Hindawi Publishing Corporation
Mobile Information Systems
Volume 2016, Article ID 2872067, 13 pages
http://dx.doi.org/10.1155/2016/2872067

Research Article

Automated Bidirectional Languages Localization Testing for Android Apps with Rich GUI

Aiman M. Ayyal Awwad and Wolfgang Slany

Institute of Software Technology, Graz University of Technology, 8010 Graz, Austria

Correspondence should be addressed to Wolfgang Slany; wolfgang.slany@tugraz.at

Received 24 February 2016; Accepted 11 April 2016

Academic Editor: Miltiadis D. Lytras

Mobile apps are everywhere. The release of apps on a worldwide scale requires them to be made available in many languages, including bidirectional languages. Developers and translators are usually different persons. While automatic testing by itself is important in general in order to be able to develop high quality software, such automatic tests become absolutely essential when developers that do not possess enough knowledge about right-to-left languages need to maintain code that is written for bidirectional languages. A few bidirectional localization tests of mobile applications exist. However, their functionality is limited since they only cover translations and adoption of locales. In this paper we present our approach for automating the bidirectional localization testing for Android applications with a complete consideration for BiDi-languages issues. The objective is to check for any localization defects in the product. The proposed methods are used to test issues of bidirectional apps in general and specifically for the Arabic language. The results show that the methods are able to effectively reveal deficiencies in the app's design, ensure that the localized app matches all expectations of local users, and guarantee that the product is culturally congruent to local conventions.

1. Introduction

The rapid proliferation of smartphones and the fast growth of the internet make it easy for "apps" (applications for mobile devices) to be accessed and downloaded from all over the world. According to the latest preliminary results from International Data Corporation (IDC), in the fourth quarter of 2015, smartphones sales topped in the worldwide with 399.5 million units and at the same time Android was the most popular operating system with market share reach of 82.8% [1, 2]. The greatest challenge is not only to develop apps but also to test them to guarantee their usability and robustness. Usually apps are distributed to many countries and language regions [1, 3]. Many apps on Google Play are available in more than 30 and up to 50 different languages from all over the world, including languages with exotic fonts such as for traditional and modern Chinese, Greek, Thai [4]. Every country or region has its own culture and customs; accordingly Google strongly encourages developers to localize their apps to meet demands of local users and thereby increase sales. However, usually developers have little knowledge about localization, and even if they would have the knowledge for all the languages they would like to support, high-frequency manual testing of all localized versions is practically infeasible, especially in a highly iterative development process where tests would have to be executed many times a day in order to avoid the accumulation and proliferation of bugs and other problems among members of the development team. Consequently, automatic app localization testing is getting more and more attention [5].

In general, software globalization (G11n) is a process which has two phases: internationalization (I18n), and localization (L10n) [6]. Internationalization refers to the process of designing and developing a system that support different languages and regions. Localization refers to the process of modifying internationalized software for use in a specific country, region, or culture, by adding local-specific features and translating text [6, 7].

The quality of international software is completely dependent on the software's localization level; therefore, localization

testing is an essential part of quality assurance of international software and thus has turned into a major type of international software testing [7, 8].

At present, most of the localization of software is only expressed in language translation and adaptation of time, date, number, and currency formats. Additionally, the localized language accuracy, integrity, interface layout, and document contents must satisfy the demands of local users and regional culture [9]. However, localization testing usually is either not done at all or infrequently done manually and only partially, usually neglecting languages not spoken by the testers; for example, localized versions of apps are only spot-tested manually for a few selected languages before a release.

Popular software often is produced in many language versions in order to be more understandable and usable for users. To an overwhelming percentage these applications are built for left-to-right (LTR) languages such as English or German. By contrast, Arabic language is written and read from right-to-left (RTL) [9]. The Arabic language additionally is cursive which means that the letters are connected together like English handwriting. The same character set has been utilized to express more than 25 languages in addition to Arabic [10].

In the area of software localization, the Arabic language is considered as one of the most challenging languages. The Arabic language "differs tremendously in terms of its characters, morphology, and diacritization from other languages, and to claim otherwise would be a mistake" [11]. It is ranked as the 5th language in the world in number of native speakers, and it is also one of the six official languages of the United Nations [10]. As a result of these, different issues in software localization to Arabic language should be considered.

Bidirectional (BiDi) languages such as Arabic, Farsi, Urdu, and Hebrew, where text is read and written from RTL while numbers are read and written from LTR, require layout customizations not only for text but also for all user interface (UI) widgets, including buttons, text views, edit texts, seek bars, check boxes, menus, and dialog boxes. There are many UI elements that need to be adjusted in both features and content localization. These include strings, layout, images and multimedia, character sets, and locale data [9].

Software localization testing is a critical method that is carried out to control the quality of a product's localization for a specific geographical region or culture. This test is like a passport for your app to transfer from country to country. The main goal of automatic software L10n testing in a test driven development process is to document the peculiarities of different localization requirements for those developers that do not know about them from their own cultural background and to make sure that bugs and deficiencies that are introduced at a later stage do not break the localization aspects of the product.

Indeed, the tests must have the ability to confirm the functionality and performance of localized software and components according to the original product and to detect linguistic and functional problems [12, 13].

It is essential that the correctness of translations is verified and that consideration of culture issues is guaranteed. However, the localization process often introduces severe issues such as the following [14, 15]:

(i) Clipped strings, or strings that overlap the edge of UI elements on the screen.

(ii) Date and time formats.

(iii) Untranslated strings (strings are displayed in the source language instead of their target language, possibly missing translation; this can happen quite frequently when the app is developed in a continuous way and not all translations can be added at the same time).

(iv) Inappropriate layout or text direction.

(v) Incorrect alphabetical sorting.

Additionally, the L10n tests are in some cases able to reveal deficiencies in the software's design [9, 12].

Many localization issues need to be reviewed as discussed by Kopsch [6]. Developers should consider the particular characteristics of each destination. As a result of the translation, there are also some cosmetic checks that need to be implemented. For example, one needs to check that the labels still fit on the screen, whether they were clipped, that they are not overlapped with other UI elements, and so forth as discussed by Cavalleri [7].

During the localization process, many good practices and tips should be considered, and the tester should follow the localization checklist and make it a priority to reveal I18n and L10n defects by concentrating first on five languages including English. Experience has shown that we are most likely to find specific issues in the following languages: German, Japanese, Arabic, and Hindi, as discussed by Kotze [12].

However, localization of mobile apps for BiDi-languages has still not reached its full potential due to a shortage of research. Most of the proposed approaches are designed to test localization issues for desktop and traditional web applications as discussed in [9, 13]. In addition, a few of the published papers identify the various unique challenges associated with localization testing of mobile applications as discussed in [5, 16] and identify the components to be included in an effective test strategy to build localization testing as discussed in [5].

In this paper, an automated BiDi L10n testing approach for smartphone applications is introduced; we have applied the proposed methods to test critical issues in an image manipulation app. The main contribution of this paper is reengineering our Pocket Paint app to localize it with a full and complete considerations for BiDi-languages issues, in particular the Arabic language, and to check the app for any localization defects in the user interface.

2. Bidirectional Localization Testing in the Mobile World

A tendency toward mobile applications development has increased in the past few years. Mobile phones and apps play an integral part of our life, and applications must work

correctly anytime and anywhere. On the developer side, offering localized versions of one's software simply increases the number of potential customers, and there thus is a strong economic motivation to offer one's software in as many major languages of potential markets as possible. However, some kind of internationalization and localization testing, be it manually or automatically, is necessary before distribution to the market makes sense.

The localization testing of apps faces many issues because of the complexity of testing these apps and the limited resources of mobile devices. Localization testing for mobile applications is more complex and challenging as compared to localization testing of traditional web-based and desktop applications [17].

Mobile applications work in multiple operating systems such as Android and iOS. Especially in the case of Android, the many different Android versions supporting totally different GUI elements, the additional GUI modifications by hardware manufacturers such as Samsung or HTC, the huge number of different keyboard apps that users can freely choose from, and the hugely varying display resolutions and aspect ratios of devices, make the development of the GUI of apps for all these combinations a difficult challenge.

Figure 1 illustrates the process of designing an effective localization testing approach for BiDi-languages. The key elements that should be considered while following this approach are as follows:

(i) The localization testing needs to be done on different types of devices on different platforms.

(ii) In addition to the original language, typically US English, the mobile application needs to support BiDi-languages.

(iii) The automation of localization testing can play an important factor in reducing the time and cost of testing the application in BiDi-languages.

3. Localization Test Automation

Manual testing for multilingual mobile apps is time and resource consuming, while automatic testing can save time and effort as well as increase accuracy and repeatability for localization testing. In general, automated testing can be considered the most desirable type of testing. The automated test can also be written to control the progress of system development and to find defects early and efficiently [5, 18].

It has been claimed that app features that are not tested are in fact not existing, the rationale being that refactoring, which by definition is the improvement of internal software quality without adding new features and a practice that tremendously helps to develop robust software, can only be done under the presence of tests, as otherwise the maxim "never change a running system" would prevail, and refactoring without tests can lead to the unintended elimination of untested functionality or could result in even worse consequences.

On the other hand, when automatic tests can be run by pushing a button and the tests are all "green," that is, there is no unexpected behavior of the software, developers can be sure that the current version of the app conforms to or, in other words, is correct with respect to these test cases. Psychologically, there is the added benefit that the feeling of the developers and, in case they are somehow in the development loop, also the customers is good when they see that the tests were all successful. If all features are tested, the development productivity is increased, among other benefits, as a result of the reduction of time spent for debugging, a time that consequently can then be spent on more productive tasks [19].

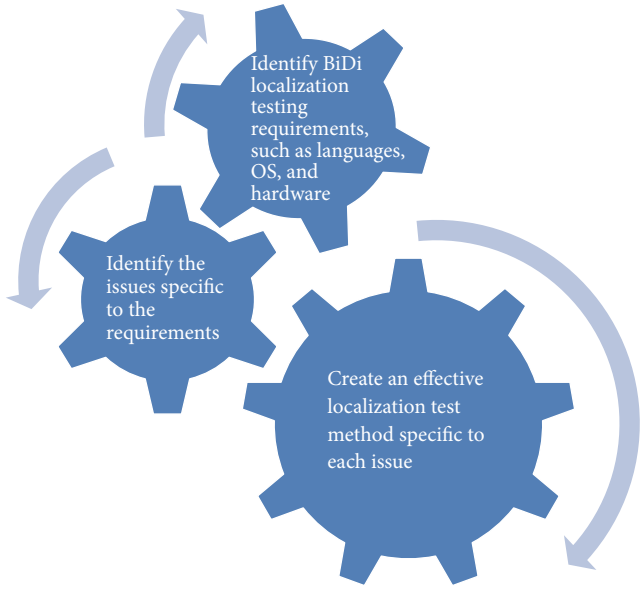


Figure 1: BiDi localization testing process.

Moreover, the automation for localization testing is very effective when test steps are consistently repeated in the same but also in a larger number of testing environments. With the different types of platform and operating systems available for smartphones, it would be impractical to manually test on all permutations [5, 19].

4. Why Test First?

In test driven development, the developer first writes an automated test which initially must fail before writing the corresponding production code. This test describes a desired improvement or new feature and thereby is both a formal specification that can be automatically checked for conformance and a human readable description of a test case. In a second step only does the developer write production code to pass the test, and then the test should run successfully [19].

In the development of our Pocket Paint app, we wrote an automated test that defines the text direction in BiDi-language. Since there was no code yet to make the test pass, this test should initially have failed. However, unexpectedly, the test case passed! How was this possible? The reason in this case was that the test case itself was incorrect as it checked the directionality of text incorrectly with the help of a locale configuration, as shown below.

```
final int expected=View.TEXT_DIRECTION_LOCALE;
assertEquals(mView.getTextDirection(),expected);
```

Hence, an immediate feedback was given to the programmer to use a different configuration for BiDi text direction, as shown below. Had the test been written *after* the production code already existed, the test would also have succeeded, but for the wrong reason!

```
final int expected=View.TEXT_DIRECTION_RTL;
assertEquals(mView.getTextDirection(),expected);
```

The automated tests are written to document software effectively for the developers, whereas the test cases describe the specifications that are defined by the customers, and also to avoid any ambiguity or miscommunication between users and developers. However, if the developer wants to write a test about any subject, he should first understand the subject under test.

One of the best ways for developers to understand the requirements is by translating them into tests. Several advantages for writing the test in advance of the code are as follows:

(i) If the tests are written in the last phase (when all coding has been done) it is highly probable that they will not be written.

(ii) Developers take more responsibility to produce high quality products.

(iii) If the tests are written early, they will be extremely helpful because more of the customer team's needs and expectations are clearly communicated to the developers, and thus there is less possibility for miscommunications.

5. Pocket Paint: A Brief Introduction

Pocket Paint is a paint editor and image manipulation Android app which allows setting parts of pictures to transparent and zooming up to pixel level. It is integrated into Pocket Code, but it can also be used on its own [20].

The Pocket Paint app has functions such as a stamp to copy-paste and resize parts of images, a pipette to pick an RGBa (red green blue alpha = transparency) colors, regular shapes (rectangle, circle/ellipse), or filling.

Pocket Paint is used for simple image editing such as rotating, flipping, zooming in/out, and cropping. The user can draw lines with different end shapes (circle or rectangle), and it also possible to thicken or thin the line by using the line width and style settings.

In the Pocket Paint color chooser, you can choose colors either from a palette of predefined colors or through an RGBa dialog.

Images are saved as PNG (with alpha channel) in the "Pocket Paint" folder and can be found via the Gallery and Photo apps. Pocket Paint is developed by the free and open source nonprofit Catrobat project [20].

6. Considerations for Localizing Bidirectional Apps

App sales should not be limited to just one market; mobile app stores are global, and so the app should be too. The app store is available in over 150 several countries, and the first step to reach this global market is to internationalize the app. A user who tries to download the app could be located anywhere. Today, customizing the app to be localized is a must.

Figure 2: Example of a localized BiDi version.

In practice, localization requires complex and technical jobs accomplished by a variety of specialists, including engineers, translators, graphics designers, testers, programmers, and project managers. The following are some best practices to consider for app BiDi localization.

Resources. A critical issue is that none of the UI element's text, colors, images, or styles should be hard-coded in the source code of software (i.e., separating them from code). All of them should be stored in resource files; the content of resource files can be retrieved by the software at run time to supply these elements to the users in their local language.

Externalize Resources. All noncode assets related to an application are considered as resources, such as content, images, and videos. The localization process requires developers to add appropriate resources to a mobile app to ensure that a given country, locale, language, or culture is supported. Therefore, all resources are placed into external files. After that, localization can become a simpler process when creating new versions of the resources files for each supported language.

User Interface Mirroring. The UI of a BiDi-language is generally mirrored and right aligned. Naturally, the common reading order for the speakers of BiDi-languages is from RTL. That is because those languages are written on a form known as RTL and strings flows in that direction as shown in Figure 2.

In general all UI elements should be mirrored, which includes lists, scroll bars, progress bars, pop-up boxes, grids, and galleries. The UI will be automatically mirrored when the user changes the system language to a RTL language. The direction of text is also changed to RTL with the exception of phone numbers, country codes, and similar numbers which are by default LTR also in RTL languages. Note that the direction of some views and widgets in the UI layout may not change automatically.

In addition, content like images, video, and maps are not mirrored. Nevertheless, some directional images such as

(a) (b)

Figure 3: Snapshot of a direction-sensitive arrow that changes meaning when mirrored. (a) UI in English. (b) RTL UI.

(a) (b)

Figure 4: Snapshots of (a) next and (b) back for BiDi-languages.

(a) (b)

Figure 5: Snapshots of (a) undo and (b) redo for RTL languages.

Figure 6: The ldrtl qualifier for drawable resources.

arrows need to be mirrored. The types of controls that should not be mirrored in BiDi-language are as follows:

(i) Video controls and timeline icons since they represent the direction of the tape.

(ii) Images (except if they indicate direction or order).

(iii) Clocks.

(iv) Graphs (x- and y-axes in RTL are the same as in LTR).

(v) Music notes.

Icons. Even though about 90% of the UI needs to be mirrored, not many of the icons in fact are affected. It is difficult to say which icons require mirroring and which do not because that depends on the icon's semantics. Therefore, cooperation between designers, developers, and language specialists (e.g., translators) should be considered. Graphics which are direction-sensitive introduce another challenge with regard to mirroring. These graphics can get an incorrect meaning when mirrored.

Within a LTR layout in a navigator, an arrow head that points to the left means that the navigation goes back to the previous page; an arrow head that points to the right means that it goes forward to the next page. When the layout is mirrored for a BiDi, the meaning will be just the opposite because a mirroring just reorders the UI elements of a layout without mirroring the actual icons as shown in Figure 3.

Therefore, special attention should be given to direction-sensitive graphics such as icons that have a specific directional orientation (back, next, undo, and redo), help balloons, or toast messages. Figures 4 and 5 show the back, next, undo, and redo icons for an RTL language.

The solution for handling direction-sensitive graphics is to have different directories of graphics in your resources to be used when the drawing destination is a RTL direction, in other words, never hard-code images or layouts. Moreover, a complete optimization for RTL layouts is provided by adding entirely separate layout files using the ldrtl resource qualifier

(a) (b)

Figure 7: Snapshot of a partial mirroring. (a) English version. (b) Arabic version.

(a) (b)

Figure 8: Snapshot of a complete mirroring. (a) English version. (b) Arabic version.

(ldrtl stands for layout-direction-right-to-left), as shown in Figure 6.

Further, for RTL languages the UI icons layout should naturally follow the RTL direction. A mirroring effect just replaces icons of a dialog box without mirroring the icons' images. This partial solution did not solve the layout problem in Pocket Paint's case for some icons such as the line and flip icons. It only gave them new positions within the mirrored dialog as shown in Figure 7.

In this case the feature itself is mirrored to address the problem of UI layout as shown in Figure 8.

Check Boxes. Check boxes and UI elements with check boxes are mirrored and right aligned, but the actual "check" symbols shall not be mirrored.

Localizing Strings. The user-facing text needs to be localized. Thus, text needs to be translated to a specific language before it is displayed to the user. More specifically, all the strings should be stored in values resource files; the text can be retrieved by the software at run time to supply these elements to the users in Arabic language. Hence, an alternative strings.xml must be created as shown in Figure 9.

In Pocket Paint's case the translation to Arabic is performed by volunteers using a community-based translation tool (Crowdin). The string file that must be translated is uploaded to a web environment in order to be accessible to our translators. Practically, in Android the UI elements are defined and stored in an XML file, and for every need-to-translate element a string item is added in the string.xml file in the values directory [21]. The Arabic translation folder

Figure 9: String file for Arabic language.

is created and named values-ar ("ar" represents the Arabic language according to ISO 639-1) [22].

Many tools for translating strings have been introduced; some are integrated with the development environment while others are platform-independent. Actually, translating strings automatically is not as succinct, complete, and correct as when a native speaker translates it. However, some words in the original language may have several meanings in the target language. For example, in some applications, there was a single-word text "kill," used to mean "stop the application." In Arabic, the literal translation of "kill" means "kill a person/kill an animal"; therefore skilled linguistic experts that also are domain experts for the app are needed for being able to correctly translate all strings. A translator must be qualified and have a base knowledge related to the project domain in order to translate this term as "stop" instead of "kill."

In real life, the miscommunication between developers and translators may produce many defects. In practice, the strings might be manipulated in the application without a translator's knowledge, and the translated string could then remain in an old state. Therefore, the developers must inform the translators about any changes in the string file and keep them up-to date.

Figure 10: An example for regional variants: English (UK) versus English (USA).

If developers need to distinguish between different regions that use, for example, the English language, they can add English regional variants for the United States, the United Kingdom, Australia, New Zealand, Canada, Hong Kong, and so forth to the app, and only the strings that are written in a different way would need to be included in the string file in order to be translated later; for example, "Colour" is favored in the UK, and "Color" is favored in USA as shown in Figure 10.

Default Resources. The default resources of app are those that are not marked with any language or locale qualifiers. If there is no default to fall back on, then the app will stop because Android looks for a resource and cannot find one that matches the configuration of the device.

Screen Size Variations and Limitations. One of the most important challenges presented by mobile phones is the limited screen size. However, if the app is designed to be used in different countries, both I18n and L10n test cases are especially essential under the constraint imposed by the different screen sizes. While developers may have designed their app to look fine in English, it may not look as well in German language or other languages where the character count will consume more space. On the other hand, some languages like Chinese contain a lot of information in each character and thus often need much less space, but this may not be true for other similar languages such as Japanese, since there characters include both Kanji that are of Chinese origin and purely phonetic Hiragana and Katakana characters, the latter two taking up comparatively more space. An additional problem with some of these Asian languages is that automatic line breaking can be very difficult as there are no space characters between words, and line breaks are not possible at all places, with additional complex rules to correctly break up lines.

Flexible Layout. Flexible layout is used to allow views relative to each other without fixed origins, widths, and heights. Any UI elements that contain text must be designed to be flexible. More space is allocated than necessary for English to accommodate most other languages (up to 30% more is normal). Practically, if fixed width constraints are used, localized text may appear truncated in some languages. Therefore, the constraints are removed and a "wrap content" attribute is used to allow limiting the width of each UI element.

Obviously, widgets and dialogs must be set to be expandable, both horizontally and vertically, in order to accommodate variations in texts width and height. The fonts in Arabic language can expand horizontally and vertically more than in English language. The result may be a truncated text because the space provided by the UI widget is not enough.

Figure 11: An example of a layout problem English > Arabic.

Arabic characters are more differentiated in structure and display than Latin characters. Sometimes translators may need two words or more in Arabic to describe one word in English. For example, in Pocket Paint there is a single-word string "Zoom." In Arabic, the literal translation of "Zoom" consists of two words and clearly does not fit on the text view that was reserved for the English "Zoom," but there is no shorter word with the same meaning in Arabic. The same is true for "Ellipse" icon as shown, as also shown in Figure 11.

Font Style for Mobile Applications. Font styles for text are discouraged in layout design such as special fonts, italic, or bold, because they may affect the readability of complex characters in some languages.

System-Provided Formatting Methods for Dates, Times, Number, and Currencies. The system formats are used to specify dates, times, numbers, currencies, and other values that can be changed by the so-called "locale," thus ensuring the correct formatting of data according to the locale. If there is a specific format that is based on assumptions about the locale of users, the problem will arise when the user changes to another locale. Typical problems are "../../...." date formats between the US and European date formats, where it is unclear whether 10/2/2016 is the 2nd of October (US) or the 10th of February (most European countries) 2016.

Importance of Textual Contents. Because of the small screen size of mobile phones, a lot of short phrases and words are suggested which lack content as well as words with multiple meaning, for example, "set." This can cause ambiguity and errors during the localization process. Therefore, to localize an app, the whole localization team should be very familiar with the application domain.

7. Continuous Integration

Continuous integration is one of the most vital practices in modern software development. The key idea behind continuous integration is to execute a predefined group of steps which are based on a specific trigger; the trigger could be a new pull request in the version control system, or trigger of time, for example, for automatically creating a nightly build and running all test cases on it, then creating a report that can be inspected via a web interface by all interested parties, and possibly reporting the main points of the test run to a developers' irc channel of the project. In short, continuous integration refers to the process of integrating all

development work at predefined times or events in order to be tested and built automatically. The idea is that this allows identifying the development errors early in the process [23].

We use Jenkins for continuous integration; it is an open source tool that allows controlling the execution of arbitrary automation steps. Jenkins has the ability to inform users about the success or failure of builds. It can be executed via the command line or can be run in a web application server. Jenkins is easy to install and has an intuitive and robust user interface [23].

Automated testing is an important part of any continuous integration environment; it allows team members to receive feedback about the state of the development in a dashboard view (not only one's own) and ensures through documentation of results that the tests succeed and are regularly executed, together with all kinds of other mechanisms and collections of statistics, for example, for automatic code quality evaluations or test coverage metrics.

8. Why Is Localization Testing for BiDi-Languages So Important?

Localization testing is a type of quality assurance testing; it mainly focuses on the evaluation of the product's functionality, cosmetics, and quality of the localization. The main goals of automatic app localization testing for BiDi-languages are as follows:

(i) To document the attributes of different localization issues for those developers who do not know about them from their own cultural background.

(ii) To make sure that bugs and deficiencies that are introduced at a later stage do not break the localization aspects of the product.

(iii) To detect and report app's localization defects.

9. Localization Testing Approach

After the product localization, localization testing is performed. The objective is to ensure that the localized product is fully functional, cosmetically correct, linguistically accurate, and culturally appropriate and that no issues have been produced during the localization process.

Figure 12 illustrates the architecture of the proposed automated localization testing approach. The proposed approach combines best-practice GUI testing tools for Android (Robotium and Espresso) and invokes them via a continuous integration server in order to execute all automatic GUI tests from a central place. This section provides substantial, complete development and user satisfaction testing methods for the localization aspects of the app.

9.1. Localized Resources Testing. This test verifies that all the required localized resources (files and folders) are present in the BiDi language version.

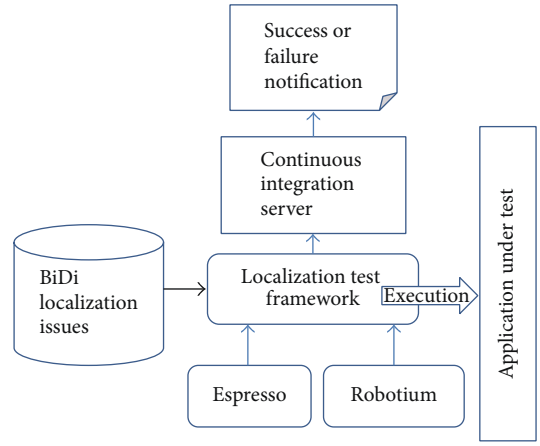


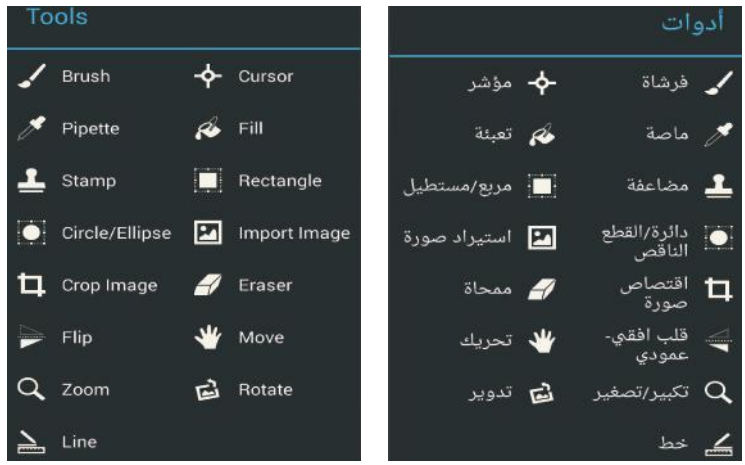
Figure 12: Localization testing architecture.

9.1.1. Fallback Language. After the app is tested in all BiDi-languages and locales, additionally the fallback language of the app is checked. To test this, the device language is changed to one that is not supported and it is checked that the app uses the implemented fallback language. This test suite makes sure that the app runs properly and reverts to default resources. In the snippet of code below, the language configuration is changed to modern Chinese, which is not supported in the mobile app, and in that case the fallback language is checked to ensure that the app reverts to US English. In addition, the method setLocale() is implemented to ensure the completeness of testing.

```
public void testFallBack() {
    setLocale(Locale.CHINA);
    String applicationStr=mSolo.getString(R.string.app_name);
    assertEquals(applicationStr,"Pocket Paint");
}
```

9.1.2. Localized Media and Default Resources. It is tested whether alternative graphics of locale-specific resources, which are provided for the RTL mode, are retrieved correctly by the app at run time when the locale of the device is changed to the RTL language. Also, the test case is written to make sure that a full set of default resources regardless of language or locale is included in the app's structure as shown below.

```
public void testDefaultResourceFileExistance() {
    boolean Exist = false;
    Context mContext = getActivity().getBaseContext();
    Resources res = getActivity().getResources();
    int checkExistenceForString = mContext.getResources().getIdentifier("app_name", "string",
                                                                         mContext.getPackageName());
    int checkExistenceForStyle = mContext.getResources().getIdentifier("CustomPaintroidDialog",
                                                                       "style", mContext.getPackageName());
    if (checkExistenceForString != 0 && checkExistenceForStyle!=0) {
     // The resouce exists...
        Exist = true;
    } else {
     // checkExistence == 0  // The resouce does NOT exist!!
        Exist = false;
    }
    assertTrue(Exist);
    String actual =res.getString(checkExistenceForString);
    String expected=mSolo.getString(R.string.app_name);
    assertEquals(actual,expected);
}
```



(a) (b)

Figure 13: Snapshots of (a) original UI and (b) Arabic UI.

9.1.3. Overtranslation. If the default texts are displayed instead of the translated texts in any part of the interface, then these texts may be overlooked or checked by reviewers. Such texts should not be translated and remain unmodified. In practice, the overtranslation testing is difficult to implement. However, the test should check that the text which is displayed in the text view is the same as the expected text defined in the strings.xml file. Hence, it might be impractical to check all elements' text in the app.

Actually, in some situations an untranslated string is not to be considered as a software defect. For example, the untranslated text might belong to a new feature added to the app that has not yet been translated, or to a UI element that should remain in the source language and is the same for all languages, such as a hyperlink or trademarked product names. Overtranslation issues are the responsibility of reviewers.

9.2. Linguistic Testing. Linguistic verification, or verifying the translation of all text on the localized product, is another very important task. In the proposed method, skilled linguistic experts can verify the linguistic content using screenshots of all the product screens to check the semantic correctness of the translations.

9.3. Cosmetic Testing. Translation has an enormous effect on the cosmetic quality of an app. The target of cosmetic testing is to verify that the localization phase did not introduce any visual defect and ensures that the UI has a consistent appearance throughout all supported BiDi-language versions, and a product contains no defects such as truncated strings, overlapping widgets, misaligned widgets, introduced during the localization process.

An important issue that needs to be addressed is the UI which contains layout, pop-up boxes, lists, and widgets.

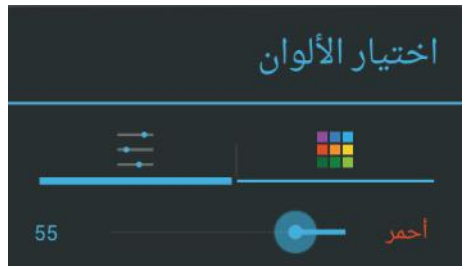


Figure 14: Snapshot of horizontal layout direction for RTL.

All UI elements need to support RTL direction as shown in Figure 13.

9.3.1. Views Swiping Direction. Most widgets such as seek bars, progress bars, spinner, and outline views appear flipped. Each seek bar view is tested. The testing method could be performed by sliding the cursor of the seek bar to a new position. The horizontal layout direction of these views should be from RTL; otherwise, the view needs to be adjusted. The assert method is used to compare the expected layout direction results from the test to the actual layout direction as shown in Figure 14.

The following code snippet illustrates the details of the test case and how the horizontal layout direction of a seek bar is tested. The test case detects in which direction the user moved his or her finger when touching seek bar, whether the direction is to the left or right. Obviously, the test case simulates user behaviors and interactions. The two variables downXValue and upXValue are defined to store the x values when the user's finger presses down and up, respectively. When the direction of the layout drawing is RTL, the test passes successfully.

```
public void testSeekBarDirection()
{
    mSolo.clickLongOnView(mRedSeekBar);
    mSolo.sleep(1000);
    mSolo.clickLongOnView(mRedSeekBar);
    mRedSeekBar.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {

        }
    });
    mRedSeekBar.setOnTouchListener(new View.OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            switch (event.getAction()) {
                case MotionEvent.ACTION_DOWN: {
            // store the X value when the user's finger was pressed down
                    downXValue = event.getX();
                    break;
                }
                case MotionEvent.ACTION_UP: {
            // Get the X value when the user released his/her finger
                    UpXValue = event.getX();
                    break;
                }
            }//end of Switch
            return true;
        }
    });
    mSolo.clickLongOnView(mRedSeekBar);
    mSolo.sleep(1000);
    mSolo.clickLongOnView(mRedSeekBar);
    mSolo.drag(mRedSeekBar.getLeft(),mRedSeekBar.getTop(),200,mRedSeekBar.getTop(),10);
    assertTrue(UpXValue>downXValue);
    String failMsg="The Direction of Red SeekBar is Left-to-Right";
    assertEquals(failMsg,mRedSeekBar.getLayoutDirection(),View.LAYOUT_DIRECTION_RTL);
    assertTrue(Integer.parseInt((String) mRedValueTextView.getText())>0);
}
```

9.3.2. Direction-Sensitive Graphics. Naturally, in the RTL interface the time moves from right to left, and thus any "back" type arrow has to point to the right and the forward arrow to the left. However, the localization process changes the directions of the undo and redo icons to be easily understood by Arabic speakers. Equally important, the undo and redo icons should correctly express the direction of time in all languages. A new test case is proposed to check the mirroring awareness in localized product for direction-sensitive graphics using some image processing techniques. However, the LTR image features after mirroring should match the BiDi features; otherwise, the test fails (see Figure 15).

A corresponding test case is implemented to check the mirroring awareness in the localized version. However, the undo feature in RTL language is the same as a mirrored one in the original version. So, the following code shows how the mirroring awareness is tested.

The two methods imageAreEquals() and doMirroring() are implemented to ensure the completeness of testing.

```
public void testUndoMirroring(){
    ImageButton undoRTL = (ImageButton) mSolo.getView(R.id.btn_top_undo);
    Bitmap RTLUndo = ((BitmapDrawable)
            undoRTL.getDrawable()).getBitmap();
    mSolo.clickOnView(mButtonTopRedo);
    setLocale(Locale.ENGLISH);//Change the GUI to LTR
    ImageButton undoOriginal = (ImageButton)
            mSolo.getView(R.id.btn_top_undo);
    Bitmap originalUndoBmp = ((BitmapDrawable)
            undoOriginal.getDrawable()).getBitmap();
    Bitmap  mirroredOriginal=doMirroring(originalUndoBmp);
    assertTrue(imagesAreEquals(RTLUndo, mirroredOriginal));
}
```

9.3.3. Auto Layout in Views. The proposed method needs to check that all fixed origins, widths, and heights are eliminated in the app's views so that the localized text can reflow automatically when the language or locale is changed as shown in Figure 16 (compared to Figure 11). The test code below asserts that the "wrap content" property is used, which means that the view can grow to become big enough to fit its own internal content.

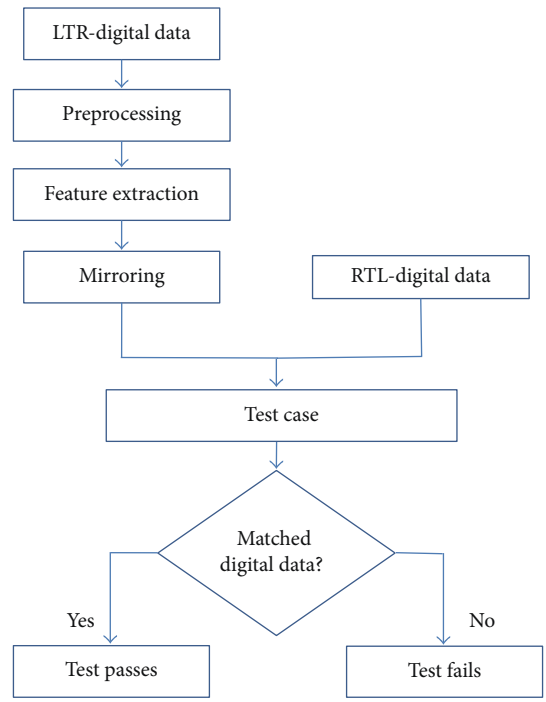


Figure 15: Flow chart for mirroring test case.

```
public void testViewWRAPCONTENT() {
    TextView mTextView = (TextView) mSolo.getView(R.id.tool_button_text);
    final ViewGroup.LayoutParams layoutParams = mTextView.getLayoutParams();
    assertNotNull(layoutParams);
    assertEquals(layoutParams.width, WindowManager.LayoutParams.WRAP_CONTENT);
    assertEquals(layoutParams.height, WindowManager.LayoutParams.WRAP_CONTENT);
}
```

9.3.4. Overlapping. After the translation is done, the localized tool can check the overlapping of widgets with other UI widgets. Mainly, the noOverlaps() method is used to ensure that the app you publish is devoid of overlapping in its UI. A bug report is generated if widgets are overlapping. For instance, in Figure 14, the RedView on the right side, the RedSeekBar in the center, and the RedValue on the left can be tested in this way. The code snippet below shows how the overlapping is tested.

```
@Test
public void assertNoOverlappingForToolsDialog()
{
    onView(withId(R.id.btn_bottom_tools))
            .perform(click());
    onView(withId(R.id.gridview_tools_menu)).check(noOverlaps());
}
```

9.3.5. Elements' Positions. The localization process changes the UI layout due to the translation, and the translated text may affect the size of the UI elements. Therefore, some views' width and height usually increase from their original size. This size stretching may cause missing views. For this reason, the localized process should ensure that all the layout views are located on the screen as the UI design dictates. For this

(a) (b)

Figure 16: Snapshot of auto layout in Arabic version views.

issue, the testOnScreen method is used, and an origin view is specified to start looking for the requested views.

In addition, the isLeftOf and isRightOf methods are used to check the positions of UI elements according to the locale direction and expose defects in the interface after mirroring. In practices, to localize the Brush tool button Figure 17(a) to Arabic, the icon should be on the right side of text as shown in Figure 17(c), not to the left as incorrectly shown in Figure 17(b).

The code snippet below illustrates the using of the isRightOf method to verify whether the icon is placed in the right of the text or not.

```
@Test
public void assertToolsRightOfText()
{
    onView(withId(R.id.btn_bottom_tools))
            .perform(click());
    onView(allOf(withId(R.id.tool_button_image), hasSibling(
                                                withText(R.string.button_brush))))
    .check(isRightOf(allOf(withId(R.id.tool_button_text), hasSibling(
                                                withText(R.string.button_brush)))));
}
```

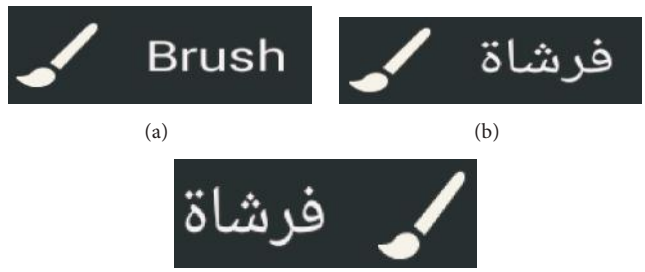
9.3.6. Misalignment. This test compares the alignment of views in the RTL dialog to the original one. It checks to ensure that if a set of views are aligned on a specific axis in the original dialog, they are all aligned on the same axis. To examine the misalignment, the following test methods are proposed.

The test case asserts that views are right aligned, that is, that their right edges are on the same x location. In order to verify that the views are aligned in the layout as we expect, the L10n tool should implement an assertRightAligned() method. For left alignment testing, the L10n tool needs to implement a test case to assert that views are left aligned.

It is also necessary for the localized tool to examine the correct positions for views before and after mirroring. Accordingly, the assertBottomAligned() method is introduced to assert that two views are bottom aligned, and their bottom edges are on the same y location before and after mirroring. In addition, a test case is introduced to verify that all views that are top aligned before mirroring are not adjusted after mirroring. Figure 18 illustrates the misalignment for the tools dialog.

If anything is found unsatisfactory, that UI element should be adjusted. Then these test cases should examine each activity, dialog, and other user interface layouts.

9.3.7. Text Direction. The direction of text becomes a critical issue when the app has Arabic texts, which are written and read from RTL. Hence, all user interface elements should support Arabic language; the text alignment and text reading order go from RTL.



(a) (b)

(c)

Figure 17: Snapshots for elements' positions in the tools dialog: (a) original version; (b) incorrect position; (c) correct position.

(a)

(b)

Figure 18: Snapshots for misalignment in the tools dialog: (a) English. (b) Arabic.

The text direction of each view such as text view, edit text, button, and pop-up menu is tested. The testing method could be performed by supplying each user view with a text string from Arabic language. The assert method is used to compare the expected text direction results from the test to the actual text direction after running with it throwing an exception if the condition is not met as shown in the code snippet below.

```
public void testRTLOrientation() {

    Locale locale = Locale.getDefault();
    assertTrue(isRTL(locale));
    assertEquals(getOrientation(locale), LayoutDirection.RTL);
}

public static boolean isRTL(Locale locale) {
    final int directionality = Character.getDirectionality(locale.getDisplayName().charAt(0));
    return directionality == Character.DIRECTIONALITY_RIGHT_TO_LEFT ||
                             directionality == Character.DIRECTIONALITY_RIGHT_TO_LEFT_ARABIC;
}

public static int getOrientation(Locale locale) {

    String language = locale.getLanguage();
    if ("ar".equals(language) || "he".equals(language)
            || "fa".equals(language) || "ur".equals(language)) {
        return LayoutDirection.RTL;
    } else {
        return LayoutDirection.LTR;
    }
}
```

9.3.8. String Truncation Validation. To test whether the UI is flexible enough to accommodate several language fonts and strings lengths, the following method is used to verify that a view is displayed correctly as it is defined in its XML layout file and that it fits into the UI without truncation. The test case checks the ellipsis count per line in a text view, not just for last line. If the ellipsis count is greater than zero, the method returns true. Figure 19 shows a tools dialog with truncated text because the space provided by the UI widget is not large enough.

```
public static boolean isTruncatedText( String text, TextView textView )
{   boolean Ellipsis_State=false; int ellipsisCount=0;
    if ( textView != null && text != null )
    {
        Layout layout = textView.getLayout();
        if ( layout != null )
        {
            int lines = layout.getLineCount();
            if ( lines > 0 )
            {
                for(int line=0;line<lines;line++)
                {
                    ellipsisCount = layout.getEllipsisCount( line );
                    if ( ellipsisCount > 0 ) {
                        Ellipsis_State=true;
                        break;
                    }
                }
            }
        }
    }
    return Ellipsis_State;
}
```

9.3.9. Keyboard Layout. A keyboard enables users to enter text. It is not a GUI aspect; rather, it is an operating system aspect. Some operating systems offer a keyboard framework that lets users create on-screen input methods such as virtual keyboards. However, the symbol input grid for RTL languages again must be mirrored and right aligned. And in an RTL UI, the cursor always has to be placed to the very right as a default. The testing is done by using keyboard keys to enter data into UI views, such as an edit text or text view. The input values are stored into data stores, and then they are displayed into the UI widgets or compared with actual text as shown below.

```
public void testEditText(){
    mSolo.clickOnEditText(mEditText);
    mSolo.enterText(mEditText, "1 برنامج");
    final String expectedStr ="1 برنامج";
    final String actualStr=mSolo.getEditText(mEditText).getText().toString();
    assertEquals(mEditText.getGravity(), Gravity.RIGHT | Gravity.TOP);
    assertEquals(mEditText.getTextDirection(),View.TEXT_DIRECTION_RTL);
    assertEquals(actualStr,expectedStr);
}
```

Figure 19: Snapshots for tools dialog with truncated text.

The following requirements must be considered for the testing methods:

(i) Arabic language package.

(ii) Font files supported by Arabic language.

(iii) Friendly encoding of strings (UTF-8).

9.4. Functional Testing. Localization functional testing focuses on whether the localization phase introduced any problems and verifies that all features work as expected after the localization. Hence, all the functional test cases are executed after the localization process and the snippet of code below is one of the functional test cases. This test case is used to check the color picker dialog and check the functionality of changing the colors.

```
public void testColorPickerDialogOnBackPressed() {
    assertTrue("Waiting for DrawingSurface", mSolo.waitForView(DrawingSurface.class, 1, TIMEOUT));
    mSolo.clickOnView(mMenuBottomParameter2);
    assertTrue("Waiting for DrawingSurface", mSolo.waitForText("Done", 1, TIMEOUT * 2));
    mSolo.goBack();
    assertTrue("Waiting for DrawingSurface", mSolo.waitForView(DrawingSurface.class, 1, TIMEOUT));
    int oldColor = mTopBar.getCurrentTool().getDrawPaint().getColor();
    mSolo.clickOnView(mMenuBottomParameter2);
    assertTrue("Waiting for DrawingSurface", mSolo.waitForText("Done", 1, TIMEOUT * 2));
    TypedArray presetColors = getActivity().getResources().obtainTypedArray(R.array.preset_colors);
    mSolo.clickOnButton(presetColors.length() / 2);
    mSolo.goBack();
    assertTrue("Waiting for DrawingSurface", mSolo.waitForView(DrawingSurface.class, 1, TIMEOUT));
    int newColor = mTopBar.getCurrentTool().getDrawPaint().getColor();
    assertFalse("After choosing new color, color should not be the same as before", oldColor == newColor);
}
```

10. Conclusion

In this paper, an automation approach for L10n testing is introduced to localize an Android app with a rich GUI for the Arabic language and culture. The Arabic version of the original product looks as if it had been developed in the Arabic user's home country. Traditional testing approaches can be

applied to localize any software from English to other Latin-based languages. On the other hand, localization of mobile apps for BiDi-languages are particularly challenging to test. BiDi-language software requires savvier testing methods and efforts to ensure efficient testing. Arabic language requires mirroring awareness to suit the RTL reading order and to provide a perfect right-to-left look and feel to the app. Localization to BiDi-languages affects not only the layout of text and UI elements but also iconography. The proposed methods make sure the UI elements and icons that communicate direction are mirrored correctly in RTL locales. Furthermore, the proposed approach fully and completely considers all BiDi-languages issues in order to effectively reveal all common localization issues in the app's design. The methods ensure that the localized software meets a local user's expectations in terms of language, features, and user experience in the traditional sense. In essence, the automation approach saves time and effort and increases accuracy and repeatability for localization testing and maintains the usability and portability of the app. Moreover, it leads to a significant decrease in effort and time spent for regression testing. Finally, the approach's principles can easily be applied to any smartphone platform.

Competing Interests

The authors declare that they have no competing interests.

References

[1] I. K. Villanes, E. A. B. Costa, and A. C. Dias-Neto, "Automated mobile testing as a service (AM-TaaS)," in *Proceedings of the IEEE World Congress on Services (SERVICES '15)*, pp. 79–86, New York, NY, USA, June-July 2015.

[2] IDC, *Smartphones*, IDC, 2015, http://www.idc.com/getdoc.jsp?containerId=prUS25988815.

[3] X. Xia, D. Lo, F. Zhu, X. Wang, and B. Zhou, "Software internationalization and localization: an industrial experience," in *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS '13)*, pp. 222–231, Singapore, July 2013.

[4] App Annie Support, Languages supported in Google Play reviews, 2015, https://support.appannie.com/hc/en-us/articles/204207814-Languages-supported-in-Google-Play-reviews.

[5] H. Dhingra and T. Roy, "Localization testing in mobile world," in *Proceedings of the Software Testing Conference*, 2013.

[6] C. Kopsch, "Localization testing one-year status report for a localization project," *Testing Experience*, no. 27, 2014.

[7] N. S. Cavalleri, "Localization testing is more than testing the translation," *Testing Experience*, no. 27, 2014.

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

[9] S. Abufardeh and K. Magel, "Software internationalization: testing methods for bidirectional software," in *Proceedings of the 5th International Joint Conference on INC, IMS and IDC (NCM '09)*, pp. 226–231, Seoul, Republic of Korea, August 2009.

[10] S. Abufardeh and K. Magel, "Software localization: the challenging aspects of Arabic to the localization process (arabization)," in *Proceedings of the IASTED International Conference on Software Engineering (SE '08)*, pp. 275–279, Innsbruck, Austria, February 2008.

[11] C. A. Sakhr, "Web-based Arabic-English MT engine," in *Proceedings of the Arabic NLP Workshop at ACL/EACL*, Toulouse, France, July 2001.

[12] N. Kotze, "Internationalization and localization testing," *Testing Experience*, no. 27, 2014.

[13] C. Zhao, Z. He, and W. Zeng, "Study on international software localization testing," in *Proceedings of the 2nd World Congress on Software Engineering (WCSE '10)*, pp. 257–260, Wuhan, China, December 2010.

[14] Microsoft, Globalization Step-by-Step, 2015, https://msdn.microsoft.com/en us/goglobal/bb688110.aspx.

[15] Net-Translators, *Localization for Mobile Apps*, 2015, https://www.net-translators.com/blog/localization-for-mobile-apps.

[16] Android By Code, Pseudo-localization testing in Android, 2015, https://androidbycode.wordpress.com/tag/rtl/.

[17] R. Selvam and V. Karthikeyani, "Mobile software testing—automated test case design strategies," *International Journal on Computer Science and Engineering*, vol. 3, no. 4, pp. 1450–1461, 2011.

[18] M. Geogy and A. Dharani, "Customising android automated testing framework to enable native hardware and software support," *International Journal of Engineering Research & Technology*, vol. 2, no. 2, pp. 1–4, 2013.

[19] K. Beck, *Extreme Programming Explained*, Addison-Wesley, Boston, Mass, USA, 2nd edition, 2004.

[20] Wolfgang Slany. 2010–2016. Catrobat, http://www.catrobat.org.

[21] R. Meier, *Professional Android*, John Wiley & Sons, New York, NY, USA, 4th edition, 2015.

[22] Android Developers, *Locale*, 2016, http://developer.android.com/reference/java/util/Locale.html.

[23] M. Soni, *Jenkins Essentials*, Packt Publishing, 2015.