Method and Tools for Automated End-to-end Testing of Applications for Sailfish OS

Andrey Vasilyev, Ilya Paramonov, Sergey Averkiev
P.G. Demidov Yaroslavl State University
Yaroslavl, Russia
{Andrey.Vasilyev, Ilya.Paramonov}@fruct.org, exanimoso@yandex.ru

Abstract—**The automated end-to-end testing of applications allows to detect regressions early during the development and provide solid foundation for future modifications. However, implementation of such tests for mobile applications on Sailfish OS platform is related to some issues, especially when the application contain custom QML components written in C++. In the paper the authors present a method to resolve these issues, including two approaches to provide custom QML types in the testing environment and corresponding architectural considerations that make the code testable. The authors also describe an open source tool for running end-to-end tests on the integration server that supports the described method and supplements tooling of the Sailfish OS SDK.**

I. Introduction

Application testing is one of the key elements of successful software development [1]. It allows to ensure correctness of functions of the application under development, prevent error proliferation, decrease risks by capturing regressions on early stages and dealing with them in a timed manner, and so on. End-to-end testing is a variety of testing method aimed at checking correctness of the application flow from the user's point of view. Usually end-to-end testing involves typical scenarios of the application usage, and the possibility to execute these scenarios according to their specification is considered as passing the test.

Sailfish OS is a rather new, promising operating system mainly targeted at mobile platforms. According to the official website (*https://sailfishos.org*), it is positioned as a "true independent mobile OS," based on open source and developed by the Finnish company Jolla Ltd. and the Sailfish OS community. Since application development for Sailfish OS is primarily realized using Qt Framework (*https://www.qt.io*), most of testing means provided by this framework are also suitable for testing Sailfish OS applications. However, there are some specificities that make end-to-end testing of mobile applications for this platform not so straightforward as it could be expected. Additionally, at the moment there is no official guide on how to combine the available tools to organize such testing fully automatic to effectively leverage the developers' efforts.

In this paper we try to overview the possible troubles of automated end-to-end testing of applications for Sailfish OS and propose a method that can be used to organize such kind of testing applicable for a rather wide class of mobile applications using a combination of existing tools. We also discuss some architectural details that should be considered to make the application painlessly testable. We also take up questions of making our method ready for continuous integration that gains more and more important role in modern software engineering practice [2].

The paper is structured as follows. Section II presents the idea of automated end-to-end testing in details and contains overview of the related work, mainly including approaches available on other platforms. In Section III we describe our method of automated end-to-end application testing for Sailfish OS and consider two architectural models of a rather common Sailfish OS application with explanation on how to apply the method when each of these models is utilized. Section IV contains a detailed description of the organization of such an application including structures of project, QML extensions, and test runner. Section V is devoted to execution of the tests on a continuous integration server and presents the tool developed by the authors of this paper to facilitate this process for developers and system administrators. In conclusion we briefly overview the main results of the paper and formulate the future work directions.

II. Background and Related Work

The purpose of the end-to-end testing is to check that the application allows the user to perform specified scenarios, e.g., add a new record, calculate statistics, generate a report, and so on [3]. These kind of tests can be executed either manually or automatically.

During the manual testing one goes through a scenario and performs the steps described in a reference sheet. While the manual testing can be beneficial in terms of variety of checks, which the tester can perform, such an approach is often inefficient, error-prone, and expensive.

To overcome these issues testers resort to automation techniques and tools that allow to describe scenarios to be tested in the form of either half-formalized texts or just testing code that can be executed automatically. The main obstacle in development of these tools is due to the necessity for test tool to perform actions in the way the end user does, whereas many environments and frameworks for software development are simply not designed to allow such interactions. This issue affects the quality and flexibility of the corresponding testing tools, in turn, holding back their adoption. That is why, the area of test automation is a popular topic of modern research [4].

Despite all these issues, nowadays there are many rather mature tools useful for automated end-to-end testing. For example, Selenium (*http://www.seleniumhq.org*) is a standard de facto for automated testing of web applications [5]. Another example is Robotium (*https://github.com/RobotiumTech/robotium*) — the most advanced framework for end-to-end testing for Android OS [6]. The maturity of these tools is reflected by the fact of existing of "cookbooks" (e.g., [7]) that provide developers with ready-to-use recipes on how to solve typical testing tasks.

Unfortunately, this is not the case regarding automated end-to-end testing of applications for Sailfish OS, mostly due to its insufficient prevalence. However, as it is shown in the following sections, the majority of the required tools exists and the most of obstacles on the way of such testing can be successfully overcome.

III. METHOD OF AUTOMATED END-TO-END APPLICATION TESTING FOR SAILFISH OS

A. *Tools for automated testing using Qt framework*

As it is already mentioned, mobile applications for Salifish OS are developed using the Qt framework. The user interface of the application is developed using the Qt Quick technology that is targeted at rapid application development using the declarative QML language for interface description and the high-level JavaScript language for domain-level logic implementation. If needed, computation-intensive and custom QML components missing in the core library can be implemented in C++.

The Qt framework provides tools to run tests written for QML components. It consists of a QML library called QtTest and the qmltestrunner tool [8]. The latter one is used to run tests written using the foremost library. The tests using this library are developed as follows. The developer creates a separate QML file that imports the component to be tested and the QtTest library. The component under test is usually declared as a root element of the test, but this is not mandatory. The TestSuite component is declared as a child of the root component in current file.

Then, the developer creates functions in the declared TestSuite component. Functions starting with `test_` or `tst_` are treated as tests to be run by the test runner. To execute actions on behalf of the user, including button pressing, swiping, and typing on the keyboard, the functions provided by QtTest and QML language are used. QtTest also provides methods to assert statements, skip tests, sleep for a certain amount of time, and wait for an item to be rendered.

To run tests the developer passes paths to the created QML files as arguments of the testing tool. It collects tests from all files and execute them. Results of the execution can be formatted in a human-readable format or presented in one of common unit test report formats.

B. *Issues of end-to-end testing of Sailfish OS applications with standard tools*

Unfortunately, the application of the approach described in the previous section faces some issues when applied to end-to-end testing of mobile applications for Sailfish OS. Particularly:

- QML components declared in the root of the created QML document are not recreated for each test. Therefore, the developer has to restore the state of the element under test manually or create it dynamically.
- The functions to interact with QML components provided by QtTest are of low-level, e.g. they allow to emulate mouse move, mouse press, key click, mouse wheel and so on. To test a typical mobile application such a set is not enough, therefore the developer should write his/her own high-level functions to make tests easier to read and maintain. For example, functions to scroll a list to the specified element, find element showing some text, long click on the element are very much desirable but they are missing in QtTest.
- Sailfish OS platform that uses custom library called Silica to provide the user interface. This library is available only for Sailfish OS and cannot be easily used on other platforms. This fact prevents developers from running tests on the developer's computer used to develop an application and forces them to run tests either on the emulator or on the real device. Here emerges the additional task of deployment of the tests on such platforms.
- The tool can be easily used to test QML-only components depending on built-in QML types and installed QML libraries, whereas testing of custom C++-based QML components is complicated. The main reason is that native components require extra setup to be available during the testing opposed to just invoking tests along the pure QML components.

The first two issues can be solved rather straightforward. The way of coping with the third one is touched upon in Section V. Regarding the last issue, the developer can resort to one of two approaches: to use the qmltestrunner tool or create separate executable using a special harness. They are described in details in the following two subsections.

C. *Using the qmltestrunner tool for testing custom C++-based QML components*

When using this approach, qmltestrunner executes tests contained in QML files. In order to use custom QML components written on C++ in tests, the developer should create a QML Extension plugin that incorporates the components and register it in the QML engine.

The application uses the QML engine to setup the enviroment for instantiating custom QML components. It is used to search and load basic QML components of the QtQuick library.

Most of the basic QML components are implemented using C++ language for performance reasons. During the initialization of the QML engine it looks for a native QML Extensions in special directories and initializes them. A set of directories can be configured using the C++ `addImportPath()` method on the `QQmlEngine` class instance with path to the QML Extension as method parameter or environment variable QML2_IMPORT_PATH. The first one can be used in the application to set the directory where to load the QML Extension from, the latter one during the execution of the tests to allow qmltestrunner to search for QML Extension in the specified path.

In order to create a QML extension developer should subclass the `QQmlExtensionPlugin` class. He should override the `registerTypes()` method and register custom QML types there by URI. Developer should provide the directory where plugin should be installed and URI to the plugin within the directory. With this approach developer is able to register custom QML components when running qmltestrunner since the registration code is moved to the plugin.

QML Extensions allow not only register custom QML types, but provide a method to setup environment in which those types can be used. Developer may override the `initializeEngine()` method to initialize the QML Extension plugin. This method can be used to set the context properties of the engine or create application directories. It takes two parameters: engine instance and URI of the plugin.

D. Involving separate executable for testing custom C++-based QML components

Another approach proposes creating a separate executable to run the tests. This allows developer to register custom QML types without creating a QML Extension plugin. This approach proposes creating alternative entry point of the application.

Alternative entry point of the application is created by defining another main function. The function is used to setup test enviroment and to register custom QML types in the same way as in `registerTypes()` and `initializeEngine()` methods as mentioned before.

After defining the main function developer should run tests using it. It is achieved by putting the `quick_test_main` macros at the end of the function. It takes a set of arguments required to setup the tests: `argc` and `argv` from main function parameters, name of the test set shown when running the tests and location of the directory with the tests.

Tests might be run by executing the created binary and it does not need any arguments to set. After executing the binary the tests would run and developer would see the results.

IV. POSSIBLE STRUCTURES OF HYBRID C++/QML APPLICATION FOR SAILFISH OS TARGETED AT AUTOMATED END-TO-END TESTING

A. Hybrid QML application

As a reference case for demonstration of the proposed testing method, we consider a common hybrid application that can contain QML components written in QML language and C++ language. The structure of the hybrid project conforms with the template project generated by the Sailfish OS IDE and does not include any modifications to the build system.

The core components of the project source code are shown in Fig. 1. They include build system configuration files, source code of the application written in C++ and QML languages, translation files and resources. During the build procedure the source code written in C++ is compiled to the native binary code and linked into the executable file. The translation files are also transformed into the binary form. Files created on previous steps and all other source components are packaged into the RPM file for the target device.

After the installation of the RPM file, the application executable is placed into the `/usr/bin/` directory. The name of the executable matches with the project name. Unmodified QML source code and binary translations files are placed into the subfolder of `/usr/share/` having the same name of the project. Resources are placed into the locations needed by the application launcher. These locations correspond to the Linux Filesystem Hierarchy standard [9].

QML components developed with the use of C++ language are simply put into the corresponding folder and specified in the build system configuration files. Then they are registered in the QML engine during the application startup sequence before loading the base QML file. Therefore they are available to all required parts of the application.

Such an application structure is common enough to cover most of the applications for Sailfish OS. This structure allows only testing of QML components that does not depend on the custom C++ components. This rest of this section explains how to make such application testable using the approaches described above.

B. Application with QML extension

In this subsection we describe how to extract C++ components into a QML extension and to ship it with the application. This structure allows to run tests using the `qmltestrunner` application.

The structure of the project that utilizes the QML extension is shown in Fig. 2. The main difference with the reference project structure is the transformation into the multi-project structure in terms of qt build system. The base project uses sub-directories template and simply includes two other projects into resulting build. It also contains the configuration files for the build system that is responsible for the creation of the RPM-archive from the compiled source code.

Most of the contents of the reference application is moved into the QML application subproject. It includes sources of the application written using QML language. These source files must not be changed in any way, there is no need for extraction of components that depend on components written using C++ language. The subproject also includes translations and other application resources. These directories are managed by the QMake build profile for a Sailfish OS application and does not require extra setup.

The main difference in structure of this subproject with the reference application lies in the structure of C++ source code. This application contains only code required to bootstrap sailfish application and specify path to the custom QML extension in QML Engine. It must not contain any code that modifies the system in some way that is required by custom C++ QML components.

The last subproject is the QML extension project. It is configured as a library project and contains all the code written using C++ language that the original project contain. Except the starting point should be transformed from the application setup code to the QML extension according to the rules stated in the previous section.

It should be noted that QMake build file in the QML extension subfolder must be modified, so the resulting library

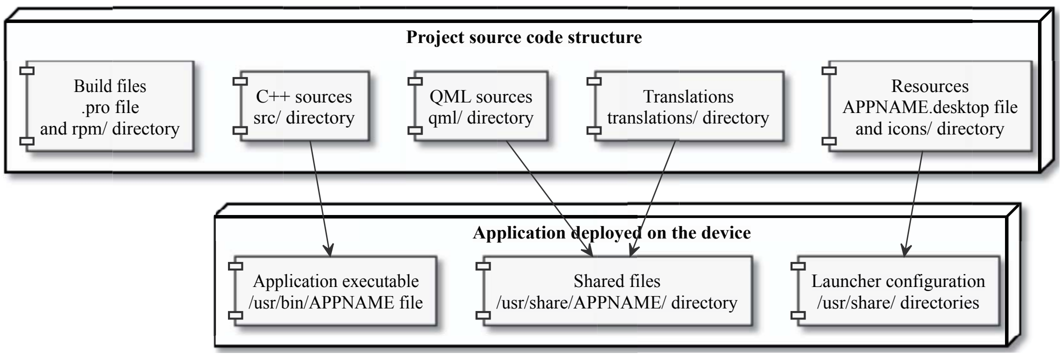

Project source code structure
Build files .pro file and rpm/ directory
C++ sources src/ directory
QML sources qml/ directory
Translations translations/ directory
Resources APPNAME.desktop file and icons/ directory
Application deployed on the device
Application executable /usr/bin/APPNAME file
Shared files /usr/share/APPNAME/ directory
Launcher configuration /usr/share/ directories

Fig. 1. Structure of the project and compiled application

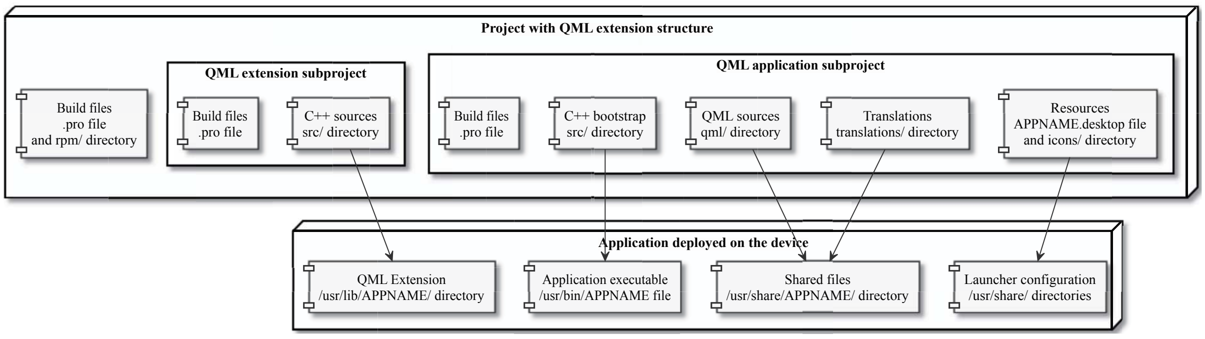

Project with QML extension structure
Build files .pro file and rpm/ directory
QML extension subproject
Build files .pro file
C++ sources src/ directory
QML application subproject
Build files .pro file
C++ bootstrap src/ directory
QML sources qml/ directory
Translations translations/ directory
Resources APPNAME.desktop file and icons/ directory
Application deployed on the device
QML Extension /usr/lib/APPNAME/ directory
Application executable /usr/bin/APPNAME file
Shared files /usr/share/APPNAME/ directory
Launcher configuration /usr/share/ directories

Fig. 2. Structure of the project using QML Extension

file would appear in the correct directory. One possible location of this library is the default location of all QML extensions for the Qt framework. Another one — the location of library files for current application. In the provided template we use the latter approach. Either way the specified file must also be added to the list of files that should be packaged in the resulting RPM file.

Summing up the first approach, the developer could easily split the original project into a multi-project build due to changes requiring only modification of couple C++ files and build configuration scripts.

C. Custom test runner

In this subsection we describe how to setup creation of custom test runner application that is able to use custom QML components developed in C++ language.

The structure of the project that is able to create custom executable to run tests is shown in Fig. 3. It is easily can be seen that the project is also transformed into a multi-component build in terms of qmake build system. The base project only includes other two projects into the build process. It also contains configuration of the packaging system.

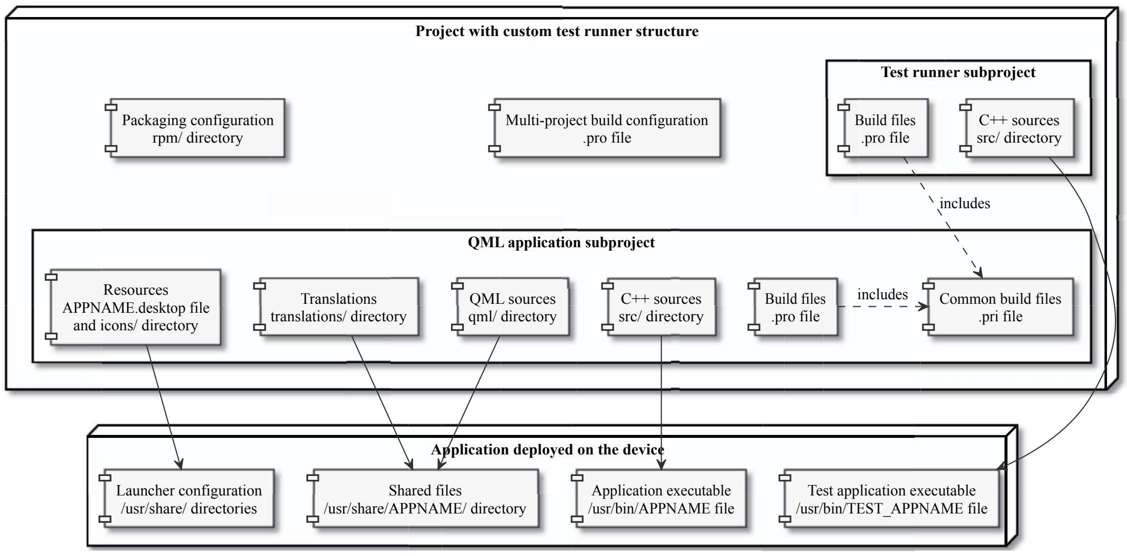
The contents of the reference project is moved into the QML application subproject. It contains all the source code of the application and mostly unmodified build files. The solely modification is the separation of list of C++ sources of custom components into a separate file. This way the list may be included in another subproject that eases the support of the project. Thought the proposed change is not mandatory for this scheme to work.

The test runner subproject is a base scenario only contains one C++ source code file and one build configuration file. The source code file simply includes header files of custom QML components and registers them in the QML Engine. Then it calls the special harness that is able to run tests and process application arguments.

If the core application during the initialization procedure not only registers the QML components, but also modifies the environment in some way, then these actions must also be executed during the startup of the custom test runner. This way the custom QML components will be executed in the correct environment. One way to ease the support of such code is the separation of it into a special functions that will be called during startup of both the original application and the test runner.

The build system configuration of the second subproject defines this application as a separate executable project and specifies that this executable should be placed in the correct location for the application. The subproject file must also include common configuration from the application project. Resulting executable must also be described in the packaging configuration.

Summing up the second approach, the second approach also requires developer to split application in a set of projects that correctly interact with each other.



Project with custom test runner structure
Packaging configuration rpm/ directory
Multi-project build configuration .pro file
Test runner subproject
Build files .pro file
C++ sources src/ directory
includes
QML application subproject
Resources APPNAME.desktop file and icons/ directory
Translations translations/ directory
QML sources qml/ directory
C++ sources src/ directory
Build files .pro file
includes
Common build files .pri file
Application deployed on the device
Launcher configuration /usr/share/ directories
Shared files /usr/share/APPNAME/ directory
Application executable /usr/bin/APPNAME file
Test application executable /usr/bin/TEST_APPNAME file

Fig. 3. Structure of the project using custom test runner

Both approaches allow developer to enable testing of custom QML components written in C++ language. Both of them viable for different application structures and different requirements for the application. If one structure seems less suitable for a developer at some point, one can switch from one project structure to another. In-depth analysis of concrete benefits and disadvantages of each method could be done based on the results of implementation in a various applications.

V. A TOOL TO RUN TESTS ON CONTINUOUS INTEGRATION SERVER

In previous section we discussed aspects of writing QML tests and providing custom QML components written using C++ language to the test environment. The next major aspect is consistent execution of those tests in a controlled environment, i.e. the continuous integration. Such an approach is a very important as it allows to constantly monitor the state of an application development and actively react to the detected issues [10].

Unfortunately, running of end-to-end tests for Sailfish OS created according to the method described above is not straightforward, and the main issue lies in the fact that Sailfish OS application SDK uses a separate virtual machine for compiling and execution of applications. It should be noted that it is not possible to execute QML test depending on Silica library outside the Sailfish OS.

Sailfish OS SDK provides an IDE that allows developer to build an application, install it on a device or an emulator and then execute it. These tools are tightly integrated into the Qt Creator and can not be called outside of it. Currently there is no tool outside the IDE that is able to execute required steps and the official guide to building and deploying application package contains a set of low-level commands that developer should manually execute on each machine.

In order to facilitate continuous integration we have decided to develop a special tool that would be able to automatically execute required steps described above, execute tests and provide results in a required format. The tool should be able run QML tests for the applications that conform with structures described in previous section in the environment of the integration server.

The flow of the tool operation is shown in Fig. 4. The tool executes the tests, provides results to the integration server, and shuts down. The scheme contains all major steps that tools goes through during one cycle of operation.

The tool is supposed to be started in the root directory of the project that should be tested. Other parameters include the location of Sailfish OS SDK, name of the application to launch and whether virtual machines should be shutdown after the execution of the tool. All these parameters can be either automatically determined or passed as parameters to the tool.

At the next step the tool tries to start the virtual machines required to create application packages and execute tests. These machines are managed by the VirtualBox hypervisor. The tool manages them through the standard management application and does not require additional setup. If the virtual machine already running, the tool tries to check connectivity via a SSH protocol and proceeds only when the virtual machine is fully running.

Then tool tries to compile the application and create a package using tooling in the MerSDK virtual machine. The tool uses the same tools to compile the project that the IDE does, so potentially it supports all projects that original IDE does. The created application package is then transferred to the Emulator virtual machine. The transferred package is installed and the package manager ensures that all the dependencies are met.

If it is required, the tool deletes all data in the user data location. It might be needed during the active development stage when internal data formats change frequently, but not needed when the application have already been provided to the end-users and migration procedures should be tested.

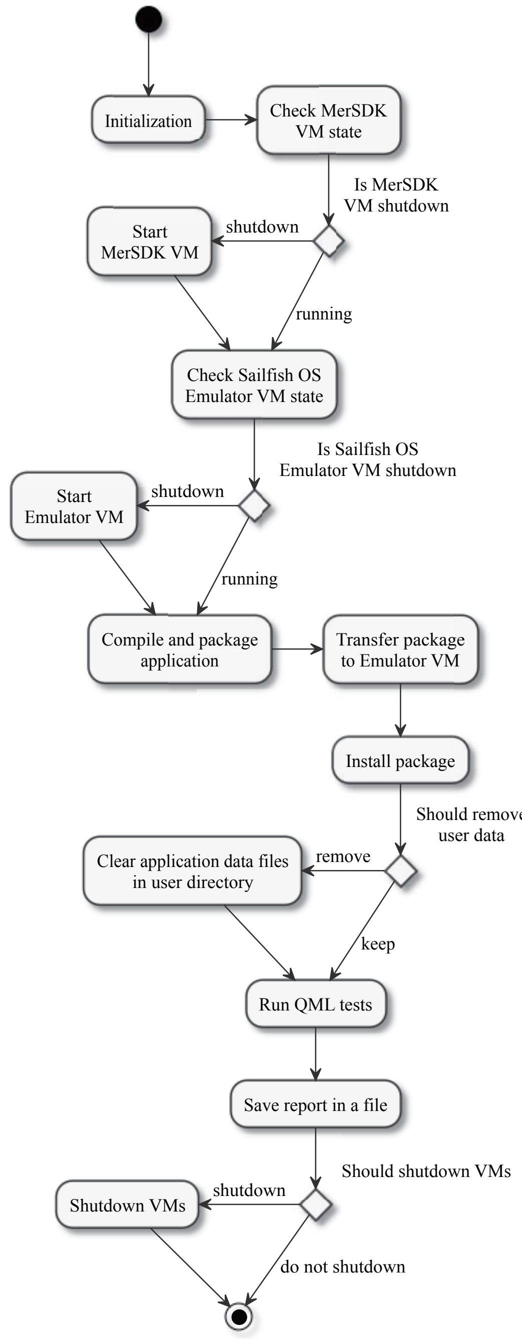


Fig. 4. Operation logic of a tool to enable continuous integration

Then the QML tests provided with the application are run. The tool assumes that files containing tests are provided along with the source QML code. Then the selected test running tool is started, so the results of the tests are written in the common format that integration server can process. Resulting file is then transferred from the emulator into the root directory of the project.

If needed, virtual machines started on the first step are shut down. It might be reasonable for the continuous integration server to have them running all the time if tests are frequently run. If tests are run infrequently there is no need to have virtual machines running all the time.

The developed tool is available at *https://bitbucket.org/yarfruct/sailfish-os-qml-test-runner* under BSD licence.

VI. Conclusion

In this paper we proposed a method for end-to-end testing of mobile applications targeted at Sailfish OS. It can be applied to a rather common class of applications that include standard and custom QML components, as well as C++ code. The main focus of the paper is on considering two approaches of code organization to make the application testable with the use of existing tools (QtTest library and qmltestrunner). Also we developed an open-source tool to facilitate the process of test running on the integration server to enable the possibility of adoption of the continuous integration technique for developer teams.

The future work may cover creation of the plugin for Qt creator that would allow to run tests straight out of the IDE. Such a plugin would improve the developers' performance by decreasing the overhead of the corresponding routine operations. Another important direction would be to extend the functionality of the developed tool for running tests on the continuous integration server to allow execution on real devices, not only on emulators.

References

[1] W. E. Lewis, *Software testing and continuous quality improvement*. CRC press, 2016.

[2] B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: trends and challenges," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM, 2014, pp. 1–9.

[3] J. Gao, X. Bai, W.-T. Tsai, and T. Uehara, "Mobile application testing: a tutorial," *Computer*, vol. 47, no. 2, pp. 46–55, 2014.

[4] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," *Journal of Systems and Software*, vol. 117, pp. 334–356, 2016.

[5] M. N. Mendiratta and R. Kumar, "Relative study of automated testing tools: Selenium, Quick Test Professional and Test Complete," *IMS Manthan (The Journal of Innovations)*, vol. 10, no. 2, 2016.

[6] H. Zadgaonkar, *Robotium Automated Testing for Android*. Packt Publishing Ltd, 2013.

[7] U. Gundecha, *Selenium Testing Tools Cookbook*. Packt Publishing Ltd, 2015.

[8] L. Johansson, "Writing and running qmltestrunner tests," Master's thesis, Häme University of Applied Sciences, 2015.

[9] D. Quinlan, P. Russell, and C. Yeoh, "Filesystem hierarchy standard version 3.0," The Linux Foundation, 2015. [Online]. Available: http://refspecs.linuxfoundation.org/fhs.shtml

[10] M. Meyer, "Continuous integration and its tools," *IEEE software*, vol. 31, no. 3, pp. 14–16, 2014.