A Model-Based Approach to Test Automation for Context-Aware Mobile Applications

Tobias Griebe
paluno - The Ruhr Institute for Software Technology
University of Duisburg-Essen
Essen, Germany
tobias.griebe@paluno.uni-due.de

Volker Gruhn
paluno - The Ruhr Institute for Software Technology
University of Duisburg-Essen
Essen, Germany
volker.gruhn@paluno.uni-due.de

ABSTRACT

Current testing tools for mobile applications do not provide sufficient support for context-aware application testing. In addition to regular input vectors (e.g. touch events, text entry) context parameters must be considered (e.g. accelerometer data interpreted as shake gestures, GPS location data, etc.). A multitude of possible application faults resulting from these additional context parameters requires an appropriately selected set of test cases. In this paper, we propose a model-based approach to improve the testing of context-aware mobile applications by deducing test cases from design-time system models. Using a custom-built version of the calabash-android testing framework enhanced by an arbitrary context parameter facility, our approach to test case generation and automated execution is validated on a context-aware mobile application.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Verification

Keywords

mobile, testing, context-awareness, model-based

1. INTRODUCTION

Contemporary mobile devices feature a set of sophisticated means to sense parameters of their physical environment. Applications can use these parameters, referred to as *context*, as additional input vectors and adjust application behavior accordingly. Using these additional input vectors introduces novel sources of potentially erroneous application behavior. For testing processes, the impact of these features needs to be carefully examined.

Mobility, device heterogeneity and context-awareness are key multipliers of testing effort. Effectively testing context-aware applications requires deploying the application on a representative set of target devices and exposing these devices to various context conditions. Testers needs to parameterize the testing environment according to test case specification, i.e. setting virtual location parameters or virtual sensor data. These tasks yield a high potential for human error when entering testing data as well as for the interpretation of testing results.

Key challenges of efficient mobile context-aware application testing are the creation of adequate test cases and their efficient execution. In the approach we describe here, both challenges are addressed by automated test case generation from design-time system models and automated test case execution in simulated execution environments as well as on actual handsets.

In our approach, test cases are generated using model transformation on context-enriched design-time system models. In a four-tier process system models (i.e. UML Activity Diagrams, section 3) are enriched with context information (tier 1) using an UML profile developed for integrating context information into UML models. Models are then transformed into Petri Nets (tier 2) for analyzing and processing structural model properties (e.g. parallel or cyclic control flows). From the Petri Net representation, a platform- and technology-independent system testing model is generated (tier 3) that includes context information relevant for the test case execution. Finally, platform- and technology-specific test cases are generated (tier 4) that can be executed using platform-specific automation technology (e.g. JUnit, calabash-android/ios, Robotium).

The remainder of this paper is organized as follows: challenges of testing context-aware mobile applications are discussed in section 2, our approach to context information modeling in section 3. Test case generation is discussed in section 4 and test case execution in section 5. We validate our approach with a case study described in section 6. Related work is analyzed in section 7; a conclusion and future work discussed in section 8.

2. CHALLENGES OF TESTING CONTEXT-AWARE APPLICATIONS

In mobile computing varying definitions for the term *context* have been proposed (e.g. [1, 7, 17]). Commonly it refers to any information that can be used to characterize the situation of an entity, where an entity is a person, place,

http://dx.doi.org/10.1145/2554850.2554942

or object that is considered relevant to the interaction between a user and an application [1], which is the definition of context applied in the scope of this paper.

In traditional software testing, a set of test cases is created by spanning a testing space over an application's functionality and possible interactions with respect to specific functions. For context-aware applications, besides manual user interactions these interactions can also be external stimuli, i.e. changes in context parameters (e.g. changing network availability, relocation, change in availability of sensor data), which may occur at any time and are usually unpredictable.

Changes in context expand the set of test cases significantly. The tester not only has to test system functions by supplying user interactions, he also has to supply these interactions under various constellations of the system's context.

2.1 Mobile Application Testing Challenges

In mobile computing, network connections are subject to frequent changes in availability, latency and rate of transmission. Sensor readings are inaccurate or ambiguous. Both, networking and sensor readings may also strongly depend on a multitude of parameters such as platform, device, version of operating system, handset manufacturer additions to the operating system, mobile telephony and internet carrier specifications. These factors may affect a mobile application significantly, e.g. prevent the communication with web services or generating time-out signals due to prolonged response latency [4, 16]. To a limited degree, parameters such as response latency or data rates may be configured when testing on emulators. But carrier-specific characteristics are generally impossible to test within an emulator. Hence, the application needs testing on a device with a set of different carriers (e.g. one carrier might support cell-site-based location information while others do not). For testing, network parameters, among others, need to be changed at run-time to ensure an application works according to specification.

One of the most challenging features in terms of testing is location data. Testing on a device requires the tester to physically move the device to cover adequate distances, move the device between cells of a GSM or UMTS grid, roam between carrier networks, etc. Testing within an emulator environment may allow for the provisioning of arbitrary location data, but will not reflect characteristics of real carrier networks (e.g. unexpected service loss or quality-of-service parameters). Eventually, for testing context-aware mobile applications an educated trade-off between device-testing and emulator-testing should be applied for best results. The approach introduced in this paper facilitates testing activities, both on emulators and devices, by automating processes of test case creation and execution.

2.2 Consistent Test Application Example

To discuss our approach to mobile application testing automation, a comprehensive example of a location-based "call-a-cab" application is introduced and used throughout the remainder of this paper. The application supports the user in calling a cab to his current location either by using GPS/cell-site-based or manual location determination. If the user selects automatic location determination, the app tries to obtain location information using GPS or cell-site information. Once the user's location has been determined, the user is presented a map dialog for location verification where the ordering process can be completed. Automatic location determination may fail due to a variety of reasons such as signal loss or poor signal quality caused by urban canyoning [6, 11] or other reasons, in which case the application falls back to manual location determination by direct user input. For the application to use GPS or cell-site information, a quality constraint is placed on location data. Location accuracy must not exceed 200 meters and location data age is required to be less than 5 seconds. Additionally, the location determination process via GPS/cell-site location may take no longer than 30 seconds before unavailability is assumed and fallback to manual location determination is initiated. Unavailable network connection renders the application inoperative, whereupon a corresponding dialog is presented to the user as soon as the network connection is lost.

Testing this application requires test cases to include both location determination modes, setting valid and invalid location data and manipulation the network connection to simulate unexpected service loss. For manual creation of test cases satisfying these requirements, a significant effort is expected. Besides simulation of valid and invalid location data, the loss of the network connection needs to be simulated on separate test runs after each individual step through the workflow.

3. MODELING TEST CASES

Models of software systems are key artifacts in software development processes that help architects, programmers and testers to align their understanding of a software system. Models are used for a variety of purposes ranging from communication to automated code generation in Model-Driven Architecture/Development (e.g. [5]) and Model-Driven Testing (e.g. [2, 3, 9]). The Unified Modeling Language (UML) has been widely accepted as de-facto industry standard providing a set of diagrams to model various aspects of software systems. However, the UML does not provide means to model context-awareness in software systems. Hence, using context parameters in software is left to the programmer's interpretation of textual requirement documentation. Consequently, model-based code generation and test case generation techniques can not yet use context parameters as input artifacts.

3.1 Enriching UML Activity Diagrams with Contextual Test Data

In our approach, we have selected UML Activity Diagrams (ACD) as base for automated test case generation. Among the diagram types provided by the UML, ACDs are well suited for modeling dynamic functional system behavior for test case generation purposes. ACDs are utilized for making user interactions explicit [10, 12] as well as for reusing existing design artifacts [13, 14]. Since ACDs do not provide model elements to include context parameters, we created an UML profile for context-awareness modeling (Figure 1). We extended various UML meta classes with adequate stereotypes to provide attributes describing context parameters. These parameters are modeled using the abstract types *TestStepAction*, *TestStepPrecondition* and *TestStepPostcondition* specified in the platform-independent test case meta model (subsection 3.2).

Applied to Activity Diagrams, the profile enables the modeling of context parameters and associating these instances with the UML ACD model elements. UML ACD elements that have the stereotype *TestStep* applied are considered rel-

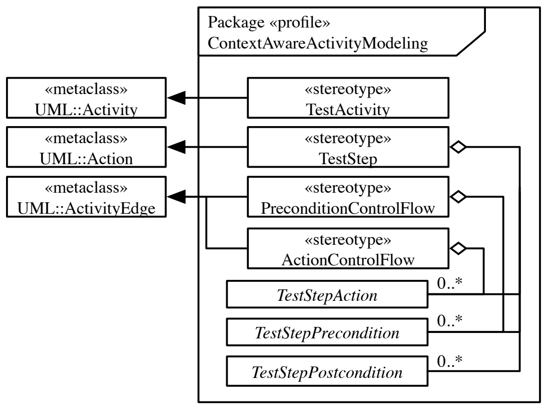

Package «profile»
ContextAwareActivityModeling
«metaclass» UML::Activity
«metaclass» UML::Action
«metaclass» UML::ActivityEdge
«stereotype» TestActivity
«stereotype» TestStep
«stereotype» PreconditionControlFlow
«stereotype» ActionControlFlow
TestStepAction
TestStepPrecondition
TestStepPostcondition
0..*
0..*
0..*

Figure 1: UML profile for context-aware activity modeling.

evant for the test case generation while all other elements are ignored. We allow for the stereotype application to edge elements to model alternative context conditions or user actions on decision node elements. Since UML models may contain an arbitrary number of ACDs, the stereotype *TestActivity* is used to denote those ACDs that shall be transformed into test cases. Model elements that have the stereotype *TestStep* applied may feature a number of test step execution preconditions, such as specific location or sensor data. Also, they may feature a number of user interactions, e.g. touching a user interface element, entering data, swipe gestures or performing a shake gesture with the device. They may also feature postconditions that the application needs to satisfy after the user interactions are completed for the current test step.

In Figure 2 the ACD modeling the "call-a-cab" application introduced in subsection 2.2 is depicted. Actions that are considered relevant for testing have the stereotype *TestStep* applied. The action "check network connection" for example is considered irrelevant because it executes as background process with no option for the user to interfere. In Figure 3, a section of the ACD is depicted where a control flow has the stereotype *PreconditionControlFlow* applied. The stereotype specifies a location that in this use case represents a valid location for the ordering process. Upon test case execution, this information is used to parameterize the test execution environment. Analogous, user actions can be defined.

3.2 Meta Model for Test Case Modeling

The platform-independent test model is based on a meta model we developed to explicitly support context parameter modeling in test cases. We define a *test case* to be a set of consecutively executed *test steps*. Individual test steps are atomic test artifacts composed of sets of preconditions, user of actions and postconditions, where each set is allowed to be empty. A test case represents a possible path through the application under test (AUT) with a defined entry point, where elements on that path are individual test steps, hence representing a use case. Alternative paths through the AUT are considered individual test cases. We define a *test suite* as a set of test cases for a AUT, the set of test suites for a AUT forms the *test model*. We developed the extensible *TestCaseModel* meta model depicted in Figure 7. It provides el-

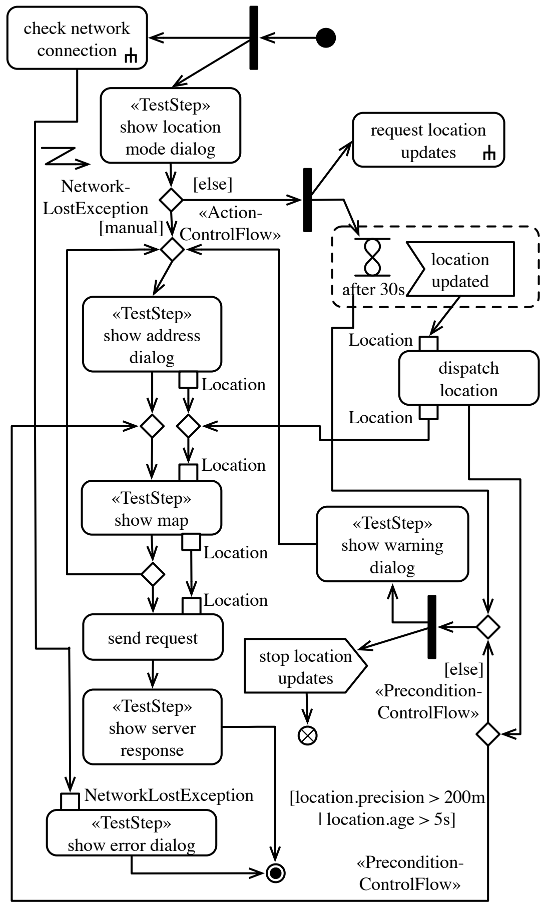
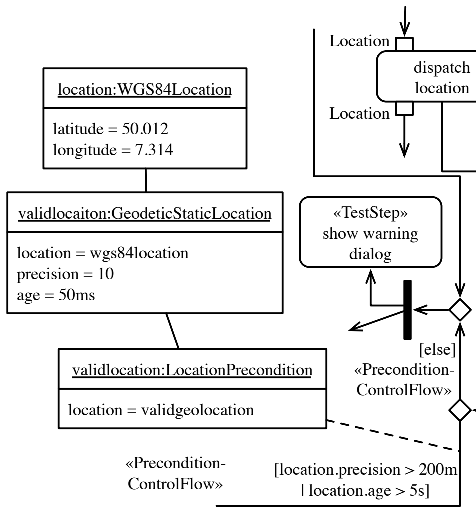


Figure 2: UML Activity Diagram of the "call-a-cab" application. The diagram has been annotated with test data using the UML profile for context-aware mobile application modeling.

ements for the modeling of context parameters (i.e. location and sensor data, user interactions, network condition etc.) to represent these artifacts in a platform-independent test case model. Using a set of transformation rules, instances of *TestCaseModel* models can be generated from input models (i.e. UML Activity Diagrams) using Petri Nets as intermediate model representation (discussed in subsection 4.1). In Figure 7 an UML Class Diagram of the *TestCaseModel* meta model is depicted in excerpt. The full meta model contains model elements to represent context parameters for static and dynamic geodetic location information, symbolic location information, sensor data for various sensors (magnetic field, proximity, ambient temperature, accelerometer, gravity, etc.), network availability etc. Also, common low-level interaction modes (e.g. touch, swipe, text entry) and high-level interaction modes (e.g. orientation change, flipping face down, pinch, zoom) are supported. Model instances created from the meta model are intended to be free of cycles, which eases model traversal for platform-specific test case generation as discussed in subsection 4.2.

4. TEST CASE GENERATION

The complexity of application faults related to context-



Location
dispatch location
Location
location:WGS84Location
latitude = 50.012
longitude = 7.314
validlocaiton:GeodeticStaticLocation
location = wgs84location
precision = 10
age = 50ms
«TestStep» show warning dialog
[else]
«Precondition-ControlFlow»
validlocation:LocationPrecondition
location = validgeolocation
«Precondition-ControlFlow»
[location.precision > 200m | location.age > 5s]

Figure 3: Excerpt of the "call-a-cab" ACD (see Figure 2) where the stereotype *TestStep* specifies a location precondition on a control flow element.

awareness easily exceeds human capabilities in test case creation and execution. Automation improves test case creation by eliminating human errors at key processes. Our approach to automated test case generation consist of a four-tier process. First, system models are enriched with context data using the UML profile introduced in subsection 3.1 (tier 1). Next, these models are transformed into instances of the *TestCaseModel* in two steps: transformation into a Petri Net representation (tier 2) and generating *TestCaseModel* instances (tier 3) from the Petri Net's reachability tree. From the platform-independent test case model, platform-specific test cases are generated (tier 4).

4.1 Platform-Independent Test Cases

Control flows in ACDs allow for the modeling of alternative and parallel flows. Our *TestCaseModel* meta model intentionally does not provide elements to model alternative and parallel flows. Instead, separate test cases are used to model alternative flows. In the transformation process, cycles in an ACD's control flow are unwound, executing each cycle exactly once before the non-cyclic alternative is selected. This satisfies the all-transition-coverage criterion [19] in which each edge is followed at least once.

The analysis of UML ACDs for parallel and cyclic structures is a non-trivial task. Hence, we choose to transform the ACD into a Petri Net as an intermediate representation that helps with the model analysis. Petri Nets provide a solid scientific base for analyzing alternative and concurrent control flows. We transform all UML ACD elements, i.e. nodes and edges, into Petri Net places. Using this approach, control flows and object flows can become active components used for context annotation. We create Petri Net transitions between the places based on the edges in the ACD, preserving alternatives by competitive transitions and parallelism by transitions with multiple target places. The association between Petri Nets places and ACD elements is preserved in the process for the next steps in tier 3. During this process, a lookup table is created that maps UML ACD elements to Petri Net nodes for later use in the transformation to *TestCaseModel* instances. A minimal example of the the Petri Net transformation is depicted in Figure 4, where a ACD contains alternative decisions and a cycle.

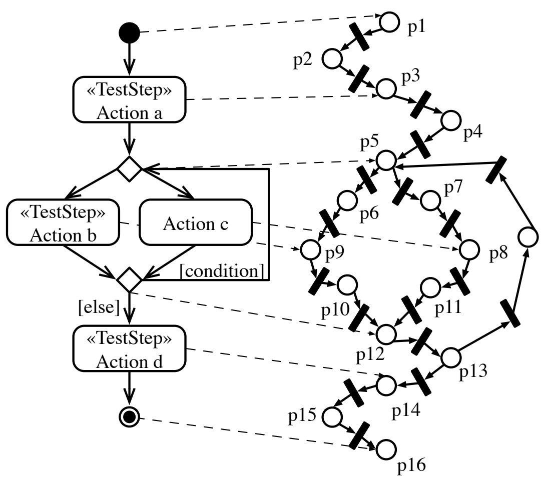


Figure 4: Example of UML Activity Diagram to Petri Net transformation. Dashed lines illustrate how UML Activity Diagram elements are associated with Petri Net places.

To generate the *TestCaseModel* instance from the Petri Net, the Petri Net's reachability tree is automatically calculated. A token is put on all places that have been created from ACD elements that qualify as starting nodes of control flows (e.g. UML Initial Node). By determining activated transitions and calculating follow-up markings, the full reachability graph is obtained. Nodes in the reachability tree represent markings of the Petri Net that refer to UML Activity Diagram nodes that are active at this point of the execution the underlying use case. The calculation stops at markings under three conditions: no more transitions can fire, a marking places a token on a place that qualifies as flow-terminating UML ACD element or a marking is created that already exists in the reachability tree and, hence, identifies a cycle.

The reachability tree may have leaf nodes whose marking does not include a reference to a final node. This indicates a non-completed action sequence in the ACD. Typically this occurs at nodes in the ACD from which cycles are initiated (e.g. by alternatives at decision nodes). In the example depicted in Figure 4, the markings $\{p5\}$ and $\{p12\}$ (underlined in Figure 5) satisfy this condition. They represent markings that already exist in the tree (denoted with the dotted line). These nodes are expanded to complete test cases using subtrees starting with cycle-identifying markings that result in a marking including a flow finalizing UML ACD elements. The complete tree for the minimalistic example ACD in Figure 4 is depicted in Figure 5. Places with a zero token count have been omitted.

From this resulting graph of markings, all paths from root node to leaf nodes are separated into individual *traces*. Subsequently, from all traces nodes are removed that do not represent relevant artifacts from the underlying UML Activity Diagram. For this purpose, we defined a relevance criteria that evaluates positive for all elements of the underlying ACD that have one of the stereotypes defined in

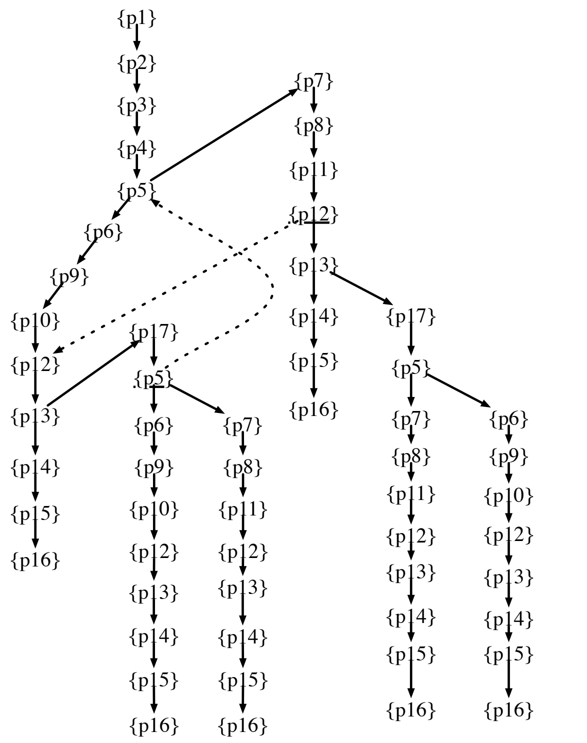


Figure 5: Reachability graph for the Petri Net depicted in Figure 4. Cycles have been unwound to form a tree structure.

the context-awareness modeling UML profile applied. Remaining markings represent those ACD elements that are considered relevant for testing. Iterating the tree depicted in Figure 5, from all nodes representing Petri Net markings the places 2, 4, 6, 7, 10, 11, 13, 15 and 17 are removed. They represent control flows having no relevant stereotypes applied. The places 5 and 12 are removed because they represent UML decision node / UML merge node elements that have no corresponding elements in the *TestCaseModel* meta model. Finally the place 8 is removed, since the associated UML Action "Action c" in Figure 4 has no stereotype applied. The set of resulting traces may contain duplicates from which all but one copy are omitted. This process is illustrated for the leftmost trace from Figure 5 in Figure 6.

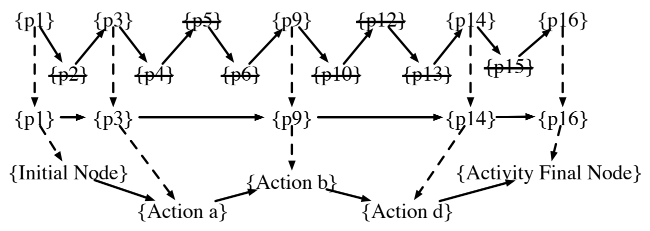


Figure 6: Traces of Petri Net markings are reduced, so that only test-relevant actions of the underlying ACD remain.

Using the lookup table created upon Petri Net transformation, the remaining markings in the traces are replaced by their corresponding ACD elements. For the model in Figure 4, this process results in 3 traces: {"Action a", "Action d"}, {"Action a", "Action b", "Action d"} and {"Action a", "Action b", "Action b", "Action d"}, representing test cases according to the all-transition-coverage criteria [19]. The "Action c" is not part of any of the test cases since it has none of the relevant stereotypes applied. The third test case contains "Action b" twice because of the cycle in the underlying ACD.

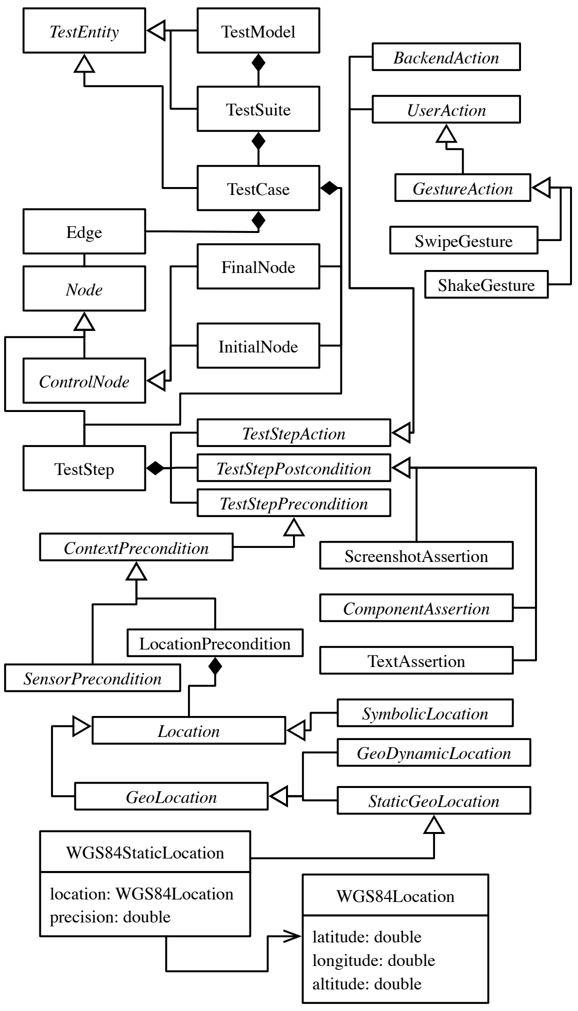


Figure 7: Excerpt from the UML Class Diagram of the *TestCaseModel* meta model. Subtypes, methods, attributes, labels and cardinalities have been omitted.

In the next step, tier 3, instances of the *TestCaseModel* meta model are created. For each element encountered in the ACD the corresponding element from the *TestCaseModel* meta model (Figure 7) is instantiated according to the applicable transformation rule (e.g. UML Action is mapped to *TestCaseModel* TestStep). Applying the transformation rules the final platform-independent test case model is obtained. In the next step, this model is transformed to platform-specific test cases.

4.2 Platform-Specific Test Cases

When executing test cases on specific devices or device emulators, platform dependency is introduced into the process. Technology-specific test cases are derived from the platform-independent test case model by using a model-to-code transformation that converts individual model elements into code that is used for automated test case execution. Mobile platforms support test automation to a limited extent by programming language-specific frameworks such as JUnit/CUnit on a close-to-code base, where test cases are written in the same language as the AUT. With a special focus on mobile platforms high-level frameworks have been developed that allow for the specification of functional application test cases where complex functional aspects, rather than individual code units, are addressed. Typically test cases are composed of three steps: initializing the test environment to satisfy case-specific preconditions, performing a set of actions according to the test case specification and verifying that the system satisfies a set of postconditions. In the platform-independent model obtained from the enriched ACDs, these three steps are modeled using designated modeling elements. Complete test scenarios are composed of a number of test suites, that are modeled in a tree structure using the *TestCaseModel* meta model. Each contains a number of test cases, where test cases are composed of test steps. Test cases are obtained traversing each path from root to leaf nodes in the test model's tree structure.

For the generation of platform-specific test cases, the test case model needs to be converted into artifacts used by the target test case automation technology. At the current stage of our research, we have selected calabash-android/ios as target automation technology. Calabash-android/ios, for the remainder of this paper simply referred to as calabash, is a functional test automation technology available for the platforms Android and iOS, where test cases are expressed as *cucumber* features. Originating from the periphery of Behavior Driven Development, cucumber is an acceptance test automation technology that allows for the specification of test cases using the domain-specific language *gherkin*.

For the technology-specific test case generation, we prepared a number of gherkin templates. Each template corresponds to *TestCaseModel* meta model elements. During the generation process, the template suitable for the specific model element encountered during the tree traversal of the test model is parameterized with the data specified in the test model element. Elements that are on the path from root to leaf node in the test model tree are aggregated into a cucumber scenario describing a complete test case. Different paths in the test model tree form different test cases and, hence, different scenarios. Multiple scenarios are then composed into complete cucumber features that can be automatically and repeatedly executed by the calabash test automation tool. Listing 1 depicts a set of gherkin statements that are part of a cucumber scenario derived from the ACD (section 3, Figure 2) representing the mobile "call-a-cab" application. In Listing 1, part of the test case is illustrated where the user opted for automatic location determination. The specific location information set to the device or emulator while testing has been set up to be rejected by the application due to failure to satisfy accuracy and age constraints. Line 1 introduces a new scenario which reflects a test case from the platform-independent test case model. Lines 2, 3, 4, 7 and 8 illustrate how test case preconditions are expressed in calabash by the keyword "Then I" followed by a constraint. These preconditions have been derived from the enriched UML Activity Diagram (Figure 2) representing the application, where the test case modeler has annotated test-relevant actions with test data describing preconditions, actions and postconditions. Line 5 represents a user interaction. Line 6 illustrates how context parameters are included in the test case. In this case, the artificial location data consisting of latitude, longitude, accuracy and age is set to the AUT. Details on how artificial sensor data, including but not limited to GPS data, is enforced on the device or emulator while testing is discussed in section 5. Line 9 represents the assertion of a postcondition for a test step.

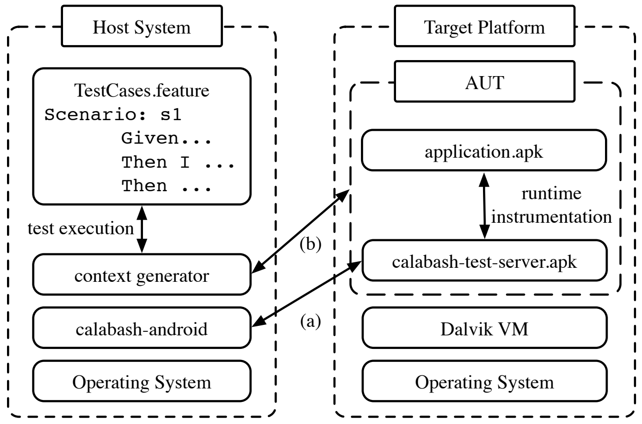
```
1 Scenario: testcase03
2   Given my app is running
3   Then I wait for the view with id
        dialog_locationmode to appear
4   Then I wait for the view with id button_gps to
        appear
5   Then I press view with id button_gps
6   Then the location is 51.461, 7.016 accuracy
        400.00 age 350000
7   Then I wait for the view with id dialog_warning
        to appear
8   Then I wait for the view with id
        button_commit_warning to appear
9   Then I see the text "Sorry your location could
        not be determined."
```

Listing 1: Example from the "call-a-cab" application of gherkin statements for use with the calabash test automation tool, generated by our approach.

In the scope of this work we have implemented the transformation of the platform-independent test model to calabash features. Test artifacts for other technologies such as JUnit, CUnit, Robotium, etc. can be implemented analogously, as long as the selected target technology is suitable to cope with context parameters.

5. TEST CASE EXECUTION

For automated test case execution, the AUT is deployed onto the execution target platform (i.e. device or emulator). Additionally, calabash creates and deploys an application test server that manages the communication of the calabash framework with the AUT (Figure 8, link *a*) and enforces context parameters onto the target where applicable and simulates user interaction using the Android instrumentation tools (Figure 8, link *b*).



```
Scenario: s1
      Given...
      Then I ...
      Then ...
```

Figure 8: Schematic view of the test execution framework and its communication links.

In Figure 8 the area to the right represents the Android device or emulator, on top of which the test execution stack is illustrated. The left part represents the test case execution engine. The test case execution engine currently is implemented using the calabash-android testing framework. The context generator, part of our custom-build calabash-android implementation, enables the setting of artificial context data provided by the context-aware test case model.

When a test case is executed the next action on the current test step sequence is selected. Depending on its type it is directed to the corresponding layer of the test case execution engine. Actions of the type context-simulator are handled by the context generator component. Depending on the type of the context-data specified in the gherkin statement representing the test step, the context generator utilizes communication link *b*, Figure 8 to enforce the specified context data onto the emulator or device. Context-related preconditions are read from the corresponding gherkin statement and supplied to the context-generator. When a user action is detected in the model, the action is executed by the test case execution component using the Android instrumentation framework via communication link *a*. Finally, the results of the interaction are extracted using the Android instrumentation technology (i.e. reading values from user interface elements etc.) and checked against postcondition elements contained in the model for the currently executed test step element (via link *a*).

6. CASE STUDY

To validate our approach, we thoroughly modeled and implemented the mobile, context-aware "call-a-cab" application introduced in subsection 2.2 and enriched its model with context data to serve as case study. Prior to test case execution, a set of test cases need to be created satisfying the following requirements: test manual location determination by entering an address where a correct positioning on the map view is expected, automatic location determination with correct positioning on the map view, specification conforming reaction to unavailable GPS or cell-site-based location information, specification conforming reaction to unexpected loss of network connection, specification conforming behavior when readjusting the location after the map dialog.

We applied our approach to automatic test case generation and test case execution to the "call-a-cab" application. The generation process generated a total of 32 test cases representing all paths of subsequent actions through the application including the cycle formed by the actions "show address dialog" and "show map". We found that the generated test cases fully covered the above requirements, i.e. all possible situations that may occur during application execution where covered by a test case.

The shortest test case generated consisted of just one element representing the error dialog when the app is started with a dysfunctional network connection in the first place. The most complex generated test case consists of 9 test steps and represents a flow through the application where the user initially selected automatic location determination which results in the fallback to manual location determination due to a timeout condition including manual location readjustment. In between all scenarios are covered where the work flow is disrupted by a lost network connection.

All steps including setting location data, enabling/disabling network connection and comparing screenshots for asserting the display of the correct map position could be executed fully automated. Hence, using our approach has proven successful in automated test case creation and execution for a mobile, context-aware application. Using our approach, manual steps in test case creation and test case execution could be reduced to enriching the application's UML model with test-relevant context parameters. By adjusting the values for the setting of a valid location, a different set of test cases could be easily created for different locations. When the app is modified or enhanced with additional functionality, a new set of test cases can be generated from the updated model.

7. RELATED WORK

A significant number of researchers have devoted their work to automating, optimizing and improving testing processes and tools to reduce the effort put into testing while at the same time increasing test coverage and reliability. These approaches take system models (e.g. UML model, Event-driven Process Chain models, etc.) as input and produce test cases or test plans to be conducted manually by testers, or a set of instructions for automated testing tools. These projects are mostly focused on traditional desktop or client/server architectures, where the system's context in terms of hardware, operating system and operational environment is largely known to the developer.

Focusing on distributed web applications, Heckel et al. [9] propose a model-driven approach to testing. The authors discuss algorithms to create test cases from system models, the generation of test oracles to determine expected test results and the execution of tests in dedicated environments. The generation of test cases from design-time models is assumed to be a platform-independent task, while the execution of the test cases is platform-specific. Despite being focused on a non-mobile scenario, the approach also includes the simulation of external factors.

Sarma et al. [15] present an approach to generate test cases from UML Use Case and UML Sequence Diagrams. By means of model transformation the diagrams are transformed into graphs representing use cases and use case sequences for the system. By integration of both graphs, test cases are deduced for a non-context-aware, non-mobile testing scenario. The approach of converting design time system models into an intermediate representation and subsequent enrichment with additional test data is also used in the approach presented in this paper.

An early approach to incorporate dynamic context-aware features into application testing has been proposed by Bylund et al. [8]. The authors present an approach to simulate context data by utilizing a modified 3D gaming engine to dynamically create artificial context data that is fed into the system under test at runtime. While the user moves within the virtual environment, context data is mapped to test data for application testing outside the virtual environment.

Satoh et al. [16] present a framework for testing applications for mobile computing devices. The focus is on external resources such as network services. In their approach, the authors create a mobile agent serving as device emulator. Physical movement of the system under test is mapped to logical movement of the mobile agent within a network. Being a pre-smart phone era approach, this method only allows for basic testing of application behavior in roaming scenarios, which cannot be expected to produce reliable results.

Taranu et al. [18] present a general method for testing context-aware applications. By introducing a context management system, context information is directly delivered to the system under test by stimulating relevant system interfaces. The system under test is embedded into a complex test bed containing high level situation generators that create context information relevant to the test case. The work presented in [18] is rather focused on networking systems than on individual mobile applications.

8. CONCLUSION AND FUTURE WORK

In the previous sections we have proposed our approach to automated test case generation. In a four-tier process, UML Activity Diagrams are enriched with context parameters and test data using a UML profile for context-awareness modeling. Using model transformation, from the ACD a Petri Net representation is obtained. The Petri Net representation simplifies analyzing the model for parallel and cyclic structures that need to be considered when choosing adequate test coverage criteria. From the reachability tree of the Petri Net all paths from root node to leaf nodes are extracted that eventually form test cases. From the Petri Net markings in each path ACD elements are obtained; those irrelevant for test case execution are removed from the path. Using model-to-code transformation, from the platform-independent model platform-specific test cases are derived apt for automated execution. We have validated our approach with a mobile, context-aware application from which a set of test cases satisfying the coverage criteria have been generated and automatically executed.

Currently our approach requires static modeling of test data. Our future research includes methods to introduce more flexibility into the test data modeling by using placeholders in user actions, preconditions and postconditions that at runtime can be provisioned with test data from a data base. Also, we are currently using an all-arc-coverage criterion, in which test cases are selected that all edges in the ACD are traversed at least once. However, in our future research we plan to integrate more sophisticated coverage criteria, especially to deal with nested cyclic structures in ACDs that potentially introduce potential application defects.

9. REFERENCES

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, UK, 1999. Springer.

[2] L. Apfelbaum and J. Doyle. Model Based Testing. In *Software Quality Week Conference*, 1997.

[3] Baker, Paul and Ru Dai, Zhen and Grabowski, Jens and Haugen, Oystein and Schieferdecker, Ina and Williams, Clay. *Model Driven Testing - Using the UML Testing Profile*. Springer–Verlag Berlin Heidelberg, 2008.

[4] B. Bergvall-Kåreborn and S. Larsson. A case study of real-world testing. In *MUM '08: Proceedings of the 7th International Conference on Mobile and Ubiquitous Multimedia*, pages 113–116, New York, NY, USA, 2008. ACM.

[5] S. Beydeda, M. Book, and V. Gruhn. *Model-Driven Software Development*. Springer London, Limited, 2005.

[6] W. Broll, J. Ohlenburg, I. Lindt, I. Herbst, and A.-K. Braun. Meeting technology challenges of pervasive augmented reality games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, NY, USA, 2006. ACM.

[7] P. Brown, J. Bovey, and X. Chen. Context-aware applications: from the laboratory to the marketplace. *Personal Communications, IEEE*, 4(5):58–64, 1997.

[8] M. Bylund and F. Espinoza. Using Quake III Arena to Simulate Sensors and Actuators When Evaluating and Testing Mobile Services. In *CHI '01 Extended Abstracts on Human Factors in Comp. Systems*, CHI '01, pages 241–242, New York, NY, USA, 2001. ACM.

[9] R. Heckel and M. Lohmann. Towards Model-Driven Testing. *Electr. Notes Theor. Comput. Sci.*, 82(6), 2003.

[10] A. Heinecke, T. Brückmann, T. Griebe, and V. Gruhn. Generating Test Plans for Acceptance Tests from UML Activity Diagrams. In *IEEE: Engineering of Computer-Based Systems (ECBS 2010)*. IEEE, 2010.

[11] M. Jabbour, P. Bonnifait, and V. Cherfaoui. Enhanced local maps in a gis for a precise localisation in urban areas. In *Intelligent Transportation Systems Conference, 2006. ITSC '06. IEEE*, pages 468 –473, September 2006.

[12] H. Kim, S. Kang, J. Baik, and I. Ko. Test Cases Generation from UML Activity Diagrams. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, ACIS International Conference on*, 3:556–561, 2007.

[13] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. *Asia-Pacific Software Engineering Conference*, 0:284–291, 2004.

[14] C. Mingsong, Q. Xiaokang, and L. Xuandong. Automatic Test Case Generation for UML Activity Diagrams. In *AST '06: Proceedings of the 2006 International Workshop on Automation of Software Test*, pages 2–8, New York, NY, USA, 2006. ACM.

[15] M. Sarma and R. Mall. System Testing using UML Models. *Asian Test Symposium*, 0:155–158, 2007.

[16] I. Satoh. A Testing Framework for Mobile Computing Software. *IEEE Trans. Softw. Eng.*, 29(12):1112–1121, 2003.

[17] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, WMCSA '94, pages 85–90, Washington, DC, USA, 1994. IEEE Computer Society.

[18] S. Taranu and J. Tiemann. General Method for Testing Context Aware Applications. In *MUCS '09: Proceedings of the 6th International Workshop on Managing Ubiquitous Communications and Services*, pages 3–8, New York, NY, USA, 2009. ACM.

[19] S. Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines*. PhD thesis, 2010.