere Komponente) interne Zustände dieser Komponente setzen oder abfragen. Jede BIT-Komponente besitzt zwei Modi: Einen „Normal Modus“, in dem die Komponente wie jede andere Komponente agiert und einen „Testmodus“, in dem der Tester interne Zustände setzen bzw. abfragen kann. Eine Komponente kann Testfälle in die „Produktionsumgebung“ mitbringen. Diese Tests können dem Selbsttest dienen, d.h. mit Hilfe dieser Tests kann die Komponente sich selbst testen [Wan99] um zu prüfen, ob sie in einer neuen Umgebung richtig funktioniert. Die mitgebrachten Tests können aber auch dazu dienen, einen der Server[2] zu testen, um zu prüfen, ob die Komponente und ihr Server dieselbe Vorstellung von dem Service haben (Contract Test) [Gro04]. In MORABIT werden beide Ansätze der BIT Technik unterstützt. In Abbildung 3 sind zwei Komponenten A und B vorhanden. Komponente A braucht den Service von Komponente B, und will, bevor sie den Server

[2]. Ein Server ist eine Komponente, dessen Service von einer anderen Komponente benutzt wird

B benutzt, testen, ob sie dasselbe Verständnis von dem Service mit Komponente B teilt (Contract-Test). Jede Komponente kann auch Tests für sich selbst (Self-Test) besitzen, um zu prüfen, dass sie ihre angebotenen Services (Provided Interface) in der neuen Umgebung erfüllen kann. Da die Tests zur Laufzeit ausgeführt werden, müssen alle Testschritte automatisch ausgeführt werden, d.h. es muss von Anfang an der Testzeitpunkt („Wann soll der Test ausgeführt werden?“) und die Testreaktion („Wie wird auf die Testergebnisse reagiert?“) festgelegt werden.

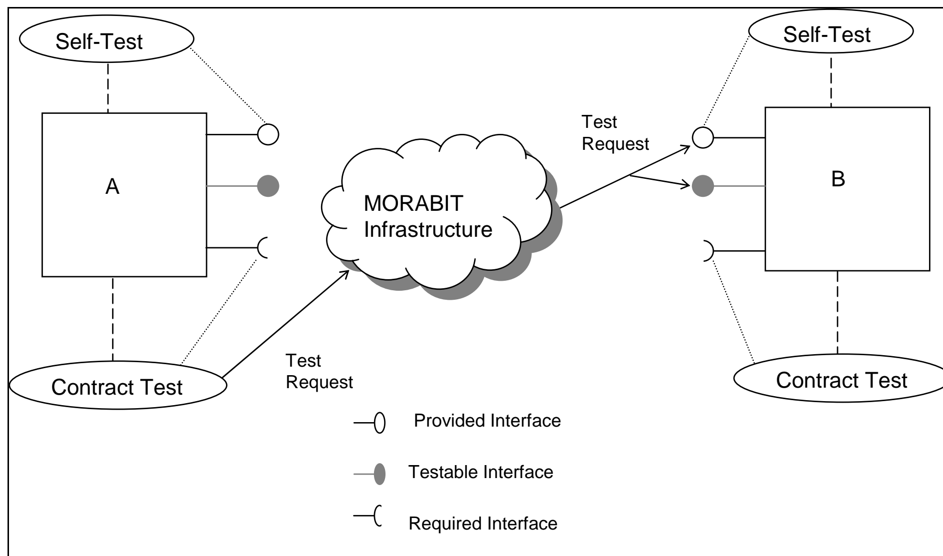

Abb. 3.: Der MORABIT Ansatz

Die Informationen über Testzeitpunkt und Testreaktion sind in MORABIT in den Testanfragen gekapselt (vgl. Kapitel 3.2). Durch die Testanfragen gelangen alle für den Test notwendigen Informationen zur Infrastruktur. Die MORABIT Infrastruktur kann anhand dieser Informationen die Tests ausführen (In Abbildung 3 vermittelt Komponente A alle für den Contract-Test benötigten Informationen an die Infrastruktur durch eine Testanfrage). Der Testzeitpunkt hängt von dem in der Testanfrage spezifizierten Testzeitpunkt und vom aktuellen Ressourcenstatus ab. Dies bedeutet, dass bei ausreichend freien Ressourcen der Test ausgeführt wird (s. Kapitel 3.3). In Abbildung 3 hat Komponente A eine Testanfrage, um zu prüfen, dass ihr Server (Komponente B) ihren Erwartungen entspricht. Die Infrastruktur führt den Test aus. Am Ende des Tests werden die in der Testanfrage spezifizierten Reaktionen ausgeführt. Während des Testprozesses muss die Trennung zwischen den durch den Testprozess manipulierten Daten und den durch normale Prozesse manipulierten Daten gewährleistet werden. Ist dies nicht der Fall, besteht die Gefahr, dass falsche Testergebnisse geliefert oder reale Daten durch die Tests manipuliert werden. Die

Auktionskomponente kann z.B. nicht einfach Geldtransaktionen zwischen zwei realen Konten ausführen um zu testen, ob das „Abbuchen“ bei der Bankkomponente richtig funktioniert. Um Kollisionen zwischen echten Daten und Testdaten zu verhindern, werden in MORABIT mehrere Ansätze unterstützt. Auf der einen Seite können die EntwicklerInnen der Komponente für die Trennung zwischen Testdaten und realen Daten Testsessions in der Testschnittstelle anbieten. Der Tester kann somit durch Aufrufen dieser Operationen die Kollision zwischen Testdaten und realen Daten vermeiden. Zusätzlich kann eine getestete Komponente, die keine Testsessions anbietet, von der MORABIT Infrastruktur unter Umständen geklont werden. Die Testfälle werden dann auf dem Klon ausgeführt. Ob eine Komponente geklont werden kann, hängt von zwei Aspekten ab. Zum einen kann der/die EntwicklerIn der Komponente das Klonen dieser Komponente verbieten. Zum anderen könnten zu wenige Ressourcen für das Klonen zur Verfügung stehen. In beiden Fällen kann die Komponente nicht getestet werden und der Testvorgang wird abgebrochen.

3.2 Testanfrage

Jede Komponente innerhalb der MORABIT Umgebung besitzt eine Konfigurationsdatei. Diese Datei enthält sämtliche Konfigurationsinformationen der Komponente und alle ihre Testanfragen. Eine Testanfrage enthält die notwendigen Daten, um die Tests auszuführen. Sie ermöglicht der Komponente, alle testbezogenen Informationen in einem flexiblen Format an die Infrastruktur zu übermitteln. Abbildung 4 zeigt ein Beispiel einer Testanfrage in Form eines XML - Dokuments. Die Testanfrage ist in der Konfigurationsdatei der Auktionskomponente enthalten, und dient dazu, die Bankkomponente zu testen. Neben dem Namen und einer textuellen Beschreibung enthält die Testanfrage den Testzeitpunkt (Test Time). Im Beispiel in Abbildung 4 wird zur „Call Time“ getestet, d.h. bevor die Auktionskomponente die „Abbuchen“ Operation bei einer Bankkomponente aufruft, wird dieser Test ausgeführt. In MORABIT werden mehrere Testzeitpunkte unterstützt, z.B. periodisches Testen oder Testen bei topologischen Änderungen, d.h. wenn sich die Struktur des Systems ändert, also eine Komponente entfernt wurde oder eine neue hinzukommt, wird der Test ausgeführt. Bei der in Abbildung 4 dargestellten Testanfrage wird bei einem erfolglosen Test die Auktionskomponente abgeschaltet (Testreaktion „shutdown“). In anderen Situation können andere Testreaktionen sinnvoll sein, z.B. eine andere Komponente, die die gesuchte Funktionalität anbietet, kann getestet und anschließend verwendet werden (try-next-Strategy), oder die Testerkomponente könnte einen anderen Algorithmus einsetzen, der zwar langsamer ist, aber dafür den korrupten Server nicht benutzt.

In einer Testanfrage können mehrere Testfälle enthalten sein, diese sind gekapselt innerhalb einer Testsuite. Für jeden Testfall ist der Name, die Beschreibung und der auszuführende Bytecode des Testfalls spezifiziert. Die Testsuite spezifiziert darüber hinaus, welcher Komponententyp getestet werden soll. In Abbildung 4 soll die Bankkomponente getestet werden. Durch Ändern der Testanfrage innerhalb der

Konfigurationsdatei einer Komponente können neue Testfälle hinzugefügt oder alte Testfälle entfernt werden. Testzeitpunkt und Testreaktion können auch nachträglich geändert werden. Dies geschieht ohne den Quelltext der Komponente ändern zu müssen.

```
<testRequest
    name="TR for Bank"
    confidence="0.5"
    reliability=„1.0"
    testTime="CallTime"
    testReaction="Shutdown">
    <description>description for: TR for Bank</description>
    <testSuite
        name="TS for Bank"
        typeUnderTest="org.morabit.app.interfaces.bank.Bank">
        <description>description for: TS for Bank</description>
        <testCase
            name="Test Case for the Bank (transfer Money)"
            implementingClass="org.morabit.hd.testing.bank.BankTestCase">
            <description> tests the transfer money operation by the bank component </description>
        </testCase>
    </testSuite>
</testRequest>
```

Abb. 4.: Beispiel einer Testanfrage

3.3 Ressourcenverwaltung der MORABIT Infrastruktur

Ein besonderes Kennzeichen mobiler Systeme ist die relative Ressourcenarmut der typischerweise verwendeten Endgeräte. Zwar entwickelt sich die Hardware auch in diesem Bereich stetig weiter, im Vergleich zu vollwertigen PCs existiert jedoch eine inhärente Beschränkung dieser Ressourcen. Innerhalb von MORABIT wird daher ein besonderes Gewicht darauf gelegt, die beschriebenen Laufzeittests im Einklang mit der Ressourcensituation durchzuführen.

Der Begriff der Ressource kann im Prinzip auf alle physischen oder logischen Entitäten angewandt werden, die zur Durchführung einer Aufgabe innerhalb eines Computersystems notwendig sind. In der Praxis beschränkt man sich jedoch meist auf einige wenige wichtige Ressourcen. Klassische Beispiele, die auch bei herkömmlichen nicht-mobilen Systemen relevant sind, sind der Prozessor und der Hauptspeicher eines Rechners. Bei mobilen Systemen kommen dann insbesondere noch die verbleibende Batterieladung und die Bandbreite der Netzwerkverbindung zu anderen Knoten des (drahtlosen) Netzwerks hinzu. In MORABIT konzentrieren wir uns zunächst auf diese vier exemplarischen Ressourcen, da die angestrebten Erkenntnisse über die ressourcenabhängige Ausführung von Tests so bereits gewonnen werden können.

Um die Ausführung von Tests von der Ressourcensituation abhängig machen zu können, muss der Status der genannten Ressourcen der MORABIT Infrastruktur

bekannt sein. Dazu muss zu einem gegebenen Zeitpunkt die jeweilige Auslastung dieser Ressourcen gemessen werden können. Diese Informationen werden beim Betriebssystem verwaltet, sind also inhärent plattformabhängig. Wie in Abbildung 5 schematisch dargestellt, lassen sich bei der Ressourcenverwaltung mehrere logische Schichten unterscheiden. In der untersten Schicht befindet sich die Ressource auf der Ebene des Betriebssystems. Da man sich bei der Programmierung der Infrastruktur (auf der obersten Ebene) nicht für derartige Details interessiert, gibt es in der Mitte eine Schnittstellenebene zwischen Betriebssystems- und Anwendungsschicht.

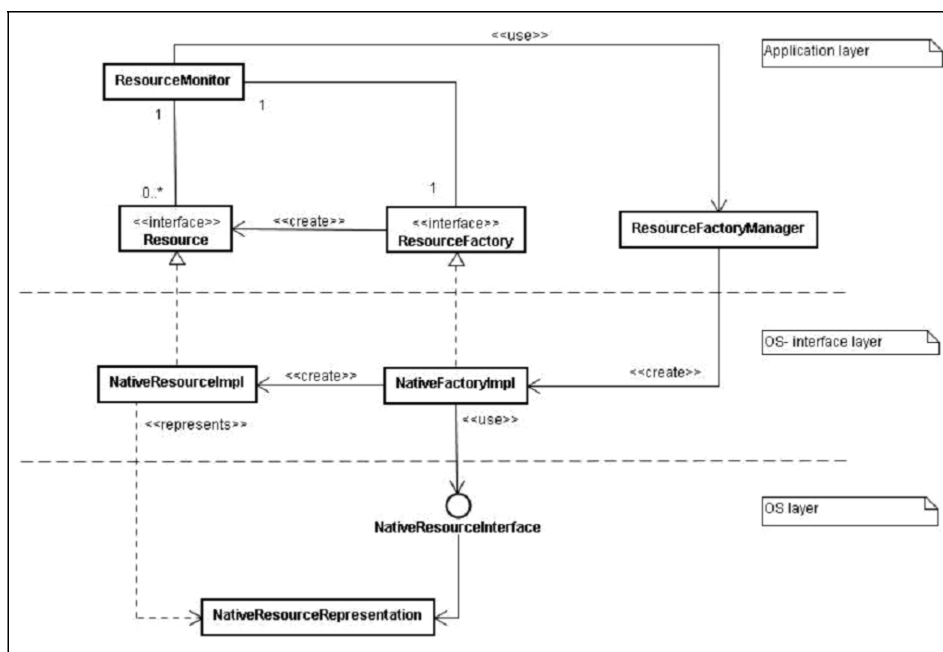

Abb. 5.: Schichtenarchitektur der Ressourcenüberwachung

Wenn auch die Infrastruktur über die Ressourcenmonitore der obersten Schicht *programmatisch* auf den Ressourcenstatus zugreift, so ist eine graphische Darstellung der Ressourcensituation über die GUI sowohl für den/die EntwicklerIn von Komponenten als auch für den/die TestadministratorIn sehr nützlich. Daher stellt die MORABIT Infrastruktur die in Abbildung 6 gezeigte Sicht auf die Ressourcensituation zur Verfügung. Hiermit kann die prozentuale Auslastung der beschriebenen vier Ressourcen beobachtet werden. Darüber hinaus lassen sich interaktiv Schwellenwerte für die einzelnen Ressourcen definieren, welche bei der Testausführung verwendet werden können.

Da die Ressourcensituation – wie bereits erwähnt – in typischen mobilen Systemen oft kritisch ist, muss die Steuerung der Testausführung ressourcenabhängig sein.

Andernfalls würden die beschriebenen Vorteile der Laufzeittests wieder verloren gehen bzw. die Ausführung der Tests oder im schlimmsten Fall der Basisfunktionalität nicht mehr möglich sein. Daher wurde im Rahmen von MORABIT eine Reihe – zum Teil sehr komplexer – Strategien für die ressourcenabhängige Testausführung definiert. Exemplarisch soll hier die Schwellenwert-basierte Strategie erklärt werden. Hierbei wird für jede Ressource ein Schwellenwert definiert, was z.B. interaktiv über die oben beschriebene GUI geschehen kann. Steht nun zur Laufzeit zu einem gewissen Zeitpunkt eine Testanfrage an, so wird geprüft, ob die momentane Last der benötigten Ressourcen geringer ist als der vorher definierte Schwellenwert. Nur wenn dies der Fall ist, darf eine Testanfrage ausgeführt werden. So kann auf einfache Weise garantiert werden, dass keine Tests zu einem Zeitpunkt ausgeführt werden, der bezüglich der Ressourcensituation ungeeignet ist und daher die Basisfunktionalität des Systems beeinträchtigen könnte.

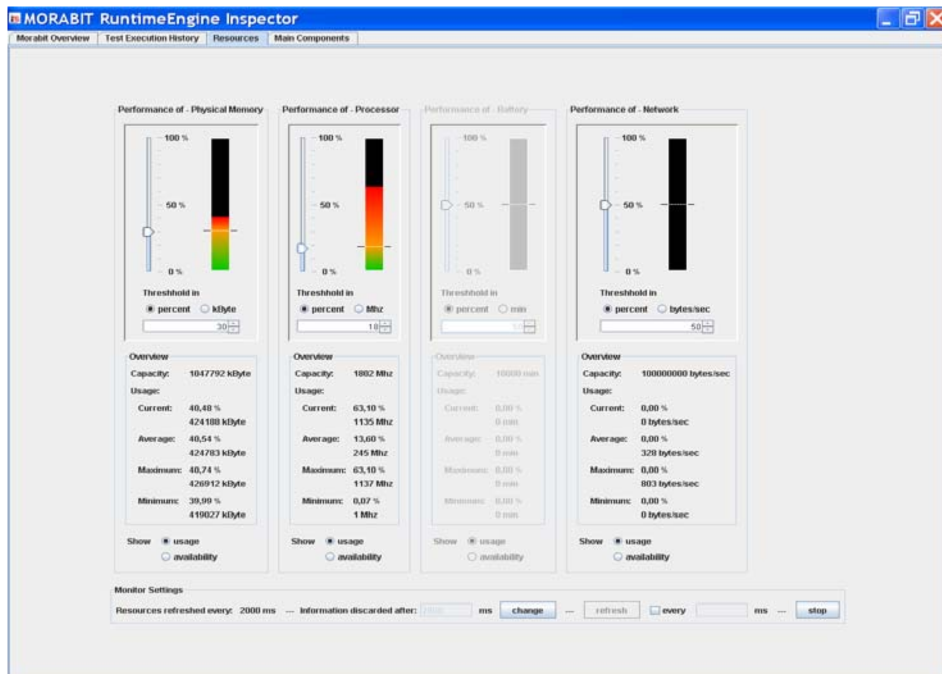

Abb. 6.: GUI zur Ressourcenüberwachung

4 Zusammenfassung

In diesem Artikel wurden erste Ergebnisse des Forschungsprojekts MORABIT vorgestellt. Eine im Projekt entwickelte Infrastruktur ermöglicht es Software-Komponenten, durch das Testen zur Laufzeit potenzielle Missverständnisse in ihren

Interaktionen aufzudecken und auf die Ergebnisse zu reagieren. Dabei berücksichtigt die Infrastruktur die auszuführende Funktionalität und die zur Laufzeit von mobilen System vorhandene Ressourcenbeschränktheit.

References

[Gro04] H.G. Gross: Component –Based Software Testing with UML. Germany. 2004

[Sul05] D. Suliman, B. Paech, and L. Borner: Testing Mobile Component Based Systems. NET.OBJECTDAYS Proceedings. 2005

[Com99] Component + Project, http://www.component-plus.org

[Wan99] Y. Wang, G. King, and H. Wickburg: A method for built-in tests in component-based software maintenance. IEEE Computer Science Press. 1999