Master of Science in Informatics at Grenoble
Master Informatique, Université Grenoble Alpes
Specialization AISSE

Context-dependent Model-based Testing of Mobile Apps

Abdallah Adwan

21st of June, 2021

Masters research project performed at
LCIS

Under the supervision of
Mme. Oum-El-Kheir Aktouf
M. Ioannis Parissis

Defended before a jury composed of
Cyril Labbé, President
M. Hubert Garavel
<external expert>
Mme. Oum-El-Kheir Aktouf
M. Ioannis Parissis

Context-dependent Model-based Testing of Mobile Apps

Abdallah Adwan

Abstract

Equipped with sophisticated sensors and tremendous computing power, modern mobile devices, such as smartphones are now capable of sensing their environment. As a result, mobile apps are increasingly incorporating functionalities whose behavior not only depends on explicit user input but also on the state of the surrounding environment, known as *context.* This type of mobile apps is known as context-aware mobile apps. While software testing is a challenging and expensive task in general, testing context-aware software is even more complex. In this report, we propose a model-based testing approach for automatically generating context-dependent test cases from a combination of behavioral and context model programs written in Domain-Specific Languages we developed. We also propose context-based test coverage criteria as a measure to describe the degree to which the generated test suite covers the context. Our approach can simplify and accelerate the modeling process for testing and gives testers a sense of context coverage in the generated test cases.

Resumé

Équipés de capteurs sophistiqués et d'une énorme puissance de calcul, les appareils mobiles modernes, tels que les smartphones, sont désormais capables de détecter leur environnement. En conséquence, les applications mobiles intègrent de plus en plus de fonctionnalités dont le comportement dépend non seulement de la saisie explicite de l'utilisateur, mais également de l'état de l'environnement environnant, appelé contexte. Ce type d'applications mobiles est appelé applications mobiles contextuelles. Alors que le test de logiciel est une tâche difficile et coûteuse en général, le test de logiciel contextuel est

encore plus complexe. Dans ce rapport, nous proposons une approche de test basée sur un modèle pour générer automatiquement des cas de test dépendant du contexte à partir d'une combinaison de programmes de modèles comportementaux et contextuels écrits dans des langages spécifiques au domaine que nous avons développés. Nous proposons également des critères de couverture de test basés sur le contexte comme mesure pour décrire le degré auquel la suite de tests générée couvre le contexte. Notre approche peut simplifier et accélérer le processus de modélisation pour les tests et donne aux testeurs une idée de la couverture du contexte dans les cas de test générés.

Acknowledgments

I would like to thank my supervisors, Mme. Oum-El-Kheir Aktouf and M. Ioannis Parissis for their consistent support and valuable guidance during the course of my research internship.

Table of Contents

1. Introduction 1

2. Background 3

2.1 Mobile App Testing 3

2.2 Mobile Context and Context-awareness 5

2.3 Model-based Testing (MBT) 6

3. Problem Statement, Analysis and State of the Art 8

3.1 Problem Statement 8

3.2 State of the Art 9

3.3 Analysis 13

4. Contribution: Context-driven Modeling 14

4.1 Domain Specific Languages 14

4.2 Context Modeling 15

4.2.1 Context Model 15

4.2.1 Context Definition Language (CDL) 17

4.2 Behavioral Modeling 20

4.2.1 Context-enriched Hierarchal Finite State Machine 20

3.2.2 Context-driven Modeling Language (CDML) 23

5. Test Case Generation 26

5.1 Contribution: Test Coverage Criteria 27

5.2 Test Subpath Generation 29

5.3 Test Path Aggregation 32

5.4 Abstract Test Case Generation 34

6. Conclusion 36

6.1 Future Work 37

Bibliography 38

1. Introduction

In the last decade, mobile devices such as smartphones, tablets, and smartwatches have experienced significant advancements in areas such as cost, processing power, storage capacity, communication bandwidths, camera capabilities, and many others. Consequently, mobile devices have become pervasive, reaching nearly 6.4 Billion users in 2020 and almost doubling since 2016 [1]. Mobile devices (especially smartphones) nowadays are an integral part of people's lives and are being used in a wide array of daily tasks anywhere at any time.

With the enormous adoption of mobile devices across the world's population and the great capabilities they possess, the development of mobile apps has significantly accelerated in the last decade. As a result, the number of apps available to download in the major app stores globally has grown rapidly and was estimated at 6.4 million mobile apps at the end of 2020 with Google's Play Store (Android) and Apple's App Store (iOS) accounting for the majority of the available apps [2]. The number of mobile apps downloaded worldwide in 2020 was estimated at 218 billion downloads, up from 140 billion in 2016 [3]. In 2018, global mobile app revenues amounted to over 365 billion U.S. dollars. In 2023, mobile apps are projected to generate more than 935 billion U.S. dollars in revenues via paid downloads and in-app advertising [4].

Equipped with a multitude of sophisticated small-scale sensors, mobile devices can now sense the surrounding environment of users and deliver context-aware services. Modern mobile apps are increasingly characterized as context-aware [5]. Such mobile apps can sense clues about the situational environment and enable proper interaction between the user and the application. A renowned example is Uber, a location-aware mobile app that helps passengers book private cars driven by casual drivers. On the user interface of Uber[1], users can see on a map their location and locations of nearby available vehicles.

Nowadays, the mobile app industry is highly competitive, and the massive number of available apps further intensfies the competition for user's attention. In order for app developers to stay competitive in the market, they need to pay close attention to the quality of their apps.

[1] https://www.uber.com/

Software Testing plays a crucial role in the development of high-quality software systems. According to Myers et al. [6], software testing "is the process of executing a program with the intent of finding errors." While software testing is a challenging and expensive activity in general, testing context-aware software is even more complicated. In the case of context-aware mobile apps, the mobile app needs to be tested in different context conditions (and changes between these conditions) under which it may run.

Conventional testing tools and techniques were shown to be inadequate to test context-aware mobile apps [7][8][9] as they fail to assess the behaviors of such apps corresponding to the various possible context conditions to which they may be exposed.

To this end, we propose in this research report a context-driven model-based testing approach that aims to simplify and structure the process of testing context-aware mobile apps by automatically generating context-dependent test cases from a combination of behavioral and context models. First, a model of the mobile app's context is built where context variables, their sources, their structure, and the relationships between them, as well as, interesting contextual situations are defined.

Then, a behavioral model based on the *hierarchal finite state machine (HFSM)* formalism [10] - where models consist of a hierarchy of finite state machines (FSMs) - is built. In the HSFM model, states at which available context information could influence the mobile app's behavior are explicitly marked as *context-aware states.* Context-aware states are linked with corresponding context variables (defined in context model), and adaptations starting from these states are explicitly modeled by *adaptation FSMs.* An Adaptation *FSM* models an adaptive action in response to a contextual situation being satisfied (as defined in the context model) at a context-aware state.

To simplify the modeling process and to enforce our model-specific constraints, we created two Domain-Specific Languages (DSLs), *Context Definition Language (CDL)* for context modeling and *Context-driven Modeling Language (CDML)* for behavioral modeling. A context model program written in CDL and its corresponding behavioral model program written in CDML represent the *test model.* The *test model* is input to the automated test case generation process.

The automated test case generation is driven by test coverage criteria. However, since the context is modeled separately, using only transition-based

coverage criteria for FSM (such as All-Transitions) is insufficient. Therefore, we defined context-based coverage criteria to be used together with transition-based coverage criteria to drive the automated context-dependent test case generation.

The remainder of this report is organized as follows: a brief background on mobile app testing, mobile context, context-awareness, and model-based testing (MBT) is given in chapter 2. The problem statement, state of the art, and related work analysis are presented in chapter 3. Context modeling, behavioral modeling, and the use of modeling DSLs are presented in chapter 4. Context-based coverage criteria and automated test case generation are discussed in chapter 5. Chapter 6 concludes the report.

2. Background

In this chapter, we begin with a brief background on testing mobile applications, and then we discuss mobile context and context awareness. Finally, we give a quick summary of black-box model-based testing (MBT).

2.1 Mobile App Testing

A mobile app is a piece of software that runs on mobile devices such as smartphones, tablets, and smartwatches. According to Goe et al. [11], mobile app testing refers to "testing activities for native and Web applications on mobile devices using well-defined software test methods and tools to ensure quality in functions, behaviors, performance, and quality of service, as well as features, such as mobility, usability, Interoperability, connectivity, security, and privacy."

When it comes to testing mobile applications, a number of unique requirements, not applicable to conventional software, need to be taken into account. First, mobile apps must function correctly anywhere at anytime. Furthermore, mobile apps are often developed to run on a spectrum of mobile devices that are diverse in terms of manufacturers, operating systems, computing resources and screen sizes. Developers must ensure their mobile apps functions correctly on all target devices.

In addition, modern mobile apps are increasingly characterized as context-aware since they use sophisticated sensors, which modern mobile devices are equipped with, to sense the user's environment and deliver context-aware services. Context-awareness makes testing more complex as the mobile app needs to be tested with different possible context conditions under which it may run.

Goe et al. [11] identified four popular approaches for testing mobile apps: Emulation-based testing, Devices-based testing, Cloud testing, and crowd-based testing.

Emulation-based testing: The emulation-based testing uses a software-emulated mobile device (virtual machine) which runs on a personal computer. Emulation-based testing helps reduce testing costs as no physical equipments need to be purchased or rented, however, it has its own limitations as well. First, emulators are usually based on specific platform or device and it is not possible to test diverse devices and platforms. In addition, the simulation capabilities of emulators are limited and they are not capable of simulating real-world contexts.

Device-based testing: The device-based approach requires purchasing physical mobile devices and setting up a testing laboratory. The main drawback of this approach is the high cost comparing to Emulation-based testing, however, it can be used to verify device-based functions and Quality of Service parameters. The major challenge in device-based testing is the rapid changes in mobile devices and platforms.

Cloud testing: The Cloud testing (also known as Testing-as-a-Service) approach is a service usually provided by Cloud venders where a farm of mobile devices are made available over the cloud for developers to test their applications. This approach offers diversity in mobile devices and platforms and can support large scale testing. Comparing to device-based testing, cloud testing is more cost-effective as help eliminate the costs of purchasing, maintaining, and upgrading mobile devices.

Crowd-based testing: The crowd-based testing approach involves using contracted or freelance testing engineers or a community of end-users such as uTest, along with a crowd-based testing infrastructure and a service

management server to support diverse users. This approach allows testing the application in the wild, which not possible using other approaches. The main drawback is that testing is mainly manual, and the opportunity for test automation is minimal.

2.2 Mobile Context and Context-awareness

The most commonly used definition of context in the literature is the one by Dey et al. [12]. They defined context as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves". They also defined context-aware software system (CASS) as "A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task." CASS captures the information about context and uses it to adapt their behavior and assist the user while they perform their tasks.

There are different classifications of mobile application context existing in the literature. Gomez et al. [13] classify mobile application context as *static context* and *dynamic context. Static context* is the context that does not change during the execution of the mobile application. Contexts falling into this category including OS version, screen size, and device manufactures. *Dynamic context,* on the other hand, varies during execution, such as network state, battery level, readings of sensors (e.g., GPS).

Chen and Kotz [14] classify mobile context as *computing context*, *user context*, *physical context,* and *time context. Computing context* refers to network connectivity, communication bandwidth, nearby devices, and hardware or software resources. *User context* may refer to the user's profile, location, social situation, and people nearby. *Physical context* refers to illumination, noise level, traffic condition, and temperature. Finally, *time context* may refer to the time of the day, day of the week, and weekends.

Marcu et al [15] classify context as *low-level context* and *high-level context.* Low-level contexts are the raw readings of sensors that themselves may not be interesting to the context-aware application—for instance, the raw readings of the accelerometer sensor. High-level contexts are derived from low-level contexts and are interesting for the semantics of context-aware

applications. For example, from the raw readings of the accelerometer sensor, two high-level contexts can be derived: (i) The user is moving or still and (ii) the user is laying or standing.

Bettini et al. [16] presented the concept of *situation* in context-aware computing. The notion of *situation* is commonly used as a higher-level concept for state representation. In context-aware applications, situations are the external semantic interpretation of context. *Situations* can be defined as a set of conditions on the context. When viewed from this perspective, situations are used to aggregate and model specific constraints that are to be recognized in order to perform adaptive actions. Situation modeling is not intended to be exhaustive, meaning not all possible situations needs to be considered. Using situations to model the context of context-aware applications can make design and implementation much easier because programmer/designer can operate at a high level of (situation) abstraction and not be directly concerned with the context cues used to define the situation.

2.3 Model-based Testing (MBT)

Model-based testing (MBT) is a black-box testing technique where the explicit expected behaviors of the software under test (SUT) and/or its environment are modeled for the purpose of automatically generating test cases. Test cases are generated from a single model or a combination of models and then executed on SUT.

According to Utting et al. [17], the use of MBT is motivated by the observation that customarily, the process of generating test cases tends to be not documented, not reproducible, and unstructured. The process is also based on the intuition of individuals and lacks rationales for test design. Using models that explicitly encode the SUT's and possibly its environment behaviors can help overcome these problems.

Also in [17], Utting et al. specified the 5 main steps of MBT ((shown in Figure 1)) as follows.

Step 1. A model of the SUT is built from existing specification documents or informal requirements. The produced model is often called t*est model,* because the level of abstraction and the focus of the model are directly linked with the testing objectives.

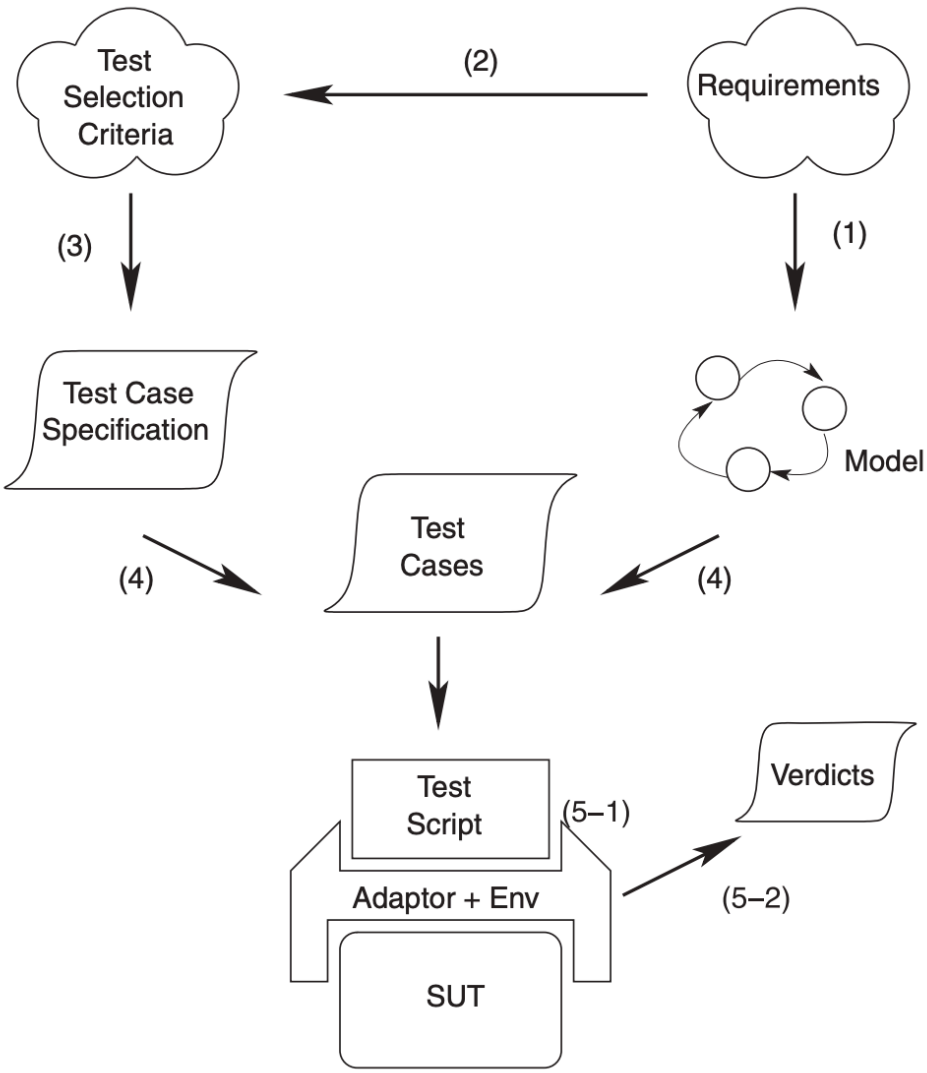

Figure 1[17]: The Process of Model-based Testing

Step 2. Test coverage criteria are selected to drive the automated test case generation. Test coverage criteria can be based on a specific functionality of the software or be based on the structure of the model (e.g. transition-based coverage for finite state machine (FSM) models).

Step 3. In this step, test coverage criteria is transformed into *test case specifications.* Test case specifications formalize the notion of test coverage criteria and render them operational. For example, a 'transition coverage' of a finite state machine (FSM) might be transformed into a set of test case specifications such as {explore t1, explore t2, explore t3,...} where t1, t2, t3, ... are the transitions of the FSM.

Step 4. A set of abstract test cases satisfying the test case specifications (step 3) is automatically generated from the test model (step 1).

Step 5. The test suite is executed. Test case execution can be manual (by a manual tester) or automated using a test execution environment. In case of automated execution, abstract test cases (step 4) need to be transformed into concrete test cases (also known as test scripts) where steps in an abstract test case are mapped to executable test code of the SUT.

3. Problem Statement, Analysis and State of the Art

3.1 Problem Statement

In the beginning, mobile apps had desktop application features adapted for mobile devices. Email applications are an example of such early adaptions. Muccini et al. [5] classified mobile apps into two categories: (i) traditional applications that have been adapted to run on mobile apps, called *Apps4Mobile*. And (ii) mobile apps that use context information as input to adapt their behavior based on context values called *MobileApps*. Thus, mobile apps have become, as time went by, increasingly pervasive. Its behavior depends not only on user inputs but also on context information. Therefore, modern mobile applications are increasingly characterized as context-aware applications.

Testing *Apps4Mobile* follows the process of traditional software testing, where a set of test cases is created to cover possible user interactions over the functionalities of the mobile application. However, testing MobileApps (context-aware applications) is more complicated. Such applications not only process direct input from the user, they also sense changes in the environment (context) of the user. Such context changes (e.g., network condition change, location change, etc.) can occur anytime and are usually unpredictable.

As a consequence of context-awareness, the test space of such mobile apps grows significantly. Testers not only need to test possible user interaction with the mobile application, but they also need to test these user interactions with the different context conditions under which the mobile application may run.

Despite the advancements made in the development of context-aware mobile applications that made them today an integral part of people's lives,

these apps are still vulnerable to faults that are burdensome to reproduce and fix [18]. Furthermore, conventional software testing techniques are not suitable to effectively assess the behaviors of such apps corresponding to dynamic environments[19]. Therefore, the following challenges need to be addressed:

Modeling of context: Existing software modeling techniques used in model-based testing, such as UML Activity Diagrams, are incapable of modeling context and context-aware behavior[7][9]. Thus, novel modeling techniques need to be developed for model-based testing of context-aware mobile apps.

Test coverage: Exhaustive testing of computer software is infeasible[6]. Test coverage criteria are used as a measurement to assess how much of the application is covered by a test suite. Conventional test coverage criteria such as data-flow criteria were shown not to be adequate to test context-aware software systems as they fail to uncover context-aware faults [8]. Therefore, specialized test coverage criteria need to be developed for the testing of context-aware mobile apps.

Acquisition of context test data and test case execution: Some complex context configurations appearing in the wild are extremely difficult to simulate and require real-world testing. Real-world testing is often very costly and rendered infeasible. Furthermore, context causes an explosion in the space of possible executions to cover.

Test case design: Conventional test cases are static in the sense that their expected output is determined prior to test case execution. In context-aware mobile apps, context information could change at any moment (e.g., during text execution) and, in turn, influence the expected output. *Context-coupled* test cases must adapt their expected output according to the available context information.

3.2 State of the Art

Research on testing mobile apps has grew significantly in the last decade. However, a modest number of studies focusing on the context-awareness challenge of testing mobile apps exists [20]. Several testing methods and tools

for context-aware mobile applications have been proposed in the literature, varying from black-box testing such as model-based testing (MBT) to white-box testing such as the data-flow technique.

Yu et al. [21] explored the use of Bigraphical Reactive Systems model for modeling the behavior of context-aware applications. Test cases are generated by using the middleware and environment models as inputs for the bigraphical model-based testing. The generated test cases are used to verify the interaction between the context-aware environment and the middleware together with domain services. Two tools are presented to support the proposal: (ii) a visual bigraphical modeling tool called BigM, and (ii) a bigraphical simulation tool called SimM.

Griebe and Gruhn [7] proposed a model-based approach to improve testing context-aware mobile applications. In their approach, first, the system models (i.e. UML Activity Diagrams) are enriched with context information. Then, these models are transformed into Petri Nets. From the Petri Net representation, a platform and technology-independent system testing model is generated that includes context information relevant for the test case execution. Tests are created to cover the all-transition-coverage criterion. Finally, platform and technology specific test cases are generated and can be executed using platform-specific automation technology. To execute the generated tests, the authors present a custom-built version of the calabash-android testing framework.

Mehmood and Khan [9] presented a testing framework for context-aware applications using an approach similar to the one proposed in [7]. The testing procedure consists of 6 phases. In the first phase, a behavioral model of the application is created using an extended UML Activity Diagram in which context-aware activities are identified and a typecast of the activity node named *context-aware activity node* is used to model them. In the second phase, the UML activity diagram is transformed into a special kind of Petri Nets named *function nets* to form the test model. In the third phase the test model is annotated to specify start and terminating states and add annotations to transitions to improve readability. Next, abstract test cases are generated from the test model in phase 4 and concrete test cases are generated from the abstract ones in phase 5. Finally, the concrete test cases are executed in phase six. Mehmood and Khan [9] claim that their approach is better than the one proposed by Griebe and Gruhn [7] because it can generate a context-coupled test cases that can adapt their expected output when context changes

during test execution. In their approach, they claim that it's easy to incorporate new context variables unlike the static context modeling in [7].

TestAWARE [22] is a laboratory-oriented testing tool for mobile context-aware applications. The tool was designed to be an all-in testing tool for context-aware applications which support both black-box, white-box, functional and non-functional testing. The tool supports both emulator-based and device-based testing. The author stressed the absence of a tool that supports multiple contextual data types, data sources, testing methods (black-box, white-box functional, non-functional), and testing environments (emulator and device). The tool uses the record-and-replay to capture and replay contextual data during testing. Testers can choose different sources for context data. namely, real-time data fed directly from physical devices, historical data from online databases such as AWARE [23], or from previous experiments and data simulated by the tester. The author stresses that having a tool supporting multiple data sources can enable the testing of applications aiming at uncommon context and can significantly simplify the testing process.

Tse et al. [8] explored applying a white-box, data-flow-based approach to test context-aware middleware-centric applications. The authors stressed that the existing conventional data-flow adequacy criteria are not suitable to test context-ware middleware-centric programs as they have shown to fail to expose a context-aware-related fault in CM-centric programs, resulting in the overlooking of a context-aware sub-condition. According to the authors, there are three kinds of obstacles that hinder the effective application of standard data flow testing criteria to testing Context-aware Middleware-Centric (CM-Centric) software, namely:

(i) Context-aware faults: faults in the triggering logics in the middleware.

(ii) Environmental interplay: environmental updates may happen anytime, and test set should be updated in time accordingly.

(iii) Context-aware control flow: it is difficult to enumerate every control-flow trace of context changing for some situations.

They [8] expressed the need for novel data-flow adequacy criteria supporting context-aware applications. They proposed to distinguish context variables which are variables used for exchanging context with the environment from other variables when performing data-flow analysis for the program. They also

defined context-aware middleware-centric applications as a set of *context variables* (C), a set of *situations* (S), and a set of *adaptive actions* (U). A situation is a set of context variables that are of interest for the situation, a predicate that is evaluated with respect to the context variables, and an adaptive action that is triggered when the predicate evaluates to true. An adaptive action is an application function invoked directly by the middleware.

Aiming at reducing the cost and improving the scalability of testing, context-aware mobile apps Zhang et al. [24] developed a tool named *RainDrops*. *RainDrops* adopts a hybrid approach that utilizes both emulator-based and device-based testing. The mobile application runs in an emulator residing in the cloud. During test execution, if context data that can't be provided by the emulator is queried via a system API invocation, the tool intercept the system API and pauses the test execution. The system API invocation is offloaded to an available crowd-sourced physical device for results. Upon receiving the results from the physical device, test execution in the emulator resumes.

Intending to improve existing test suites, Wang et al. [25] developed an approach that automatically generates variations of test suites by statements associated with context changes. By analyzing the source code of an application, it generates the function call and control flow graphs to find such statements (marked as nodes in CFGs). Based on the flows containing these CFG nodes, it searches for variant test suites involving context conditions that cause different application behaviors.

Song et al. [26] presented a technique to generate test cases for Android context-aware applications by inspecting their requested permissions. Android applications contain a file called AndroidManifest.xml that contains the list of permissions the application needs in order to deliver its functionalities, such as accessing location services. Using this technique, all possible combinations of possible states of each requested resource are explored during testing.

In [27], Majchrzak et al. introduced a grey-box testing technique referred to as block-based context-sensitive testing. The source code of a context-aware application is split into blocks containing assertions. A test case can update the context so that assertions can examine context-aware behaviors. To create test cases automatically, generators are implemented.

To help developers detect and reproduce context-related bugs appearing in the wild after deployment, MOTIF[13] uses crowdsourced feedback to detect crash patterns. By analyzing crash patterns, MOTIF creates ad-hoc

test suites to attempt to reproduce the bugs on crowdsourced devices running in different contexts.

3.3 Analysis

Given the test space explosion imposed by the context-awareness of mobile applications, it becomes evident that any proposed solution should apply test automation. Testing activities, including test case generation and execution, should be automated whenever possible. Black-box model-based testing (MBT) techniques are usually used to help reduce testing efforts by automating test case generation [28].

The literature has several studies that propose model-based solutions to test context-aware applications. Griebe and Gruhn [7] have used UML Profile to extend some UML Activity Diagram elements to enable context modeling. Their approach consists mainly of using static context data in preconditions at control flow elements. We think such an approach of static modeling of test data lacks flexibility and is difficult to extend for more contexts.

Similar to [7], Mehmood and Khan [9] also used UML Activity Diagram to model context-aware mobile apps. They expressed that the default notation of the UML Activity Diagram is not suitable for behavioral modeling of context-aware applications and proposed extending it by adding a context-aware activity node. A context-aware activity node represents a functionality in the application that is context-aware. Using such a technique where context-aware functionalities are identified and explicitly marked in the model has a significant advantage. It simplifies the context-aware testing by focusing only on context-aware functionalities whenever context is concerned.

We have observed that existing model-based solutions[9][7] lack context coverage criteria to drive the automated test case generation. Coverage criteria are essential to the tester to get a sense of how much testing could be enough.

We also think having separate models for modeling the context and the behavior of the mobile application instead of adding context data directly to the behavioral model, as in [7], can be a better approach as both models can evolve and be maintained separately. In our solution, separate models are used for context and behavior.

To make a given solution easier to adopt for large mobile applications, a modular modeling technique can be used. For example, FSMWeb[29] is a

model-based testing tool for web applications where the *Hierarchal Finite State Machine (HFSM)* formalism [10] is used to model the application's behavior. Using HFSM to model mobile applications seems natural as mobile applications are usually built using a hierarchy of UI components, and interacting with these components consists of a hierarchy of user interactions. In our solution, we used context-enriched HFSM where context-aware states are explicitly marked and adaptations at these states are explicitly modeled.

4. Contribution: Context-driven Modeling

In this chapter, the modeling aspect of the proposed solution is presented. Firstly, we present the approach we used to create modeling languages that are used to build the context and behavioral model programs. Secondly, we present our context model. The chapter is concluded with the behavioral modeling.

4.1 Domain Specific Languages

A Domain Specific Language (DSL) is a programming language with a higher level of abstraction optimized for a specific class of problems. A DSL uses concepts and rules from a field or domain. A DSL can be created to be used as a modeling language for a specific domain. Compared to general-purpose programming languages such as Java, DSLs are usually less complex and intended to be used by non-software people (e.g., designers, domain experts).

In this project, we have chosen to create and use DSLs to instantiate the proposed context and behavioral models. The main advantages that motivated our use of DSLs are, firstly, a DSL tailored for a specific domain is often easy to learn and use by the experts in that domain and increases productivity as opposed to general-purpose programming languages, which are hard to learn by non-software people and involve a lot of redundancy which can be abstracted by the use of a DSL. Secondly, it enables the enforcement and validation of domain-specific constraints, which is otherwise hard or not possible using a graphical modeling notation such as UML. Lastly, a model

program written in a DSL can easily be transformed to other representations (e.g., Java model) in order to be conveniently analyzed.

On the other hand, using DSLs also have drawbacks. The most distinguished drawback is that building DSLs from scratch can be a tedious and expensive activity. However, there exist specialized tools that aim at simplifying the process of creating DSLs. A popular framework for creating DSLs is *Xtext*[2] which is part of the *Eclipse Modeling Framework*[3]. *Xtext* can be used to create DSLs either by supplying a metamodel or by defining a grammar for the language. Furthermore, *Xtext* automatically generates a parser and a text editor supporting syntax coloring and error highlighting.

In this project, we used *Xtext* to create two modeling languages, *Context-driven Modeling Language (CDML)* for behavioral modeling and *Context Definition Language (CDL)* for Context Modeling.

4.2 Context Modeling

In this section, we introduce the building blocks of the proposed context model and present *Context Definition Language (CDL)*, a DSL to create model programs conforming to the proposed context modeling approach.

4.2.1 Context Model

The central element in our context model is the context variable. A context variable represents a dimension of the mobile application's context, and a context-aware mobile application can have one or more context variables.

There exist different classifications of context variables in the literature (discussed in section 2.2). After reviewing the literature, we found that context classifications presented in [13] and [14] are the simplest and the most expressive, and we based our context model on them. In our context model, there are three types of context variables as follows:

(i) **Dynamic Context**: Context information acquired from sensors in high frequency. Changes in such context can occur unpredictability at any

[2] https://www.eclipse.org/Xtext/

[3] https://www.eclipse.org/modeling/emf/

point in time during the execution of the mobile application. An example of such context is the condition of network connectivity.

(ii) **Static Context**: Context information that do not change during the execution of the mobile application. Examples of such context the operating system and screen size.

(iii) **Derived Context**: Context information derived from other context variables. Such context presents a layer of abstraction and makes it easier to deal with context information independent of the low-level details on which it is based.

To put *derived context* into perspective, let's take, for example, the geographical location context provided by the GPS sensor, which is in the form of decimal numbers representing the latitude, the longitude, and sometimes the altitude of a geographical position. Such low-level context information may not be convenient to deal with directly. Therefore, a higher-level context can be derived from this low-level context to represent something more meaningful and specific for the mobile application. For example, the context variable *Country* - which represents the current country in which the user is located - can be derived from the location's latitude and longitude by defining some mapping rules.

Another concept adopted by our context model is *contextual situation* presented by Bettini et al. [16]. In context-aware applications, Situations are external semantic interpretation of context. Situations can be defined as a set of conditions on the context. When viewed from this perspective, situation are used to aggregate and model specific constraints that are to be recognized in order to perform adaptive actions. As highlighted by Bettini et al. [16], situation modeling is not intended to be exhaustive and not all possible contextual situation needs be explicitly modeled. In our approach, only situations that require adaptive actions are intended to be explicitly modeled.

Figure 2 shows the metamodel of the proposed context model. The root element of a context model is of the type ContextModel. ContextModel element contains elements of type Context, Situation, Provider, and DefinedType. Context elements can be either StaticContext, DynamicContext, or DerivedContext. DerivedContext elements reference one or more other Context elements from which it derives.

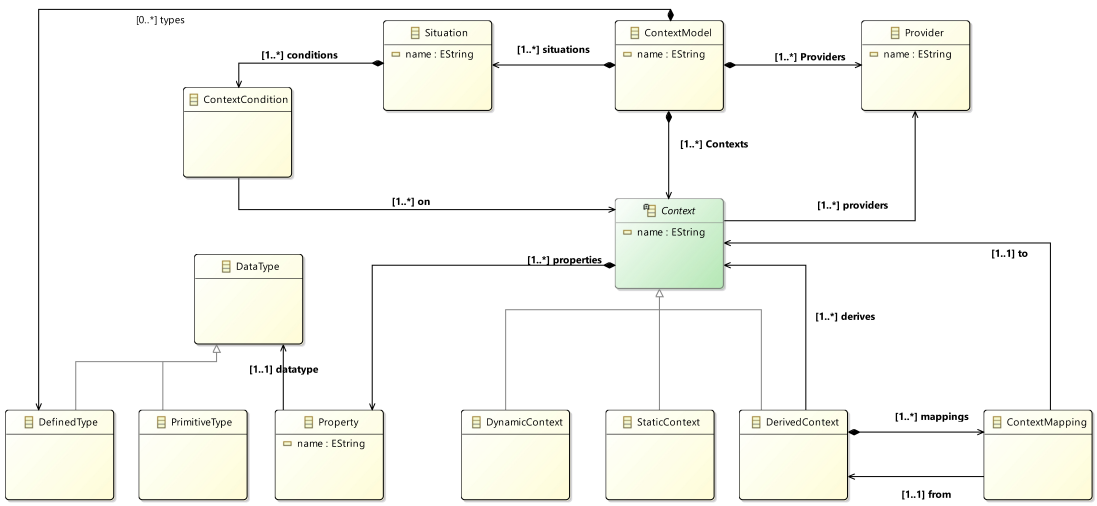

Figure 2: Metamodel of the Proposed Context Model

Context elements contain one or more Property elements, and a Property element is associated with a DataType element. Context elements are also associated with one or more Provider elements. DerivedContext elements contain one or more ContextMapping elements, and a ContextMapping element references two Context elements representing the two ends of the mapping rule. Situation elements contain one or more ContextCondition elements, and ContextCondition elements reference one or more Context elements on which the condition is based.

4.2.1 Context Definition Language (CDL)

To make it simpler to create context models and to use these models in the test case generation phase, we created a DSL called *Context Definition Language (CDL)* that captures the concepts of our context modeling approach as described in the previous subsection.

Using CDL, a ContextModel element is declared using the keyword "*contextModel*" followed by an identifier and a pair of curly brackets. The ContextModel element is the root element of the context model program and contains Context, Situation, Provider, and Type elements. Listing 1 shows an example of context model declaration.

Context elements represent context variables and are defined using the "*context*" keyword followed by an identifier. Listing 1 shows an example of a default context declaration. Prefixing the keyword *context* with the

```
contextModel TranslationApp {

    context LOCATION {
        providers: [GPS_SENSOR, CELL_ADAPTER],
        properties: [longitude: double, latitude: double],
    }

}
```

Listing 1. Context Model in CDL

keyword *"static"* is used to define a static context variable, as shown in Listing 2.

```
static context OS {
    providers: [DEVICE_API],
    properties: [os: OS],
}
```

Listing 2. Static Context Declaration

Derived context variables are declared by adding the keyword *"derives"* after the identifier of the context variable followed by one or more references to the other context variables from which it derives. For derived context variables, mappings from its possible values to values of contexts it derives can be specified as shown in Listing 3.

```
context COUNTRY derives LOCATION {
    properties: [country: Country],
    mappings: {
        country.france -> LOCATION.longitude == 48.8534 and LOCATION.latitude == 2.3522,
        country.germany -> LOCATION.longitude == 52.5200 and LOCATION.latitude == 13.4050,
        country.spain -> LOCATION.longitude == 40.4168 and LOCATION.latitude == 3.7038
    }
}
```

Listing 3. Derived Context Declaration

Context variables have one or more data properties. As shown in Listing 1, context variable *location* has two data properties, longitude and latitude. Data properties have datatypes that can be either a primitive datatype such as double or a user-defined enumeration datatype. The modeling language allows the declaration of enumeration datatype using the keyword "*type*"

followed by an identifier and values between a pair of curly brackets as shown in Listing 4.

```
type OS {android, ios}

type Country {france, germany, spain}

type Availability {available, unavailable}
```

Listing 4. Declaration of enumeration datatypes

Context variables are associated with *context providers.* Context providers are the source of context information; they can be a physical sensor, web API, a user profile, etc. For derived context variables, context providers are inherited from the context variables they derive and are not declared explicitly. Listing 1 shows two providers, GPS_SENSOR and CELL_ADAPTER linked to context variable *location.* Listing 5 shows how context providers are declared CDL.

```
providers {
    CELL_ADAPTER,
    GPS_SENSOR,
    DEVICE_API
}
```

Listing 5. Declaration of enumeration datatypes

For declaring *Situations*, the keyword *"situation"* is used and then followed by an identifier. Then, a condition or a set of conditions on one or more context variables are provided between a pair of curly brackets. Listing 6 shows an example of a Situation declaration.

```
situation INTERNET_DISCONNECTED {
    INTERNET_CONNECTIVITY.connectivity == offline
}
```

Listing 6. Declaration of Situation

In this section, we introduced our context modeling language, A DSL created using *Xtext* and based on concepts from the literature. We demonstrated how the language could be easily used to create context models. Context models are used as input to the automated test case generation phase introduced in chapter 5.

4.2 Behavioral Modeling

In this section, we present our behavioral modeling approach for context-aware mobile applications, which is based on an existing modeling approach for web applications using the *Hierarchal Finite State Machine* formalism [29]. We extend the existing approach by introducing *Context-Aware State,* which is a state where context influences the behavior of the mobile application. We also introduce *Adaptation Finite State Machine,* which is used to model an adaptive action at a context-aware state.

We also present our behavioral modeling language *Context-driven modeling language (CDML)*, a DSL to simplify creating model programs conforming to the proposed modeling approach. A behavioral model program written in CDML is input to the automated test case generation process presented in the chapter 5.

4.2.1 Context-enriched Hierarchal Finite State Machine

The use of *Hierarchal Finite State Machine (HFSM),* where a hierarchy of finite state machines (FSMs) is used to model mobile applications, feels natural. Mobile applications are usually built using a hierarchy of UI components and interacted with using a hierarchy of user interactions. In addition, having a hierarchy of finite state machines instead of a single one allows us to break the modeling problem into smaller sub-problems and makes the model modular and easier to maintain and extend. Such a model can scale for large applications without having a large and unmanageable model.

The behavioral modeling starts by partitioning the mobile application into *clusters*. A cluster represent a software module or an application component that represents a high-level functionality that abstract other lower-level functionalities. For example, for an application that requires the user to

be authenticated, the functionality of logging-in can be considered a high-level user interaction that abstract other lower-level interactions of entering a username, entering a password and pressing the login button. The identified application clusters form the top level FSM of the HFSM model, is called the *Abstract Finite State Machine* (*AFSM)* where each cluster is represented by a *super state* that is modeled by a lower-level FSM. Figure 3 shows the AFSM for a simple pizza delivery mobile application where the identified clusters are Login Activity, Main Activity, Past Orders Activity, New Order Activity and Exit.

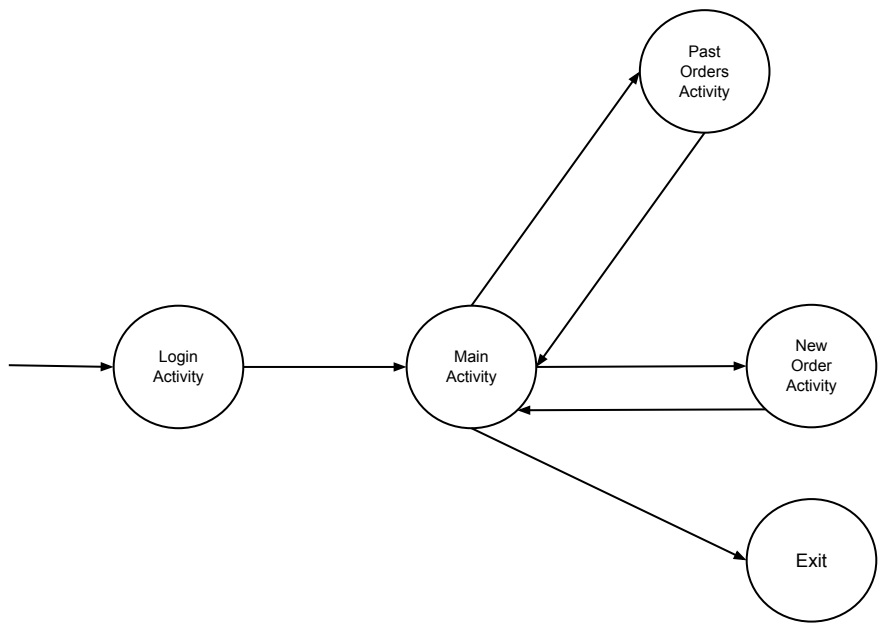

Figure 3. Abstract Finite State Machine for a Pizza Delivery Application

Super states in the AFSM are modeled by lower-level FSMs. These FSMs can also have super states that represent sub-clusters that need further modeling or atomic states. An *atomic state* represents states where single atomic user interaction is permitted (e.g., pressing a button). Modeling activity continues recursively until all state machines at the bottom of the hierarchy consist only of atomic states. Figure 4 shows the lower-level FSM that models the super state Login Activity in Figure 3.

The next step in the behavioral modeling is to identify and mark *context-aware states*. A context-aware state represent a point in the mobile app where

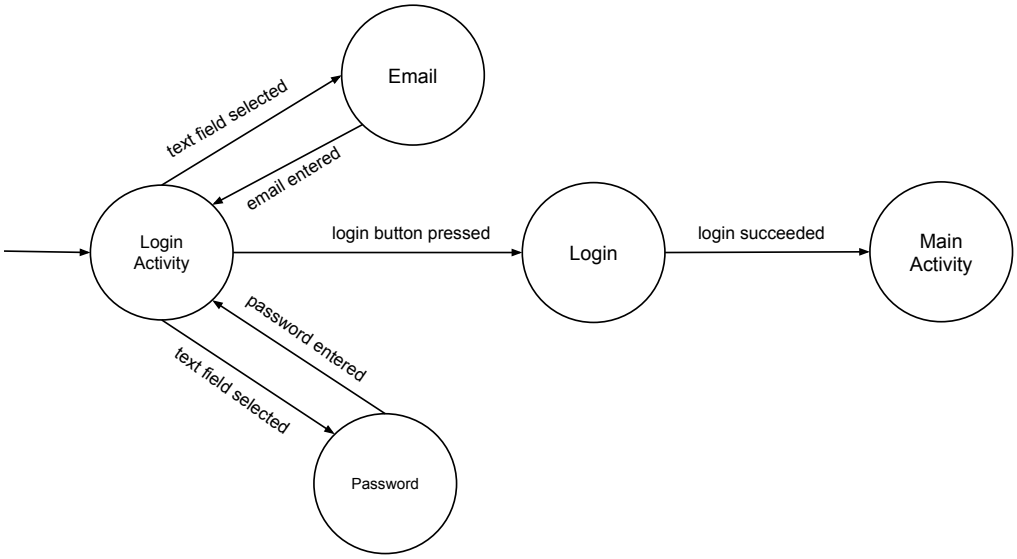

Figure 4. Login Activity Finite State Machine

its behavior is influenced not only by user input, but also by the available context information. Context-aware states also represent linking points between the behavioral model and the context model, where context variables defined in the context model are associated with context-aware states in the behavioral model.

State "Login" in Figure 4 represents a state where the mobile application is trying to authenticate the user by communicating with an authentication web service. Since authentication requires internet connection, state "Login" is considered a context-aware state depending on the context variable representing the internet connection conditions.

In the previous section (context modeling), we introduced the concept of *contextual situation* which can be defined as a set of conditions on the context. For a context variable representing the internet connection conditions, a set of situations can be identified, to name a few: disconnected, connected to WIFI, connected to the cell network, high latency. In our context modeling language, we choose only to explicitly define situations that influence the default behavior of the mobile applications. For example, a situation representing the unavailability of internet connection causes the authentication attempt at state "Login" in Figure 4 to fail and such situation requires an adaptation (e.g. displaying a warning message).

Adaptations are modeled using an *adaptation finite state machine* which is an FSM representing an adaptive action in response to a contextual

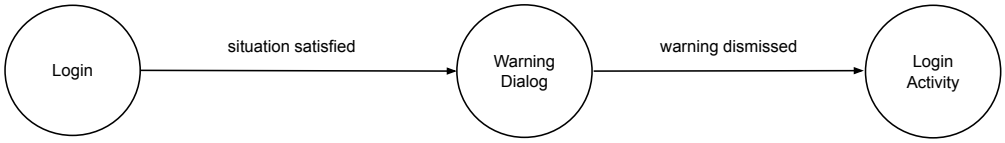

Figure 5. Adaptation FSM for Situation "Internet Disconnected" at State "Login" in Figure 4

situation being satisfied at a context-aware state. Figure 5 shows the adaptation FSM at state "Login" in Figure 4 when the internet is disconnected.

3.2.2 Context-driven Modeling Language (CDML)

In this section, we present the *context-driven modeling language (CDML),* a DSL we created using *Xtext* to create model programs conforming to the behavioral modeling approach we introduced in subsection 3.2.1. To demonstrate the language's features, we use it to model a simple location-based translation application where the target translation language is automatically determined according to the language spoken where the user is located. The application uses a web service to carry out the translation. Figure 6 shows the meta-model on which CDML is based.

The root element in the model program is the Model element, and is declared using the keyword "*model*" followed by an identifier and a pair of curly brackets. The Model element contains elements of type Context, Situation, StateMachine and Adaptation. Context elements are created by using the keyword "*contexts*" and then adding identifiers separated by comas between a pair curly brackets. Situation elements are declared similarly but using the keyword "*situations*". Context and Situation elements are references to Contexts and Situations defined in the context model. Listing 7 shows the model element of a simple translation application with Context and Situation elements defined.

StateMachine elements are declared inside a Model element using the keyword *"statemachine"* accompanied with an identifier and a pair of curly brackets. The first declared StateMachine element is deemed the Abstract Finite State Machine (the top of the hierarchy). StateMachine elements must

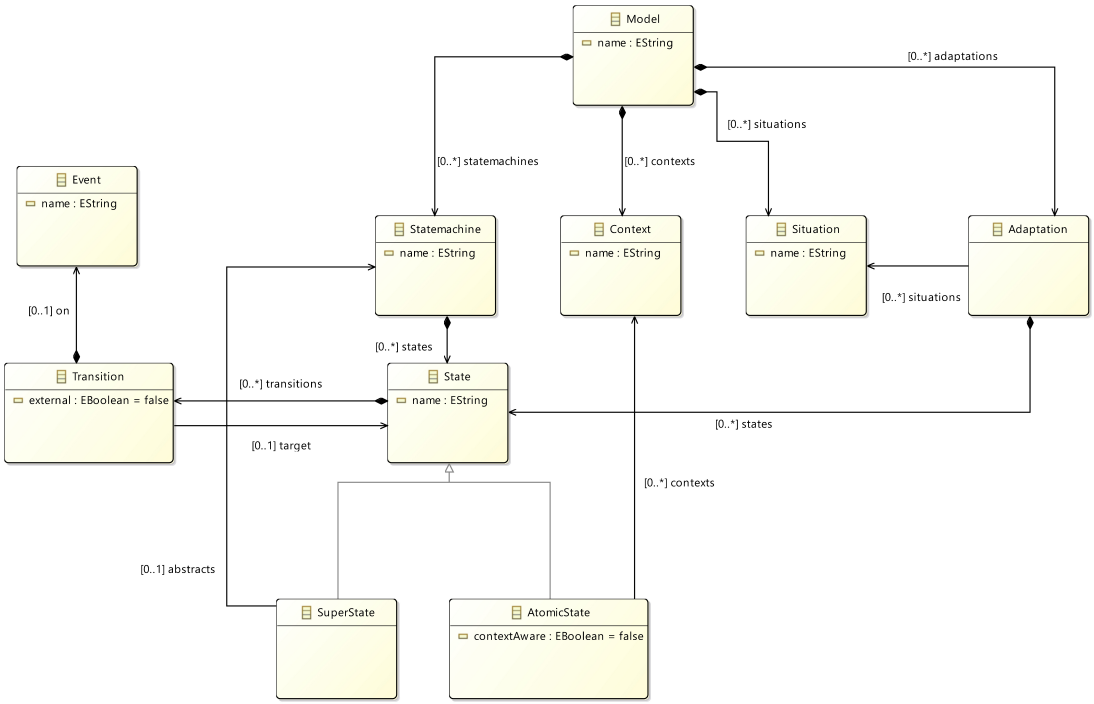

Figure 6. CDML Metamodel

```
model TranslationApp {
    contexts {
        INTERNET_CONNECTIVITY,
        LOCATION
    }

    situations {
        INTERNET_DISCONNECTED,
        LOCATION_UNAVAILABLE
    }
}
```

Listing 7. Model Element in CDML

contain one or more State elements. Listing 8 shows an example of a StateMachine declaration.

State elements can be of type AtomicState or SuperState; state elements are declared within the curly brackets of a StateMachine element using the keyword "*state*" followed by an identifier. To declare a State element as a SuperState, the keyword "*state*" is prefixed by the keyword *"super"*. Following

```
statemachine ABSTRACT_SM {
    state START {
        transition on APP_STARTED -> ENTER_PHRASE_ACTIVITY
    }
    super state ENTER_PHRASE_ACTIVITY abstracts ENTER_PHRASE_ACTIVITY_SM {
        transition on TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY
        transition on TERMINATE_BUTTON_CLICKED -> EXIT
    }
    super state VIEW_TRANSLATION_ACTIVITY abstracts VIEW_TRANSLATION_ACTIVITY_SM {
        transition on TERMINATE_BUTTON_CLICKED -> EXIT
        transition on BACK_BUTTON_CLICKED -> ENTER_PHRASE_ACTIVITY
    }
    state EXIT
}
```

Listing 8. StateMachine Element in CDML

its identifier, the keyword *"abstracts"* is used, followed by a reference to a StateMachine element.

For the declaration of context-aware State elements, the keyword "*state*" is used, followed by an identifier and then followed by the keyword "*awareof*." After the keyword "*awareof*" one or more references to Context elements separated by commas are added. Listing 9 shows an example of a context-aware State element.

```
state TRANSLATE awareof LOCATION, INTERNET_CONNECTIVITY {
    external transition on TRANSLATION_SUCCEEDED -> TRANSLATION_ACTIVITY
}
```

Listing 9. Context-aware State Element in CDML

State elements may contain zero or more Transition elements. Transition elements are declared between the curly braces of a State element by using the keyword "*transition*" followed by the keyword "*on*" which is followed by an identifier for the event that fires the transition. After the event's identifier, the reserved symbol "->" is used, followed by a reference to the target State element. Listing 8 shows examples of Transition elements.

A transition is considered external if its target State element is part of another StateMachine element. A Transition element is marked external by prefixing it with the keyword "*external*" as shown in Listing 9.

For the Adaptation elements creation, the keyword "*adaptation*" is used, followed by the keyword "*for*" followed by a reference to a Situation element. The Situation element reference is followed by the keyword "*at*" which is followed by a reference to a context-aware State element. Between curly

braces, State elements representing the adaptive action are added where the initial state must be the context-aware state that the adaptation is associated with. Listing 10 shows an example of an Adaptation element.

```
adaptation for INTERNET_DISCONNECTED at TRANSLATE {
    state TRANSLATE {
        transition on SITUATION_SATSFIED -> DISPLAY_WARNING
    }

    state DISPLAY_WARNING {
        external transition on WARNIMG_DISMISSED -> ENTER_PHRASE_ACTIVITY
    }
}
```

Listing 10. Adaptation Element in CDML

In this section, we presented the theoretical foundations for the proposed context-driven modeling approach for mobile applications. We introduced the *Context-driven Modeling Language (CMDL),* a DSL for creating model programs conforming to the proposed modeling approach. The next chapter will demonstrate how a context model program written in CDL and a behavioral model program written in CDML are used for automated abstract test case generation.

5. Test Case Generation

In this chapter, we present the process for automated test case generation, which is based on [29]. The process takes as inputs a context model program written in CDL and a behavioral model program written in CDML and produces abstract test cases as outputs. Executable test cases are out of the scope of this project and are considered for future work.

The first phase consists of generating test subpaths for all the FSMs of the behavioral model, the generation is driven by test requirements expressed in terms of coverage criteria. These subpaths are then combined to form complete test paths. Finally, test paths are transformed into abstract test cases. Figure 7 shows the test case generation process.

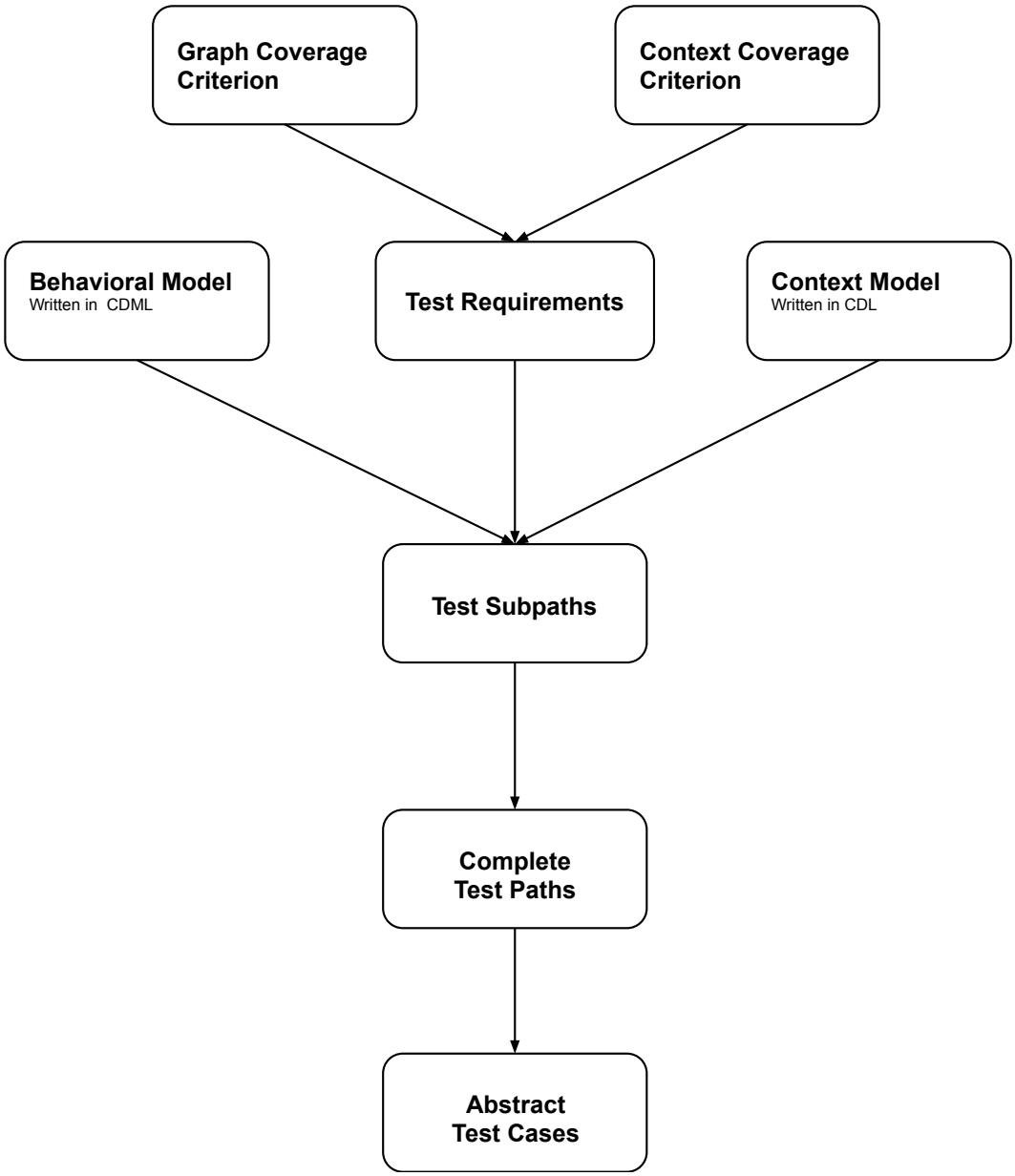

Figure 7. Test case generation process

5.1 Contribution: Test Coverage Criteria

We use test coverage criteria to drive the test sub-paths generation through the finite state machines of the hierarchal state machine model. Since a finite state machine can contain cycles, the set of possible paths it contains can be infinite. To overcome that, a transition-based coverage criterion is

selected to terminate the paths exploration when the criterion is met. The most notable transition-based coverage criterions for state machines are [30]:

All-States: A test suite satisfy All-States coverage criterion if all states in the finite state machine model are visited at least once.

All-Transitions: A test suite satisfies the All-Transitions coverage criterion if all possible transition sequences of length up to one are traversed at least once. Satisfying All-Transition criterion satisfies All-States criterion.

All-Transitions-Pairs: A test suite satisfies the All-Transitions-Pairs coverage criterion if all possible transition sequences of length up to two are traversed at least once.

The use of only transition-based coverage criteria to generate test cases for context-aware mobile applications is insufficient. Such coverage criteria would only target the behavioral model to cover possible execution paths. We are also interested in covering such executions under different context conditions of the mobile application as defined in the context model. Therefore, we are also introducing new coverage criteria targeting context to be used along transition-based converge criteria to generate context-dependent test cases:

Definition 1. **All-Situations Coverage Criterion:** A test suite satisfies the All-Situation coverage criterion if all Situations belonging to a context variable (as defined in the context model) are explored at the context-aware state associated with that context variable.

Definition 2. **All-Situation-Pairs Coverage Criterion:** A test suite satisfies the All-Situation-Pairs coverage criteria if all possible switches between a pair of Situations belonging to a context variable are explored at the context-aware state associated with that context variable.

For example, given the context variable C, that represents "Internet Connection" with the following set of Situations: {*offline*, *wifi*, *3G*, *4G*, *high* latency, *timedOut}* and given a context-aware state S where internet is accessed to perform an action. To satisfy the All-

Situations criterion, all Situations must be explored at least once at the context-aware state S.

Using the same example, the All-Situation-Pairs criterion is satisfied if all possible switches between Situations of context variable C are explored at context-aware state S.

In next section, we discuss the test subpath generation step which is driven by a pair of transition-based criteria and context-based criteria.

5.2 Test Subpath Generation

In this step, test subpaths are generated for each finite state machine in the context-enriched hierarchal finite state machine behavioral model. First, a pair of transition-based criteria (e.g All-Transitions) and context-based criteria (e.g All-Situations) are selected, then, the extraction of test paths begins until both coverage criteria are satisfied.

Given the FSM written in CDML in Listing 10, when selecting All-Transitions and All-Situations criteria, Table 1 shows the produced test subpaths. In this case, since there are no context-aware states, satisfying only All-Transitions is sufficient.

```
statemachine ABSTRACT_SM {
    state START {
        transition on APP_STARTED -> ENTER_PHRASE_ACTIVITY
    }
    super state ENTER_PHRASE_ACTIVITY abstracts ENTER_PHRASE_ACTIVITY_SM {
        transition on TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY
        transition on TERMINATE_BUTTON_CLICKED -> EXIT
    }
    super state VIEW_TRANSLATION_ACTIVITY abstracts VIEW_TRANSLATION_ACTIVITY_SM {
        transition on TERMINATE_BUTTON_CLICKED -> EXIT
        transition on BACK_BUTTON_CLICKED -> ENTER_PHRASE_ACTIVITY
    }
    state EXIT
}
```

Listing 10. An FSM Written in CDML

Table 1. Test Subpaths of the FSM In Listing 10

Subpaths	
SP1	START -> APP_STARTED -> ENTER_PHRASE_ACTIVITY -> TERMINATE_BUTTON_CLICKED -> EXIT

Subpaths	
SP2	START -> APP_STARTED -> ENTER_PHRASE_ACTIVITY -> TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY -> TERMINATE_BUTTON_CLICKED -> EXIT
SP2	START -> APP_STARTED -> ENTER_PHRASE_ACTIVITY -> TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY -> BACK_BUTTON_CLICKED -> ENTER_PHRASE_ACTIVITY -> TERMINATE_BUTTON_CLICKED -> EXIT

In case the FSM contains context-aware states, context conditions corresponding to context Situations, as defined in the context model program, are added as constraints in the produced test subpaths. If the context-aware state has Adaptations linked to it, the FSM corresponding the the Adaptation is also explored. Listing 11 shows a FSM written in CDML which represents the super state ENTER_PHRASE_ACTIVITY in the higher-level FSM shown in Listing 10. The FSM in Listing 11 has the context-aware state

```
statemachine ENTER_PHRASE_ACTIVITY_SM {
    state ENTER_PHRASE_ACTIVITY {
        transition on PHRASE_TEXTFIELED_SELECTED -> ENTER_PHRASE
        transition on TRANSLATE_BUTTON_CLICKED -> TRANSLATE
        external transition on TERMINATE_BUTTON_CLICKED -> EXIT
    }
    state ENTER_PHRASE {
        transition on PHRASE_ENTERED -> ENTER_PHRASE_ACTIVITY
    }

    state TRANSLATE awareof INTERNET_CONNECTIVITY {
        external transition on TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY
    }
}

adaptation for INTERNET_DISCONNECTED at TRANSLATE {
    state TRANSLATE {
        transition on SITUATION_SATSFIED -> DISPLAY_WARNING
    }

    state DISPLAY_WARNING {
        external transition on WARNIMG_DISMISSED -> ENTER_PHRASE_ACTIVITY
    }
}
```

Listing 11. Excerpt From A Model Program Written In CDML Containing A FSM That Has A Context-Aware State And An Adaptation At That State.

TRANSLATE that is associated with the context variable INTERNET_CONNECTIVITY. Listing 11 also shows Adaptation FSM for Situation INTERNET_DISCONNECTED. Listing 12 shows the excerpt of the context model program where context variable

```
type Connectivity {offline, wifi, slow3G, fast3G, _4g, high_latency}

context INTERNET_CONNECTIVITY {
    providers: [WIFI_ADAPTER, CELL_ADAPTER],
    properties: [connectivity: Connectivity],
}

situation INTERNET_DISCONNECTED {
    INTERNET_CONNECTIVITY.connectivity == offline
}
```

Listing 12. Excerpt from a Context Model Program Written in CDL Containing the Context Variable INTERNET_CONNECTIVITY and Situation INTERNET_DISCONNECTED

INTERNET_CONNECTIVITY and Situation INTERNET_DISCONNECTED are defined.

Table 2 shows the generated test subpaths for the FSM in Listing 11 when the All-Transitions criterion is used for transition-based coverage and All-Situations criterion for context-based coverage. Context constraints are added between brackets.

Table 2: Test Subpaths of the FSM in Listing 11

Subpaths	
SP1	ENTER_PHRASE_ACTIVITY -> PHRASE_TEXTFIELD_SELECTED -> PHRASE_ENTERED -> ENTER_PHRASE_ACTIVITY -> TRANSLATE_BUTTON_CLICKED -> [INTERNET_CONNECTIVITY==wifi] TRANSLATE -> TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY
SP2	ENTER_PHRASE_ACTIVITY -> PHRASE_TEXTFIELD_SELECTED -> PHRASE_ENTERED -> ENTER_PHRASE_ACTIVITY -> TRANSLATE_BUTTON_CLICKED -> [INTERNET_CONNECTIVITY==slow3G] TRANSLATE -> TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY
SP3	ENTER_PHRASE_ACTIVITY -> PHRASE_TEXTFIELD_SELECTED -> PHRASE_ENTERED -> ENTER_PHRASE_ACTIVITY -> TRANSLATE_BUTTON_CLICKED -> [INTERNET_CONNECTIVITY==fast3G] TRANSLATE -> TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY

Subpaths	
SP4	ENTER_PHRASE_ACTIVITY -> PHRASE_TEXTFIELD_SELECTED -> PHRASE_ENTERED -> ENTER_PHRASE_ACTIVITY -> TRANSLATE_BUTTON_CLICKED -> [INTERNET_CONNECTIVITY==4G] TRANSLATE -> TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY
SP5	ENTER_PHRASE_ACTIVITY -> PHRASE_TEXTFIELD_SELECTED -> PHRASE_ENTERED -> ENTER_PHRASE_ACTIVITY -> TRANSLATE_BUTTON_CLICKED -> [INTERNET_CONNECTIVITY==highLatency] TRANSLATE -> TRANSLATION_SUCCEEDED -> VIEW_TRANSLATION_ACTIVITY
SP6	ENTER_PHRASE_ACTIVITY -> PHRASE_TEXTFIELD_SELECTED -> PHRASE_ENTERED -> ENTER_PHRASE_ACTIVITY -> TRANSLATE_BUTTON_CLICKED -> [INTERNET_CONNECTIVITY==offline] TRANSLATE -> DISPLAY_WARNING -> WARNING_DISMISSED -> ENTER_PHRASE_ACTIVITY -> TERMINATE_BUTTON_CLICKED -> EXIT

In this section, we presented how test subpaths are generated for the different levels of FSMs in behavioral model. In the next section, we show how these subpaths are combined to form complete test paths.

5.3 Test Path Aggregation

In this section, we present the process of aggregating test subpaths generated in the previous step to form complete test paths. The process start with the subpaths generated from the AFSM (the top level of the hierarchy) and recursively replace super states with their corresponding test subpaths generated from the lower-level FSM they represent until all produced test paths are free from super states. Algorithm 1 shows the procedure of generating aggregated test paths.

For subpath **SP1** in Table 1 that has one super state ENTER_PHRASE_ACTIVITY, which is modeled by the FSM that belongs to the subpaths in Table 2, following Algorithm 1, the super state is replaced by **SP6** in Table 2. The resultant aggregated path is shown in Listing 13.

```
Input: AFSM + FSMs test subpaths
Output: Set of of aggregated test paths
unfinishedPaths = Subpaths of AFSM
finishedPaths = {}
while unfinishedPaths not empty:
    currentPath = get path from unfinishedPaths
    superState = get super state from currentPath
    for subpath in subpaths of FSM modeling superState:
        if superState.nextTransition == subPath.LastTransition:
            newPath = replace superState with subpath
            if newPath has super states:
                add newPath to unfinishedPaths
            else:
                add newPath to finishedPaths
            endif
        endif
```

Algorithm 1: Test Path Aggregation

```
START -> APP_STARTED -> ENTER_PHRASE_ACTIVITY
-> PHRASE_TEXTFIELD_SELECTED -> ENTER_PHRASE
-> PHRASE_ENTERED -> ENTER_PHRASE_ACTIVITY ->
TRANSLATE_BUTTON_CLICKED ->
[INTERNET_CONNECTIVITY==offline] TRANSLATE ->
DISPLAY_WARNING -> WARNING_DISMISSED ->
ENTER_PHRASE_ACTIVITY ->
TERMINATE_BUTTON_CLICKED -> EXIT
```

Listing 13: A Complete Test Path

5.4 Abstract Test Case Generation

In this step, complete test paths are converted to abstract test cases. We have chosen to use *Cucumber*[4] to express abstract test cases based on an approach presented in [28]. *Cucumber* is a Behavioral Driven Development (BDD) tool widely used in the industry. Testers use Cucumber to manually write test scenarios in a simple logical language called Gherkin. Then mappings from steps in the scenarios to executable test code for different platforms can be defined for test automation. Listing 14 shows an example of a Cucumber scenario written in Gherkin. The keywords *Given, And, When,* and T*hen* are used to give clearness and structure to the expected software behaviors.

Scenario: User Login

Given app has started
And login screen is displayed
Then user enters his username
And user enters his password
When user presses on login button
And authentication succeeded
Then user is directed to main screen

Listing 14: A Cucumber Scenario Example

Automatically generated Cucumber scenarios are maybe not as readable as manually written ones. Still, they serve their purpose of enabling easy mapping from the steps of a scenario to corresponding executable test code for the target platform.

To convert the complete test paths - which consists of states, transitions, and context constraints - to Cucumber scenarios, we followed a process similar

[4] https://cucumber.io/

to the one presented in [28]. Test paths are converted to Cucumber scenarios and included in a single Cucumber feature file. The conversion rules are as follows:

1. The initial state in the test path is ignored.
2. The first transition in the test path is converted to a Cucumber *Given* step.
3. The state following the first transition is converted to a Cucumber *And* step.
4. For the rest of transitions and states, transitions are converted to Cucumber *When* step and states to Cucumber *Then* steps.
5. Context constraints are converted to Cucumber *And* steps and are added before the state they target.

Scenario: Attempt to translate when internet is disconnected.

Given APP_STARTED
And ENTER_PHRASE_ACTIVITY
When PHRASE_TEXTFIELD_SELECTED
Then ENTER_PHRASE
When PHRASE_ENTERED
Then ENTER_PHRASE_ACTIVITY
When TRANSLATE_BUTTON_CLICKED
And **[INTERNET_CONNECTIVITY==offline]**
Then TRANSLATE
When TRANSLATION_FAILED
Then DISPLAY_WARNING
When WARNING_DISMISSED
Then ENTER_PHRASE_ACTIVITY
When TERMINATE_BUTTON_CLICKED
Then EXIT

Listing 15: Cucumber Scenario for the Test Path in Listing 13

Applying these rules to the test path in Listing 13 results in the Cucumber Scenario as shown in Listing 15.

6. Conclusion

Mobile apps that deliver context-aware functionalities are now widely available and used; however, these apps are burdensome to test and vulnerable to faults that are difficult to catch and reproduce. Aiming at simplifying, structuring, and automating the generation of context-depending test cases, we introduced in this report a context-driven model-based testing approach for context-aware mobile apps. We first introduced our context model where context variables representing the mobile app's context are defined together with their structure, relationship between them, and interesting contextual situations based on them. We also presented *Context Definition Language (CDL),* a domain-specific language (DSL) to create context model programs conforming to our context modeling approach.

Then we introduced our behavioral modeling technique, which uses a special Hierarchal Finite State Machine (HFSM) that has *context-aware states* which represent points where the context may influence the behavior of the mobile app. *Context-aware states* are linked with their corresponding context variables (defined in the context model). An *Adaptation FSM* is used to model an adaptive action at a context-aware state for situations (defined in the context model) that require the mobile app to adapt its behavior. Similar to the context model, we introduced *Context-driven Modeling Language (CDML)*, a DSL to create behavioral model programs conforming to our behavioral model approach.

We also presented the automated test case generation process, which takes as input a behavioral model program written in CMDL and its corresponding context model program written in CDL. The automated test case generation is driven by test coverage criteria; since we modeled context separately, using only transition-based coverage criteria is insufficient. Therefore, we introduced context-based coverage criteria (*All-Situations* and *All-Situation-Pairs*) to be used together with transition-based criteria to drive the automated test case generation.

6.1 Future Work

Due to time constraints, we could not finish building the automated test case generation tool at the time of writing this report. Thus, experimentation and validation of the proposed solution are considered for future work. First, we want to validate our solution on the simple Android location-based translation app used to illustrate our modeling approach in Chapter 4. In the next step, for checking the scalability of our modeling approach, we want to validate our solution on medium and large-sized mobile apps that use more complex context variables.

Generating executable test cases and test execution is out of the scope of this research project. We would like in future work to review the available test automation tools for mobile apps that possess the most capabilities of simulating various context conditions and select one tool to test execution. Abstract test cases (Cucumber scenarios) would be transformed into executable test cases for the target test automation tool.

The context-based test converge criteria we proposed in this report is rather primitive; the first criterion, *All-Situations*, covers all contextual situations of a context variable at context-aware states associated with it. The *All-Situations-Pairs* criterion covers all possible switches between these contextual situations. We would like to develop more sophisticated criteria to cover more complex scenarios involving context which may cause faults.

Bibliography

1. Statista, R.D. *Number of smartphone users worldwide from 2016 to 2026*. 2021 [cited 2021 May 26, 2021]; Available from: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

2. Statista, R.D. *Number of apps available in leading app stores*. 2021 [cited 2021 March 26, 2021]; Available from: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

3. Statista, R.D. *Annual number of global mobile app downloads 2016-2020*. 2021 [cited 2021 May 26, 2021]; Available from: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/.

4. Statista, R.D. *Worldwide mobile app revenues*. 2021 [cited 2021 May 26, 2021]; Available from: https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/.

5. Muccini, H., A. Francesco, and P. Esposito, *Software testing of mobile applications: Challenges and future research directions.* 2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings, 2012.

6. Myers, G.J., C. Sandler, and T. Badgett, *The Art of Software Testing*. 2011: Wiley Publishing.

7. Griebe, T. and V. Gruhn, *A model-based approach to test automation for context-aware mobile applications*, in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 2014, Association for Computing Machinery: Gyeongju, Republic of Korea. p. 420–427.

8. Lu, H., W.K. Chan, and T.H. Tse, Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation, in Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. 2006, Association for Computing Machinery: Portland, Oregon, USA. p. 242–252.

9. Mehmood, A. and M. Khan, An Automated Functional Testing Framework for Context-Aware Applications. IEEE Access, 2018. **PP**: p. 1-1.

10. Alur, R., S. Kannan, and M. Yannakakis, *Communicating Hierarchical State Machines*. Vol. 1644. 1999.

11. Gao, J., et al., Mobile Application Testing: A Tutorial. Computer, 2014. **47**: p. 46-55.

12. Dey, A. and G. Abowd, Towards a Better Understanding of Context and Context-Awareness. 2000. 304-307.

13. Gomez, M., R. Rouvoy, and L. Seinturier, Reproducing Context-sensitive Crashes in Mobile Apps using Crowdsourced Debugging. 2015.

14. Chen, G. and D. Kotz, A Survey of Context-Aware Mobile Computing Research. 2000, Dartmouth College.

15. Marcu, M., N. Ghiata, and V. Cretu, Extracting high-level user context from low-level mobile sensors data. 2013. 449-454.

16. Bettini, C., et al., A survey of context modelling and reasoning techniques. Pervasive and Mobile Computing, 2010. **6**(2): p. 161-180.

17. Utting, M., A. Pretschner, and B. Legeard, *A taxonomy of model-based testing approaches.* Software Testing, Verification and Reliability, 2012. **22**(5): p. 297-312.

18. Garzon, S.R., et al., CATLES: a crowdsensing-supported interactive world-scale environment simulator for context-aware systems, in Proceedings of the International Conference on Mobile Software Engineering and Systems. 2016, Association for Computing Machinery: Austin, Texas. p. 77–87.

19. Matalonga, S., F. Rodrigues, and G. Travassos, Challenges in Testing Context Aware Software Systems. 2015.

20. Almeida, D.R., P.D.L. Machado, and W.L. Andrade, Testing tools for Android context-aware applications: a systematic mapping. Journal of the Brazilian Computer Society, 2019. **25**(1): p. 12.

21. Yu, L., et al., Generating Test Cases for Context-Aware Applications Using Bigraphs, in Proceedings of the 2014 Eighth International Conference on Software Security and Reliability. 2014, IEEE Computer Society. p. 137–146.

22. Luo, C., et al., TestAWARE: A Laboratory-Oriented Testing Tool for Mobile Context-Aware Applications. Proc. ACM Interact. Mob. Wearable Ubiquitous Technol., 2017. **1**(3): p. Article 80.

23. Ferreira, D., V. Kostakos, and A.K. Dey, AWARE: Mobile Context Instrumentation Framework. Frontiers in ICT, 2015. **2**(6).

24. Zhang, L.L., et al., Towards A Contextual and Scalable Automated-testing Service for Mobile Apps, in Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications. 2017, Association for Computing Machinery: Sonoma, CA, USA. p. 97–102.

25. Wang, Z., S. Elbaum, and D.S. Rosenblum, Automated Generation of Context-Aware Tests, in Proceedings of the 29th international conference on Software Engineering. 2007, IEEE Computer Society. p. 406–415.

26. Song, K., et al., Generating various contexts from permissions for testing Android applications. 2015. 87-92.

27. Majchrzak, T.A. and M. Schulte, Context-Dependent Testing of Applications for Mobile Devices. Open Journal of Web Technologies (OJWT), 2015. **2**: p. 27-39.

28. Li, N., A. Escalona, and T. Kamal. Skyfire: Model-Based Testing with Cucumber. in 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). 2016.

29. Andrews, A., J. Offutt, and R. Alexander, Testing Web applications by modeling with FSMs. Software and System Modeling, 2005. **4**: p. 326-345.

30. Fakult‰t, D.d.M.-N., et al. Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines. 2010.