Ubiquitous RATs: How Resource-Aware Run-Time Tests can improve Ubiquitous Software Systems

M. Merdes[1], R. Malaka[1,4], D. Suliman[2], B. Paech[2], D. Brenner[3], and C. Atkinson[3]

[1]EML Research gGmbH, Heidelberg, Germany,
[2]University of Heidelberg, Institute of Computer Science,
[3]University of Mannheim, Institute of Computer Science,
[4]University of Bremen, Center for Computing Technologies (TZI)

Abstract. In this paper we describe a new approach for increasing the reliability of ubiquitous software systems. This is achieved by executing tests at run-time. The individual software components are consequently accompanied by executable tests. We augment this well-known built-in test (BIT) paradigm by combining it with resource-awareness. Starting from the constraints for such resource-aware tests (RATs) we derive their design and describe a number of strategies for executing such tests under resource constraints as well as the necessary middleware. Our approach is especially beneficial to ubiquitous software systems due to their dynamic nature – which prevents a static verification of their reliability – and their inherent resource limitations.

1 Introduction

In Pervasive and Ubiquitous Computing, human users are confronted with a high number of computing devices and services. The ensuing paradigm shift from personal to ubiquitous computing empowers people to access many services anywhere and anytime. Many researchers sketched scenarios in which modern computer users benefit from such a federated infrastructure [29] where various devices and services can build ad-hoc ensembles and self-organize in order to provide added value.

However, it is not possible anymore for a human user to test each component and verify that all ensemble players interact in the desired way. Let us consider for instance a scenario where a user carries a shopping assistant on a PDA that may interact with his refrigerator and its content as well as with the shelves in a store in order to advise the user which goods to buy and to compare the prices. But when connecting to the supermarket services, the user cannot be sure that they actually provide the services the PDA software expects. The provider of the components in the supermarket might (deliberately or not) advice the user to buy too many or too expensive goods.

Future UbiComp scenarios consist of an arbitrary number of users, devices, and services of unknown origin and implementation. From a software engineering point of view these services are realized as composable software components [25] and each of these components contributes to the overall system behavior. However, the *emergent* ensemble of system components (software and hardware) cannot always be

planned in advance – let alone be subjected to traditional integration testing. Therefore, the question of how to make sure that interacting components actually fit together correctly at run-time becomes a crucial question for the whole paradigm. In particular, syntactic and meta-description of components and services are not sufficient to verify what a component actually does. Therefore, run-time testing is a necessary requisite for stable and trusted interactions between UbiComp components. Such tests need to be done *at run-time* since components can connect at any time and may not be known before. However, such run-time testing can be very resource-intense. In particular, when each component may test any other component at any time, the load on some devices can become high potentially rendering the system unusable. Therefore, new methods are necessary for designing resource-aware run-time tests (RATs) for UbiComp systems which are often deployed on inherently resource-restricted devices.

So far, systematic testing has been studied surprisingly little from a ubiquitous computing viewpoint. In the software engineering community there is a huge body of literature [2] covering all aspects from unit to integration, systems, acceptance, reliability, and usability testing. While the ubiquitous computing community has a clear focus on acceptance testing and user evaluation studies [15], [19] our work can be thought of as an integration testing approach to ubiquitous software systems. We view this run-time integration testing approach to ubiquitous systems construction as a contribution to systems support [11]. Morla and Davies [20] present a test and simulation environment for location-based services (LBS) systems which covers some testing aspects but based on simulating location and networks from a systems testing rather than from an integration testing point of view. While the Speakeasy approach [9] supports run-time *integration* of components or services with limited a priori knowledge it does not include notions of run-time *testing* or test-based reliability.

Although not the focus of this paper the work done in the component software field is generally important to our approach. While Szyperski's seminal work [25] gives a comprehensive overview Gao et al. focus on testing component-based software [12]. Our approach to include tests with components and to execute these tests at run-time is not in itself new. Such built-in tests (BITs) have been proposed by Wang [28] and further developed by, e.g. Gross [14] and Vincent et al. [26]. Historically, this approach has focused on components in standard enterprise systems with fairly stable topologies. Recent overviews of component testing approaches can be found in [1] and [23] covering built-in tests and mobile component testing, respectively.

Much work has also been done in the area of resource-aware and resource-adapting services and systems [4], [18], [21]. These approaches, however, do not include testing. We adopt some of the concepts of Ding et al., who describe a resource-aware agent infrastructure [6], and adapt these concepts for resource-aware testing, in particular, the separation of base functionality (a.k.a. business logic) and resource-dependent behavior. This resource-dependent behavior is often a domain-specific one (e.g. fidelity adaptation in multi-media applications) [7], [27] or more generally, a QoS related one [22]. In our work the possibility to adapt testing behavior to the given resource situation is studied systematically for the first time. Indeed, the specific contribution of our work lies in the novel application of resource-

awareness to run-time tests. In this paper we present this approach from a ubiquitous computing perspective.

This paper describes the MORABIT approach: a conceptual framework for components enhanced with resource-aware built-in testing capabilities as well as the design of the middleware necessary for such components. The remainder of this paper is as follows: In section 2 we describe constraints and requirements for the application of resource-aware built-in tests in ubiquitous computing scenarios. While the general conceptual infrastructure necessary to organize and run such tests is described in section 3, section 4 provides a more detailed description of our version of BIT tests. Section 5 describes specific approaches to the execution of these tests at run-time, namely strategies for resource-aware execution. An example application is described in section 6 before we conclude and discuss some possibilities for future work.

2 Constraints for Resource-Aware Built-in Testing in UbiComp

In the application scenarios we have in mind for this work any two software components may find each other and initiate some interaction. They can syntactically verify that they are – in principle – of a desired type, but they cannot look inside of each other to validate the implementation. More formally, A needs to verify that A and B share a common interpretation of the underlying contract of B's provided interface. Thus a component A that wants to use component B, needs to verify that component B fulfills some tests in order to minimize the risk that B fails in some way necessary for A's usage of it. An example could be an email component on a user's PDA (component A) that wants to connect to a server component offering email delivery (component B) on the premises where the user actually walks around on a business trip. Through some lookup services, the component A finds service B but since some servers may not send correct error messages or deliver mails in a corrupt format, tests are necessary in order to find out if A's understanding of what B should do matches to B's service. Of course, A needs to bring its own test cases for components of type B since another client A* of B might have other needs and may be less (or more) tolerant against B's implementation. Thus A must have some built-in tests for connectivity with B. The need for different tests may not only depend on different clients that come with their own needs. Even the same client-server combination may come with different tests in different contexts. If for instance our mailing components A and B are used in a high-security application domain, the tests may look differently than when they are deployed in a leisure environment. Therefore, tests must be described in a way such that both the components and the tests can be reused in other contexts. Additionally, testing B should not prevent other users from using B. Thus, the execution of tests needs to be adapted to the current load of the system and available resources. Since the results of the tests are open, the reaction to the outcome of the tests must be flexible and various reaction strategies should be possible. In our case, A might decide to use B without concern, might decide to use B only for uncritical mails or search for another server B*. In this case, A would test B on connection time. But during usage of B, B might change its behavior and a more 'paranoid' component A* might want to execute tests on B more often.

From this discussion we can see that for designing a framework of resource-aware built-in tests, we need to

- have a mechanism for describing built-in tests,
- have flexible test strategies for scheduling tests,
- define mechanisms for resource-aware test execution, and
- provide appropriate mechanisms for reactions to test results.

In the following, we present a framework that supports all four constraints. Moreover, we want to allow for a high level of flexibility and component reusability and thus want to separate all testing concerns from application logic. In many ways we follow the philosophy of resource-aware agents (RAJA) where basic agents implement the core application functionality and controllers supervise resources and adapt the agents' core behavior to the available resources [6].

3 Infrastructure

Every component system needs a run-time infrastructure. This can be an agent system or some component middleware. We are open for multiple such infrastructures and our conceptual approach can be used together with essentially any component middleware or agent system. In our generic demo implementation, we use a custom Java system that focuses on the novel aspects of our approach. In general, our framework and its respective infrastructure could be integrated into many existing platforms and thus allow all existing software components or agents to migrate to a system that is enhanced with capabilities for resource-aware built-in testing. These existing components could then stay as they were or be themselves enhanced with resource-aware test aspects (see Fig. 2). Therefore, we want to separate the implementation of the components' application logic and all testing and resource concerns as much as possible. This way, a scenario could be inhabited by both legacy components and MORABIT components. Moreover, programmers of application components are able to focus on their components and developers of tests can add their tests including test reaction strategies independently. Thus, it is possible to deploy the same components together with various different tests in different application scenarios. A database component, for instance, might have different test requirements in a hospital application than in a tourist application.

In the following we will sketch the overall design of the MORABIT middleware and describe its core components to provide an overview of the services provided to MORABIT components (see Fig. 1). Basic services include component instantiation and management including a service locator facility which allows components to lookup other components (servers) offering services for their required interfaces. A resource-aware infrastructure must keep track of the current state of the relevant resources. This is achieved by a set of resource monitors which observe the respective values for processor load, main memory, battery charge, network bandwidth, and other relevant resources.

Test support services play a crucial role within the MORABIT infrastructure. First of all, there is a test handler component which realizes the test concern on behalf of

every component. Technically, this can be realized by employing, e.g., Java's DynamicProxy facility. In order to properly balance the test execution and the processing of the 'normal' functionality, a component responsible for test scheduling is needed. It employs the test execution strategies described in section 5 to provide an appropriate trade-off between testing and core functionality in an intelligent manner. Furthermore, the test results are stored by the test result logger component both for inspection by a human administrator as well as a potential source of test history knowledge. Such a knowledge source can be exploited by more advanced strategies which might consider replacing actual test execution by a reliability estimate based on past test execution results.

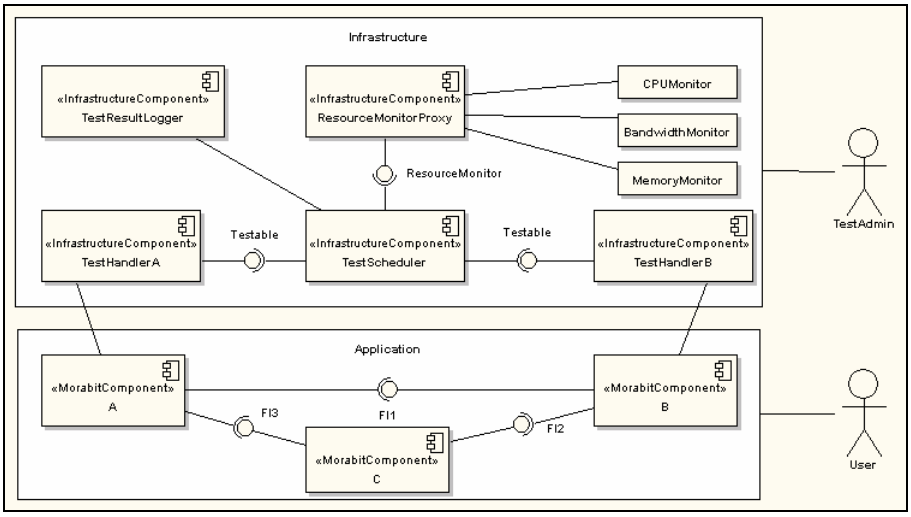

Fig. 1. High-level architecture of the MORABIT run-time infrastructure

A high-level view of the conceptual architecture is illustrated in Fig. 1. In this diagram we (logically) differentiate between the infrastructure and the application subsystems. While the former provides the infrastructure components with the services described above, the latter contains the actual application components. We have implemented a prototype of this infrastructure in Java [24].

4 Description of Built-In Tests

The 'built-in test' (BIT) approach implies that components 'bring their tests along', or more formally, that there is an unambiguous mapping from a given component to its associated test cases. The term 'built-in test' is not strictly correct in our approach. While earlier work [8] physically included the test code in the component code – either manually in the raw source code or semi-automatically via some wrapper mechanism – we decided to use a looser association between component and test definitions. The hard-coded approach implies serious problems for maintenance and

other problems associated with poor separation of concerns. Our approach to merely associate the test definitions logically with the component has all advantages of the strict built-in approach but avoids its inherent drawbacks by allowing an independent variation of component functionality and test definition, respectively. This flexibility is not only relevant at development (or maintenance) time, but may also be exploited at deployment or even run-time.

In order to understand the relationship between the core component and its associated test cases and metadata it can be helpful to distinguish between physical and logical components. In this metaphor, the core component with the basic functionality (a.k.a. the business logic) plays the role of the physical component. The logical component is comprised of the physical component, its associated test cases and additional meta-data concerning the execution of the test cases and the reaction of the component to the outcome of the tests. This relationship is illustrated in Fig. 2.

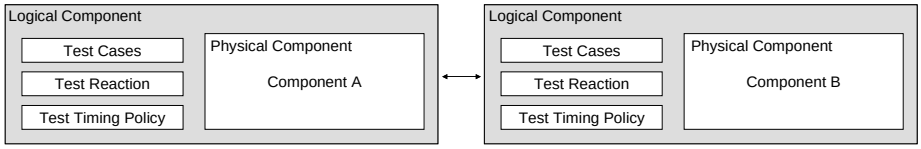

Fig. 2. Logical components consisting of the physical components and test metadata

The entities describing the test concern are modeled in detail in the schema shown in Fig. 3. The most comprehensive abstraction is a test request which in turn includes other relevant concepts such as test suites, individual test cases and other associated meta-data. In the following we will discuss the constituents of such a test request following the structure of this schema. An example document that conforms to the given schema can be seen in Fig. 4. This example describes a test request for a bank component.

Every entity in this schema has the properties of ‘name’ and ‘description’, both inherited from the abstract base type ‘DescribableItem’. This assures a uniform way of organizing basic descriptive data about the entities involved. A test request contains numerical thresholds with respect to some quality criterion, e.g. a reliability value (here: 0.8) together with a desired confidence level (here: 0.95). This reliability value defines failure or success of a test request such that the test request is considered successful if and only if the reliability value computed from the execution of the contained test suite exceeds the specified reliability value. Of course, the comparison of such a numerical value is only valid if we can determine a confidence level for the measurement.

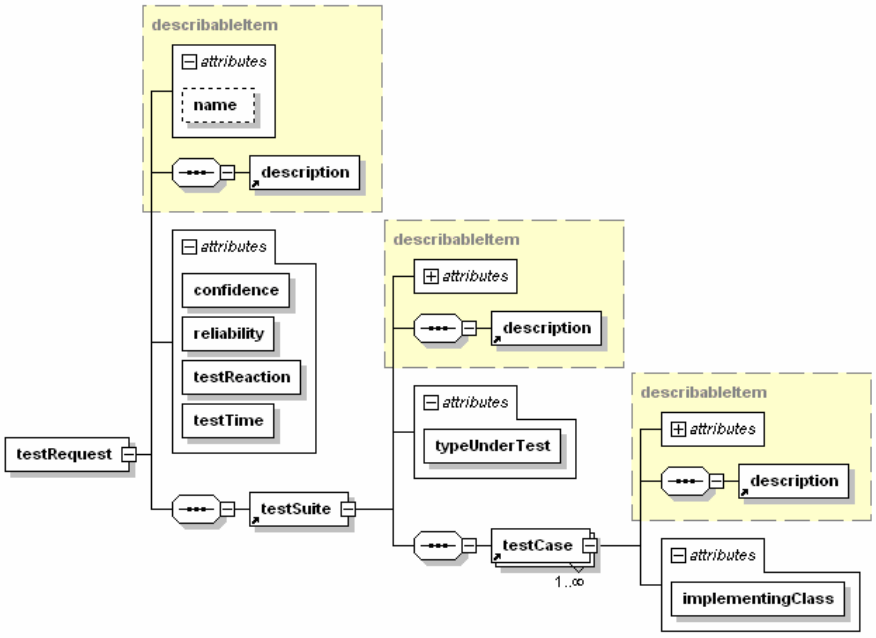

Fig. 3. XML schema definition of a test request in a graphical representation

The test request further contains a property ‘testTime’ (here: Lookup-time) which specifies a logical test time or, more precisely, test timing policy, that is, a well-defined point in time during the execution of a program. These logical points during the execution of a program are conceptually related to join points [16] known in aspect-oriented programming [17]. Logical test timing policies 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 the topology of the component nodes changes because a component leaves or joins the network
- *Periodic-time*: in a fixed time interval independent of functionality execution
- *Random-time*: in a random time interval with respect to some temporal distribution
- *Idle-time*: when there is a (local) minimum of the system load

In many practical situations the Lookup-time and Call-time timing policies are the most important ones. While these two have general importance in static as well as dynamic scenarios the Topology-change-time is especially important in more dynamic scenarios such as those commonly found in pervasive and ubiquitous systems. In particular, when more complex dependencies between components in a UbiComp scenario exist, it is not granted that a test that has passed (or failed) will pass again some time later even when the testing and the tested components remain the same. A simple example could be a dependency of the tested component on other components

that have been removed. Other examples could involve the use of resources that indirectly influence the behavior of a tested component. A server could, for instance, fail to respond adequately with respect to some QoS criteria (e.g., in time) to a response due to a higher load in the overall system even though it worked perfectly before.

```
<testRequest name="Mobile Bank Component"
             confidence="0.95"
             reliability="0.8"
             testTime="LookupTime"
             testReaction="TryNextComponent">
   <description>A test request for a server representing a bank</description>
   <testSuite name="Bank test suite"
              typeUnderTest="org.morabit.app.interfaces.bank.Bank">
       <description>A test suite for a server of type bank</description>
       <testCase name="Withdrawal test case"
                 implementingClass="org.morabit.testcases.WithdrawalTestCase">
           <description>a test case for a withdrawal transaction</description>
       </testCase>
       <testCase name="Payment test case"
                 implementingClass="org.morabit.testcases.PaymentTestCase">
           <description>a test case for a payment transaction</description>
       </testCase>
   </testSuite>
</testRequest>
```

Fig. 4. Example XML document describing a test request for a bank component with specified reliability, confidence level, test timing and reaction strategy, and two test cases

The test request also includes the specification of a test reaction strategy (here: Try-next-component). These strategies specify predefined reactions to the failure of a test request. Note that failure is defined only with respect to a given reliability criterion as described in the last section. As such reactions to a failure are often generic in nature it is desirable to be able to specify them declaratively. This avoids error-prone and repetitive manual implementation and resulting inconsistencies and supports a clean separation of concerns. Important test reaction strategies include:

- *Use-anyway*: in certain non-critical situations even a negative test result might be acceptable. A warning to the user or to the log can be issued potentially providing valuable data for debugging and problem analysis.
- *Try-next-component*: try to find another component offering the same desired service but providing better test results.
- *Use-best-component*: a strategy related to the Try-next-component strategy. At a given fixed point in time it guarantees the best component with respect to the given quality criterion. This guarantee is statistically associated with a higher test execution cost compared to the Try-Next-Component strategy.
- *Shut-down*: useful for critical tests; affects a shutdown of
    - Either the currently executing thread
    - Or the whole system/application (in cases where this notion is well-defined)

These are examples for strategies which easily can and therefore should be specified declaratively at development or deployment time. Such predefined strategies can be

supported by the run-time system and as with the testTime setting, they may depend on the actual deployment and context. Therefore, a component can provide different strategies. These test reaction strategies and test timing policies can be changed without modifying (i.e. changing the source code and recompiling) the physical component which implements the business logic as illustrated in Fig. 2.

In some situations more powerful reaction strategies including domain knowledge or even run-time information might be useful. In these cases it is necessary to provide reaction behavior that is programmed in the component code and to export some hooks for these custom reactions that can, in turn, be activated by the test strategy. This allows for a high degree of flexibility, while at the outside, it is still visible which reaction strategies are applied and all test-dependent behavior can still be changed later on depending on the particular needs in the scenario.

A test request also contains a test suite which is itself a complex entity. The test suite specifies a list of test cases and a type under test. Conceptually, this is the type of the target component under test; technically this is specified by a Java interface describing the service which is currently tested. Every test case in the list contains the definition of a test case in physical form, here, the Java class which provides the test case behavior. By such a declarative specification of test definitions the test cases can be loaded dynamically and instantiated (or even generated if necessary) at run-time which would not possible had the test cases been hard-coded within the host components.

5 Resource-Awareness and Test Execution

So far, we enriched the UbiComp paradigm with means for run-time testing and components can incorporate additional tests and strategies that help them to define their interactions with other components depending on the outcome of tests. This can already be sufficient and beneficial for many scenarios. However, just adding tests may sometimes even worsen the situation because of resource consumption and availability. If all components just go along and test, networking bandwidth might dramatically decrease or processor load might increase and a system might even collapse. Relevant resources include, for instance, computing power, network bandwidth, or even battery charge of mobile devices. It is one of the key features of our approach to adapt the testing process at run-time to the resource situation of the computing device. An important step in this direction is to treat the decision to execute a test request explicitly. If this decision is modeled explicitly as a strategy it is possible to reason about its different dimensions. These include the following:

- Approval or cancellation of the request
- Test start time (in a chronological sense, not to be confused with logical start times such as Lookup-time, or Call-time)
- Allocation of resources for tests in an explicit resource allocation scenario or assignment of priorities, e.g. thread priorities
- Selection of a partial test suite

A number of basic strategies can be identified. In many cases it might be possible to combine multiple strategies to form a composite strategy. Several such basic or atomic strategies are described in the following.

The *Constant Strategy* is the simplest strategy possible and explicitly ignores the resource situation. While it does not constitute a progress compared to other run-time testing approaches with respect to UbiComp applicability it can serve as a baseline for comparison with more advanced strategies.

The simplest strategy supporting resource-awareness is the *Threshold Strategy* under which tests are executed only if the resource availability exceeds a certain threshold for every individual resource. This strategy can be parameterized by allowing violations of the threshold condition by a certain percentage or for a number of resources. If estimations or measurements of the tests' resource consumption are available an additional threshold for maximum resource consumption for single test cases can be defined.

The *Weighted-Resource Strategy* allows treating resources non-uniformly. By associating different weights with resources it is possible to treat, e.g., memory in a different manner than network bandwidth. This applies to situations where a component knows that it will need a given resource (e.g. the network) rarely but will in these rare situations use a large percentage of the resource for a limited time or where it needs a lot of memory for a prolonged period of time but will access the network rarely.

The *Priority Strategy* provides a mechanism to privilege the 'normal' functionality of the deployed applications over the test execution. Unlike the Threshold Strategy which also supports reserving resources this strategy can differentiate between load generated by the core functionality and by tests. If necessary, variable priorities can be assigned to individual functional and test requests, respectively.

The *Meta-Data Aware Strategy* interprets the additional meta-data of the test request as hints for modifying the test request priority dynamically. As detailed in section 4, these additional meta-data include:

- (logical) test execution time (e.g. Call-time, Lookup-time, or Idle-time)
- test reaction strategy (e.g. Shut-down, Try-next-component, or Use-anyway)

Clearly, the specified test reaction information can be used to reassign priorities or to even cancel a test request altogether if the resources are currently scarce depending on the implicit importance of the test request deduced from the test reaction. If additional domain and/or application-specific knowledge are available then it might be possible to derive test priorities from test timing policies, e.g. decide automatically whether a call-time or a lookup-time test is more important in a given scenario.

The *Delay Strategy* supports controlled postponement of tests during resource shortages with global or individual (per test request) maximum delay values. This 'poor man's optimization' heuristically avoids local maxima in the resource demand of tests while avoiding complex and thus expensive planning algorithms. While this does not globally optimize the resource allocation and test scheduling it can increase the probability for proper system operation in many situations. Multiple waiting test requests can be managed in a FIFO queue.

The *Statistical Significance Strategy* computes a reliability measure for a given component during the execution of a test suite until the computed reliability can be determined with statistical significance thus assuring that the test execution itself uses the given resources economically. For any given reliability model [10] (based on test execution results) confidence in the computed reliability depends monotonically on the number of test executions and thus on the effort invested. The optimal number of tests with respect to resource economization and confidence maximization is reached when the calculation of the confidence reaches statistical significance. If information on the expected cost of a test case is available then even the *order* of test cases can be changed to optimize the ratio between test execution cost and contribution to the confidence level.

The *Custom Strategy* allows explicit programmatic access to resource measurements and to the test execution moving the control of the test execution process from the infrastructure to the component and thus the programmer. This approach increases the flexibility and the set of possible strategies by taking into account both static domain as well as dynamic run-time knowledge. Disadvantages include that the programmer of such a strategy cannot easily be forced to actually take the resource situation into account and that it becomes difficult to enforce a uniform resource consideration strategy across different components - especially from disparate sources.

The *Full Planning Strategy* treats the allocation of resources to tests and the test scheduling as a full planning problem [13]. While such a planning approach has the theoretical benefit of providing the optimal allocation and scheduling results it is a problematic approach in a UbiComp scenario with limited hardware capabilities due to the computational complexity and associated high computational cost.

These nine strategies have very different properties in a number of respects. They vary from basic resource aware strategies like the Threshold Strategy to more advanced strategies. The latter take the ‘real’ functionality of the application into account, consider additional test meta-data, have a statistical foundation, or use AI techniques such as planning. The Custom Strategy represents a special case as the overall control is moved away from the infrastructure to the responsibility of the programmer. The strategies introduced here vary greatly in both their own resource consumption as well as effective allocation of resources to the test execution process. For the framework of this paper, we want to show the possibilities for realizing such strategies – from simple to complex. However, for practical considerations, simulations and further studies (theoretical and experimental) are necessary for comparing the appropriateness of these strategies depending on a particular scenario.

It should be noted however, that in most realistic settings, heuristics are needed that work sufficiently well. In most UbiComp scenarios that are sufficiently complex, many parameters will be unknown and replaced by estimates. Thus, on the one hand, complex strategies like the full planning strategy might lack enough knowledge. On the other hand, simple strategies might be just good enough.

Depending on the knowledge of the application domain and the usage of resources, one might also consider mixtures of strategies, e.g., a threshold strategy for battery power and a delay strategy for CPU load. Since we separated the test strategy from the design of the tests and the implementation of the components, all parameters and settings for the test strategy can be designed independently of other implementation

and testing issues. The strategy can even be modified at run-time in case that some shortages appear.

6 Implementation and Sample Application

We have implemented a prototypical version of a middleware supporting the described concepts. Its light-weight component model and some technical issues as well as a test isolation mechanism are described elsewhere [24]. While the current implementation still has limitations w.r.t. distribution and test execution strategies many important properties of resource-aware test execution can already be studied.

In order to evaluate and demonstrate the usefulness of our approach we have developed an application scenario which involves mobile users and devices in changing compositions. A prototypical version of this application has been implemented in Java and is currently run experimentally on the developed middleware.

Even in the times of ebay, classical auction houses such as Sotheby's or Christie's are still popular. Especially high-priced goods are traded there. People physically visit these auctions to either sell items or to bid for offered items. Instead of placing bids by raising the hand, the bidders could use their mobile devices. The bid is entered into the mobile phone or PDA and is then sent to the auctioneer. The (ad-hoc) network of connected mobile devices might change rapidly because new bidders can come into the network and old bidders can leave the network. In any case, in the end the highest bidder wins. In order to get the won item, the bidder has to authorize the payment at her bank to pay off the auction house. For this, the auction house is connected to a bank service. This bank has to fulfill several different services so that the auction house can use it. To be sure that both the auction house and the bank have the same understanding of the services tests are executed. Further details on the demo application can be found in [3].

In this scenario the auction house component could bring along a test request for its required server, i.e. the bank component. A simplified version of such a test request including two separate example test cases for payment and withdrawal actions is illustrated in Fig. 4 in section 4. This scenario also provides a wide field for experimentation with the different test execution strategies discussed in section 5. For instance, for tests executed on the PDAs resource-cautious test execution is more important compared to tests running on the (potentially more powerful) computer which hosts the auction house component. Also, some functional requests such as payments are more critical and thus require higher reliability than others such as browsing of auction item images. So the tradeoffs between test execution and basic functionality as well as between performance and reliability can be studied in relation to test start times, test reactions, and test execution strategies.

This application scenario does not yet provide rigorously measurable numbers of increased reliability for a given performance level or increased general performance during run-time test execution. However, it serves to validate the conceptual approach and to uncover flaws in the design and implementation of the run-time infrastructure.

7 Conclusion and Future Work

There has been a widespread initial enthusiasm in the years following Weiser's seminal paper [29] and many new application domains such as location-based systems, wearable computing, intelligent homes, and ambient intelligence have been proposed in the ubiquitous and pervasive computing communities. This ongoing search for 'killer applications' has provided a strong impetus for the development of the UbiComp field as a whole and has furthered the study of many important properties of ubiquitous computing systems. However, in recent years it has become clear that after the initial hype a more systematic approach to the actual development and deployment of ubiquitous computing applications is gaining importance. This trend is documented by, e.g., several UbiSys workshops [11]. While it is likely and appropriate that the focus of UbiComp research will remain on more fundamental issues we consider it important to acknowledge that more research is needed in the area of systematic support for the development of deployable ubiquitous computing software systems. Here, software engineering inspired approaches can contribute to progress. The dynamic formation of many ubiquitous computing systems calls for improved support especially in the area of integration and run-time testing. In fact, Davies and Gellersen [5] explicitly name integration issues as a major challenge in the deployment of ubiquitous computing systems and emphasize the need for reliability metrics for deployed ubiquitous computing components. Our paper provides a first step in the direction of increasing the run-time reliability of ubiquitous computing systems by introducing systematic run-time testing even in scarce resource situations. Thus, combining the application of software engineering principles from (run-time) testing and component technologies with appropriate middleware support can contribute to the improvement of open and dynamic systems.

We have introduced the MORABIT approach to run-time testing in ubiquitous software systems. This approach combines run-time testing and resource-awareness in a novel way. We have shown why such an approach can be considered beneficial in many current and future ubiquitous scenarios, namely due to the inherent resource limitations of typical UbiComp devices and the dynamic nature of the network topologies involved. The traditional approach to the formation of dynamic ubiquitous computing systems relies mainly on syntactic (and some semantic) description of the services offered. This implies that chance and coincidence still play a major role during run-time. Our approach enhances run-time compatibility by enabling run-time tests and, in turn, flexible reactions to the ensuing test results. The typical resource-constrained environments of ubiquitous computing systems call for a resource-aware management of the testing process. To this end we have proposed and discussed a number of test execution strategies for resource-aware test management. These strategies are at the heart of the novel combination of run-time testing and resource-awareness. After describing the constraints and requirements for built-in tests we showed how such tests can be represented by appropriate meta-data information. These requirements led to the design of a conceptual infrastructure which we described in detail with special emphasis on a clear and clean separation of testing and business logic (i.e. 'normal' functionality) concerns. While the described concepts are

useful independent of an implementation it is, however, important to validate the approach within a real implementation.

Future work includes the study of distribution-specific issues, namely the possibility to distribute the execution of test suites in a load balancing manner. An important part of our future work is a systematic empirical evaluation of the effects achieved by means of the resource-aware run-time testing approach described herein. To this end a number of metrics such as reliability measures will have to be defined in terms of the number and kind of tests executed. We can then measure these metrics when running carefully crafted sample applications and their tests with different degrees of resource-awareness and compare the numerical results.

Although we have demonstrated the usefulness of the MORABIT approach in ubiquitous systems – mainly based on structural (with respect to dynamic topologies) and device-related arguments – one important characteristic of ubiquitous systems has not yet been included: By considering an explicit context dependency in terms of the (extra-device) situation and the user, the execution of tests could be further improved.

Acknowledgements

This work has been funded by the Klaus Tschira Foundation (KTS) and the Landesstiftung Baden-Württemberg within the MORABIT research project. We thank our colleague Christian Elting for reviewing the manuscript.

References

1. Beydeda, S.: Research in Testing COTS Components - Built-in Testing Approaches. In: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications. IEEE Computer Society Press (2005)
2. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional (1999)
3. Brenner, D.: A Case Study for Resource Adaptive Built-in Test Components. Diploma Thesis, University of Mannheim (2004)
4. Capra, L., Emmerich, W., Mascolo, C.: CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. In: IEEE Trans. Softw. Eng., Vol. 29. (2003) 929-945.
5. Davies, N., Gellersen, H.-W.: Beyond Prototypes: Challenges in Deploying Ubiquitous Systems. IEEE Pervasive Computing, Vol. 1, no. 1. (2002) 26-35
6. Ding, Y., Malaka, R., Kray, C., Schillo, M.: RAJA: a resource-adaptive Java agent infrastructure. In: Proceedings of the 5th International Conference on Autonomous Agents. ACM Press (2001)
7. Ding, Y., Pfisterer, D., Walther, U.: Resource-adaptive Video Streaming for Mobility. In: Proceedings of Workshop on Artificial Intelligence in Mobile Systems (2002)
8. Edwards, S.H.: A framework for practical, automated black-box testing of component-based software. In: Software Testing, Verification and Reliability, Vol. 11. (2001) 97-111
9. Edwards, W. K., Newman, M. W., Sedivy, J., and Izadi, S.: Challenge: Recombinant Computing and the Speakeasy Approach. In: Proceedings of the 8th International Conference on Mobile Computing and Networking MobiCom'02. ACM Press, New York (2002) 279-286.

10. Fenton, N., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, Revised. Course Technology. (1998)
11. Friday, A., Roman, M., Becker, C., Al-Muhtadi, J.: Guidelines and open issues in systems support for Ubicomp: reflections on UbiSys 2003 and 2004. In: Personal Ubiquitous Computing, Vol. 10. (2005) 1-3.
12. Gao, J.Z., Tsao, J., Wu, Y.: Testing and Quality Assurance for Component-Based Software. Artech House Publishers (2003)
13. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann (2004)
14. Gross, H.-G.: Component-Based Software Testing with UML. Springer Verlag (2004)
15. Jöst, M., Häußler, J., Merdes, M., Malaka, R.: Multimodal interaction for pedestrians: an evaluation study. In: Proceedings of the 10th International Conference on Intelligent User Interfaces. ACM Press (2005) 59-66
16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), Springer Verlag, Berlin Heidelberg New York (1997) 220-242
17. Kiselev, I.: Aspect-Oriented Programming with AspectJ. Sams (2002)
18. Kon, F., Yamane, T., Hess, C.K., Campbell, R.H., Mickunas, M.D.: Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In: Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001) (2001) 15-30
19. Love, S.: Understanding Mobile Human-Computer Interaction. Butterworth-Heinemann (2005)
20. Morla, R., Davies, N.: Evaluating a Location-Based Application: A Hybrid Test and Simulation Environment. In: IEEE Pervasive Computing, Vol. 3. (2004) 48-56
21. Poladian, V., Sousa, J.P., Garlan, D., Shaw, M.: Dynamic Configuration of Resource-Aware Services. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society (2004) 604-613
22. Rajkumar, R., Lee, C., Lehoczky, J., Siewiorek, D.: A Resource Allocation Model for QoS Management. In: IEEE Real-Time Systems Symposium. (1997) 298-307
23. Suliman, D., Paech, B., Borner, L.: Testing Mobile Component Based Systems. In: Proceedings of Net.ObjectDays 2005. (2005)
24. Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.: The MORABIT Approach to Runtime Component Testing. In: Proceedings of the Second International Workshop on Testing and Quality Assurance for Component-Based Systems. (TQACBS06). (2006). (to appear)
25. Szyperski, C.: Component Software. Addison-Wesley Professional (2002)
26. Vincent, J., King, G., Lay, P., Kinghorn, J.: Principles of Built-In-Test for Run-Time-Testability in Component-Based Software Systems. In: Software Quality Control, Vol. 10. (2002) 115-133.
27. Walpole, J., Koster, R., Cen, S., Cowan, C., Maier, D., McNamee, Pu, C., Steere, D., Yu, L.: A player for adaptive mpeg video streaming over the internet. In: Proceedings of the 26th Applied Imagery Pattern Recognition Workshop AIPR-97. SPIE (1997) 249-258
28. Wang, Y., King, G., Patel, D., Court, I., Staples, G., Ross, M., Patel, S.: On built-in tests and reuse in object-oriented programming. In: SIGSOFT Softw. Eng. Notes, Vol. 23. (1998) 60-64.
29. Weiser, M.: The Computer for the 21st Century. In: Scientific American. (1991) 94-104