UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Lais Pereira Felipe

Detecting Localization and Internationalization Failures in Android Apps: A Semi-Automated Approach and An Empirical Study of Developer Response in Open-Source Projects

Recife

2025

Lais Pereira Felipe

Detecting Localization and Internationalization Failures in Android Apps: A Semi-Automated Approach and An Empirical Study of Developer Response in Open-Source Projects

Trabalho apresentado ao Programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador (a): Breno Alexandro Ferreira de Miranda

Recife

2025

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Felipe, Lais Pereira.
Detecting Localization and Internationalization Failures in Android Apps: A Semi-Automated Approach and An Empirical Study of Developer Response in Open-Source Projects / Lais Pereira Felipe. - Recife, 2025.
75f.: il.

Orientação: Breno Alexandro Ferreira de Miranda.
Inclui referências.

1. localization; 2. internationalization; 3. software testing; 4. open-source. I. Miranda, Breno Alexandro Ferreira de. II. Título.

UFPE-Biblioteca Central

Lais Pereira Felipe

Detecting Localization and Internationalization Failures in Android Apps: A Semi-Automated Approach and An Empirical Study of Developer Response in Open-Source Projects

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagem de Programação.

Aprovado em: 29/07/2025.

BANCA EXAMINADORA

_____

Profa. Dra. Paola Rodrigues Godoy Accioly
Centro de Informática / UFPE

_____

Profa. Dra. Vânia de Oliveira Neves
Instituto de Computação / UFF

_____

Prof. Dr Breno Alexandro Ferreira de Miranda
Centro de Informática / UFPE
(orientador)

To my family.

ACKNOWLEDGEMENTS

I think I cannot start this section without saying how grateful I am for all the support and love I've received during this journey. Time flies really fast when you have a dissertation to write. But I have had many good people to help me finish it.

I would like to thank my family, my beloved mom Auda, who dedicated so much so I could be here today. My dear dad Antonio, who has never even had the opportunity to finish school, but never let me do anything so that I could focus on my studies. My dear brothers Vitor and Otávio who believe in me much more than I do myself or anyone else in this world.

I also would like to thank my advisor Breno Miranda. Thank you for all the trust and support. I have learned a lot as a professional, as a researcher, and as a human being.

Last but definitely not least, I would like to thank my friends who encouraged me not to give up. Raquel, there is no one like you. Davi, thank you for kindly helping me. Dany, Fernanda, Mauro and Glauciele, I would not even start without your words of wisdom. Ruan, you are such a kind soul, I'm really thankful for your help.

To all of you, my heartfelt THANK YOU.

"Software testing is and will continue to be a fundamental activity of software engineering: notwithstanding the revolutionary advances in the way it is built and employed (or perhaps exactly because of), the software will always need to be eventually tried and monitored." - *Antonia Bertolino* (BERTOLINO, 2007)

ABSTRACT

The process of globalization (g11n) is crucial for expanding the global reach of software. It involves two key components: localization (L10n) and internationalization (i18n). Localization and Internationalization testing are essential to ensure an ideal user experience, regardless of the software's language settings, as L10n/i18n errors are easily noticed by the end user and can cause discomfort or lead to misinterpretations of the software's use. This work proposes a semi-automated approach for identifying L10n/i18n failures in mobile applications on the Android platform. The proposed approach uses an open-source tool called Droidbot for exploring the interface of applications, executing them in different locales defined by the user. At the end of the exploration, reports are generated containing screenshots organized by locales. Subsequently, a human tester manually analyzes each screenshot in the report to identify and catalog the failures. The evaluation of the usefulness and efficiency of the proposed approach was conducted through an empirical study with 50 open-source Android applications available on the F-Droid platform, automatically explored in up to seven distinct locales, resulting in 237 rounds of automated exploration. The manual analysis enabled the identification of 41 failures related to localization and internationalization, which were reported to the developers of the evaluated open-source projects on F-Droid. Regarding the reported issues, 85.71% were responded by the developers, of which 80.49% failures were accepted, showing the usefulness of the proposed approach in identifying L10n/i18n failures relevant to open-source project developers. Furthermore, it was identified that some type of failures occur significantly more frequently than other types of L10n/i18n failures for the evaluated projects. In this study the most frequent failure found was *Missing Translation*. The exploration for all 50 open-source projects was executed automatically, taking 711 hours of execution for all the supported locales. In human workdays (considering 8 hours per day) it would be 89 days.

Keywords: localization; internationalization; software testing; open-source.

RESUMO

O processo de globalização (g11n) é crucial para expandir o alcance global do software. Ele envolve dois componentes principais: localização (L10n) e internacionalização (i18n). Os testes de Localização e Internacionalização são essenciais para garantir uma experiência de usuário ideal, independentemente das configurações de idioma do software, visto que erros de L10n/i18n são facilmente notados pelo usuário final e podem gerar desconforto ou levar a interpretações errôneas do uso do software. Este trabalho propõe uma abordagem semi-automática para a identificação de falhas de L10n/i18n em aplicativos mobile da plataforma Android. A abordagem proposta utiliza uma ferramenta de código aberto chamada Droidbot para a exploração da interface dos aplicativos, executando-os em diferentes locales definidos pelo usuário. Ao final da exploração, são gerados relatórios que contém capturas de tela organizadas por locales. Posteriormente, um testador humano analisa manualmente cada screenshot no relatório para identificação e catalogação das falhas. A avaliação da utilidade e eficiência da abordagem proposta foi realizada por meio de um estudo empírico com 50 aplicativos Android de código aberto disponíveis na plataforma F-Droid, automaticamente explorados em até sete locales distintos, o que resultou em 237 rodadas de exploração automatizada. A análise manual viabilizou a identificação de 41 falhas relacionadas à localização e internacionalização, as quais foram reportadas aos desenvolvedores dos projetos código aberto do F-Droid avaliados nesse estudo. Dos problemas reportados, 85.71% foram respondidos pelos desenvolvedores, dos quais 80.49% falhas foram aceitas, o que ratifica a utilidade da abordagem proposta em identificar falhas de L10n/i18n relevantes para os desenvolvedores de projetos de código aberto. Além disso, foi identificado que alguns tipos de falhas ocorrem com significativamente mais frequência do que outros tipos de falhas L10n/i18n nos projetos avaliados. Neste estudo, a falha mais frequente encontrada foi *Missing Translation*. A exploração de todos os 50 projetos *open-source* foi realizada de forma automatizada, levando 711 horas de execução. O que em dias de trabalho humano (considerando 8 horas por dia) seriam 89 dias.

Palavras-chave: globalização; localização; internacionalização; teste de software; código aberto.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

app	Application
ASCII	American Standard Code for Information Interchange
CLDR	The Unicode Common Locale Data Repository
CLI	Command Line Interface
ET	Exploratory Testing
FOOS	Free and Open Source Software
g11n	Globalization
GQM	Goal Question Metric
i18n	Internationalization
L10n	Localization
LTR	Left-to-right
OS	Operating System
RTL	Right-to-left
UI	User Interface
URL	Uniform Resource Locator

CONTENTS

1 INTRODUCTION

The target audience is an important part of the software context that must be specified during the development. When discussing a product intended for a worldwide market, the software should undergo the Globalization (g11n) process, which implies two important concepts: Internationalization (i18n), which refers to the software being developed to accommodate various languages and locales with distinct cultural conventions, and Localization (L10n) which refers to adapting the software to meet linguistic grammar structures or local/cultural aspects (YNION, 2020).

When discussing languages in the context of software, the textual contents that are presented are typically termed strings. Grammar standards for words and phrases, structures, date formats, and even units of measurement vary from language to language (TIJERINO, 2010).

For instance, the result of the division "1/2" in a calculator software should be shown as "0.5" in English, while "0,5" should appear if the software is configured for Portuguese. That is a basic illustration of the impact of localization on an software product.

Another important concept in g11n is locale. The same language is spoken in several nations, some of which are located on separate continents. Although the fact that communication is possible, this geographic difference carries with it certain subtle (and occasionally not so subtle) variances.

For example, if an Application (app) for clothing shopping configured for English from the United States (en-US), the word "pants" can be displayed, but if it is set for British English (en-GB), the value "trousers" should be displayed in the same string.

The term "locale" designates both the language and the geographical area[1]. The language is denoted by the first two letters (pt for Portuguese, for example), and the region from which the language originates is denoted by the remaining two letters (BR for Brazil, for example)[2] (MICROSOFT, 2023).

Localizing software is a challenging task since it requires more than just translation; it must also be consistent with local grammatical norms and cultural variances (SANTOS et al., 2019). These details make the difference from the user's perspective. Additionally, problems with `L10n/i18n` are immediately detected by the end user and might result in discomfort or lead to wrong understanding (COUTO; MIRANDA, 2023).

[1] https://learn.microsoft.com/en-us/globalization/locale/locale
[2] https://learn.microsoft.com/en-us/globalization/locale/standard-locale-names

In g11n software development the locale in which the software is developed is called source. When this software goes through the process of localization all the target locales (all the locales supported by the software) are localized based on this source.

The more screens, strings, and locales that the software supports, the more complicated this process becomes. If an application supports 41 locales, then all strings added or changed on the source must be localized for each of the 41 locales supported.

To make sure that the software content was correctly localized, it has to be verified. In order to verify it, the tester must first explore the app in a familiar locale in order to get familiarized with the software screens, functionalities, and which actions trigger specific strings (e.g., notifications, alerts, warnings, errors).

Then, the tester has to explore the application in all the target locales supported by the software to validate whether it meets grammar, date format, and cultural aspects for each specific locale. Besides that, another additional verification is evaluating whether the User Interface (UI) was not affected by the localized strings' length.

This task can be highly repetitive and time-intensive since the number of strings the tester will look for is directly related to the locales supported by the software. For instance, software that has 1000 strings and supports only 5 locales will have a total of 5000 strings to be verified during the L10n/i18n testing.

Scaling this example to software that supports more than 20 locales and has well over 5000 strings, the time and manual effort required make L10n/i18n testing highly resource-intensive in terms of manual work and project costs.

1.1 CONTRIBUTIONS

Exploring applications for the purpose of L10n/i18n testing is often repetitive and time-consuming. However, this step is both critical and necessary as part of the overall g11n process. Motivated by these challenges, the main contributions of this work are:

1. A Semi-Automated Approach for Detecting L10n/i18n Failures in Android Apps: This work proposes a semi-automated approach that combines automated exploration with manual human validation to support the detection of localization and internationalization failures. The app is automatically downloaded, and the locales are also switched according to user's input. After that an open-source tool explores Android apps across

multiple locales and captures screenshots. Then a report separated by locales is generated for further analysis.

2. Empirical Evaluation on 50 Open-Source Android Applications: A large-scale empirical study was conducted with 50 real-world open-source apps from the F-Droid platform. These apps were tested across up to seven locales (en-US, de-DE, es-ES, pt-BR, ja-JP, zh-CN e ar-EG), resulting in 237 exploration rounds of automatic exploration and the identification of 41 L10n/i18n failures. It is also identified that some failures occur significantly more often than others for the evaluated apps. In this study, *Missing Translation* was the most frequent L10n/i18n failure found. These results contribute to a better understanding of L10n/i18n failures in the evaluated apps. The study reports 85.71% response rate from developers for the 14 submitted issues, with 80.49% of the reported failures accepted. This confirms the relevance and practical impact of our findings and reinforces the usefulness of the our semi-automated approach.

1.2 DISSERTATION ORGANIZATION

This section describes the structure of the remaining chapters in this work:

- **Chapter 2: Background**

  Introduces key concepts related to Localization and Internationalization Testing. It discusses the processes of Globalization, Internationalization, and Localization, and provides clarification on both manual and automated testing. Additionally, it presents common types of L10n/i18n failures and an overview of test generation tools, establishing the technical foundation necessary for understanding the rest of this work.
- **Chapter 3: Related Work**

  Reviews existing studies and approaches in the areas of L10n/i18n testing, mobile testing automation, and exploration tools.
- **Chapter 4: A Semi-Automated Approach for Identifying Localization and Internationalization Failures**

  Describes the semi-automated approach developed in this study, including its implementation details. A walkthrough is also provided to familiarize the reader with its execution.
- **Chapter 5: Evaluation**

  Explains the experimental setup used to evaluate the proposed approach, detailing the

selection criteria for applications and locales, the metrics used, and the research questions addressed.

- **Chapter 6: Results**

  Presents the results obtained through the application of the approach, including answers to the research questions, analysis of identified failures, statistical findings, and developer feedback. Threats to validity are also discussed.

- **Chapter 7: Conclusion and Future Work**

  Summarizes the main findings and contributions of the study, and points out potential directions for future research.

2 BACKGROUND

In this chapter we describe the main concepts and definitions related to this work. We introduce the concepts of Globalization, Internationalization and Localization. After introducing these concepts we present a brief description about how a localized software is tested. Finally, the most common types of failures revealed by internationalization and localization testing are presented.

2.1 GLOBALIZATION, INTERNATIONALIZATION, AND LOCALIZATION IN SOFTWARE

Software Globalization is the process that comprehends both Internationalization and Localization. Software Globalization means providing a software product that gives the user the ability to use the product in different supported languages, in addition to the source (MCKETHAN; WHITE, 2005). In other words, it ensures that software can be adapted and works seamlessly for different markets with users from different cultures and languages.

G11n goes beyond translating software to different languages, it starts way before its textual content, in the design phase the software should already be planned in a way that accommodates different cultural and language differences, which is directly connected to other concept: Internationalization (MCKETHAN; WHITE, 2005).

Internationalization refers to the design and development of a software product in a way that facilitates its adaptation to various languages, regions, and cultural conventions without requiring significant engineering changes (AUER et al., 2010).

Localization refers to the process of adapting internationalized software ensuring that contents like language, date, time, and currency adhere to the language requirements and cultural differences to ensure that the software product can be used effectively and comfortably by users worldwide (ZHAO et al., 2010). Also, it's important to make sure that expressions/slang are still in use in the locale. Thus providing the users with a good experience while using the software.

2.2 LOCALIZATION AND INTERNATIONALIZATION PROCESS FOR ANDROID

Android is a a mobile Operating System (OS) used worldwide. Therefore `L10n` and `i18n` processes are fundamental during Android application development to ensure that apps are usable, and culturally appropriate for users across different languages and regions.

2.2.1 Internationalization for Android

During the design of the software, it has to be projected in a way that facilitates future adaptation. In Android it involves separating language and culture specific content from the source code to facilitate future adaptation. Best practices for internationalization in Android include

- **Strings**: Textual content visible to the final user should be put in correct nominated resource files `res/values-pt/strings.xml` for Portuguese for example. This way, it makes it easier during the localization process to translate and load the correct strings for the locale set. Having hard coded strings in the application's source code can lead to untranslated text (ANDROID, 2025a).
- **Resources**: Specific resource files such as graphics and icons (e.g., `res/drawable-pt/` contains graphics optimized for use with Portuguese), which the system automatically selects based on the device's locale set (ANDROID, 2025a).
- **RTL**: For locales Left-to-right (LTR) like en-US, the elements of the UI are aligned on the left side usually, but when accommodating Right-to-left (RTL) languages, it is important to enable the mirroring of layouts and UI components for better usability in languages such as Arabic, Hebrew, Persian, and Urdu (ANDROID, 2025b).
- **Format**: Date, time, number, and currency formats should also follow the locale's correct format (ANDROID, 2025c). To make sure that for different locales the correct standard format will be presented, developers can consult The Unicode Common Locale Data Repository (CLDR) (CLDR, 2025).

2.2.2 Localization for Android

The process of localizing software includes translating textual content, modifying icons, adjusting layouts, and adapting cultural aspects to ensure that it is following the locale's grammatical and cultural specifications. Main points of the localization process include:

- **Correct resources**: During the localization, it is important to make use of the adequate resource for the specific locale that it is being localized to make sure that the Android system automatically loads the appropriate resources based on the locale set[1].
- **Runtime changes**: Applications should manage locale changes at runtime to update resources without requiring a complete restart of the app or interfering with the user experience.[2]

2.3 MANUAL TESTING AND AUTOMATED TESTING

Software testing assesses a software's quality and determines if it satisfies the intended requirements (SANTOS et al., 2019). As a fundamental phase in the software development, testing helps ensure that applications perform reliably and as expected. According to Ardic et al., to develop software that we can trust, testing is essential (ARDIC; ZAIDMAN, 2023). Similarly, Beller et al. emphasize how crucial software testing is to ensuring a software product's quality (BELLER et al., 2019).

2.3.1 Manual Testing

According to Bertolino (2007), testing in general remains a crucial phase in the software development lifecycle, as it provides confidence in the system's behavior under real user conditions. Manual testing is one of the most traditional approaches to software quality assurance. It usually involves testers manually executing verifications often guided by predefined test plans or exploratory techniques (MYERS et al., 2011).

Manual testing remains essential in software quality assurance, as it allows testers to identify subtle issues that may not be detectable through automated methods. Human observation and

[1] <https://developer.android.com/guide/topics/resources/localization?>
[2] <https://developer.android.com/training/basics/supporting-devices/languages#FormatTextExplanationSolution>

interpretation can capture contextual nuances, usability problems, and visual inconsistencies that automation typically overlooks. According to Dukes et al. (2013), certain failures can only be uncovered through manual testing and the insights provided by the tester's direct interaction with the system.

There are many manual testing approaches discussed in the literature, but for this study we are focusing specifically on exploratory strategy to align with the goals of our proposed methodology. Itkonen et al. (2009) provides an overview of commonly adopted strategies and techniques for manual testing, including the exploratory strategy.

One example highlighted by Itkonen et al. (2009) involves systematically testing all features of the user interface using an experience-based approach. In this strategy, the tester navigates through the software's UI and examines each feature individually, aiming to uncover defects through interactive exploration. One possible technique applied for this strategy can be Exploratory Testing (ET) using session-based testing, where the test is conducted during a predetermined time-box (SOUZA et al., 2019).

However, this type of testing is inherently repetitive and time-consuming. To reduce the burden on testers and minimize the monotony associated with such tasks, it becomes essential to combine different testing strategies and adopt an effective automated testing approach (ITKONEN et al., 2009; BERNER et al., 2005).

2.3.2 Automated Testing

Software quality is more important than just being functionally correct. Jazayeri (2004) highlighted an absence of tools and technologies to create high-quality software and emphasizes the relevance of software quality that cannot be ignored.

Thus, testing software is an essential step in the software development process (SANTOS et al., 2019). This is primarily a repetitious activity that requires some level of investigation and product knowledge beyond the verification itself. As a repetitive task, testing can be really time-consuming and demand high effort. As such, automating this task would be highly suitable (RAMLER; HOSCHEK, 2017).

Test automation can help identify failures more quickly and at earlier stages of development. It can also reduce costs and significantly minimize the need for manual effort, but this is only accomplished when automation is applied with an appropriated strategy (BERNER et al., 2005).

While automation increases efficiency and reduces human workload during development,

it is not without its challenges. As noted by Garousi et al., one of the most challenging parts of software testing is automation (GAROUSI et al., 2017). Automation itself presents significant challenges, such as tool integration, maintenance overhead, and dynamic interfaces.

2.3.3 Manual Testing in the Context of Localization/Internationalization Testing

Alphabets, currencies, date and time formats, and other customs may vary depending on the location. In pt-BR, for instance, the temperature is typically presented in Celsius, but in en-US, it is typically presented in Fahrenheit. Weather forecasting software must be able to display the appropriate unit for each of these two locations if it supports them.

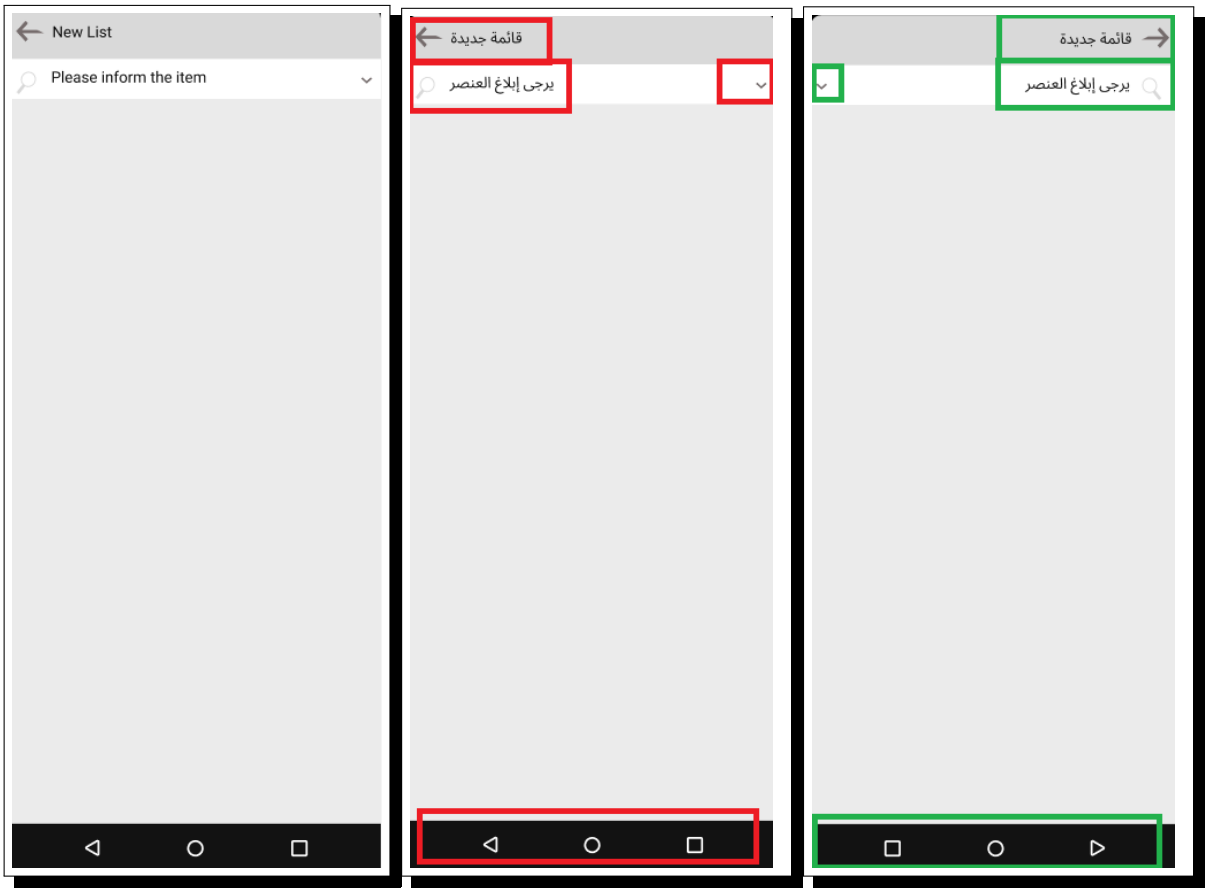
For software developed in en-US, the text presents American Standard Code for Information Interchange (ASCII) characters. In other hand, Non-ASCII characters languages present another challenge for the g11n process. Unknown or wrong characters can occasionally appear on screen for software that was created in an ASCII locale but needs to support a non-ASCII locale. It happens when the software was not designed to handle strings with different alphabets.

Furthermore, only strings are verified by L10n/i18n testing, UI elements can also be incorrectly localized. Most of the locales are usually read and written from left to right, like en-US, but some locales, such as ar-EG (Arabic from Egypt), are the opposite. This means they are read and written from right to left.

Therefore, it's essential to keep in mind that the UI elements are aligned on the right side of the screen while testing a RTL locale. Figure 1a displays the screen of an application on the source en-US, whereas 1b displays the identical screen for ar-EG with the strings translated. However, the elements are not properly aligned for the locale on the right side, as they should be. The correct alignment for ar-EG is shown in Figure 1c to ensure that the screen is localized for ar-EG conventions.

The examples mentioned above illustrate that it is necessary to execute on-screen verifications even when projects include files that contain all strings. For instance, misalignment and overlap related issues need on-screen analysis because context is essential to identifying these issues (RAMLER; HOSCHEK, 2017).

Thus, the tester should verify an application by checking every string on every screen at least once, usually when the string is introduced or its value is changed. Usually, the testers have to explore the app until they locate the string they need to verify because they are not



(a) Source in en-US (b) Not mirrored for ar-EG (c) Mirrored for ar-EG

Figure 1 – RTL example

usually told where to find it, most of the times only the value of the string is given.

Additionally, it might be more challenging and time-consuming because the tester may not be fluent in the language being tested. When this occurs, the tester typically sets the software to a locale they are familiar with (which is usually the source) to explore the app. Once they have located the string's screen, they set the locale that has to be verified.

The string "+ More options" in Figure 2, for instance, was recently changed for ko-KR (Korean from South Korea) and needs to be verified for this locale; however, the tester does not speak Korean, which may make it challenging to locate the string within the app. In order to verify the string, the tester will first set the locale to en-US because the source is en-US.

Once the string has been located, they will change it back to ko-KR. It shows that the number of screens and supported locales can increase the work and time spent on the investigation in order to perform L10n/i18n testing, as the tester had to navigate through five screens before reaching the proper screen.

Internationalizing a software product has become important in bringing in a large number of new clients and increasing income. L10n/i18n testing should be carried out at various

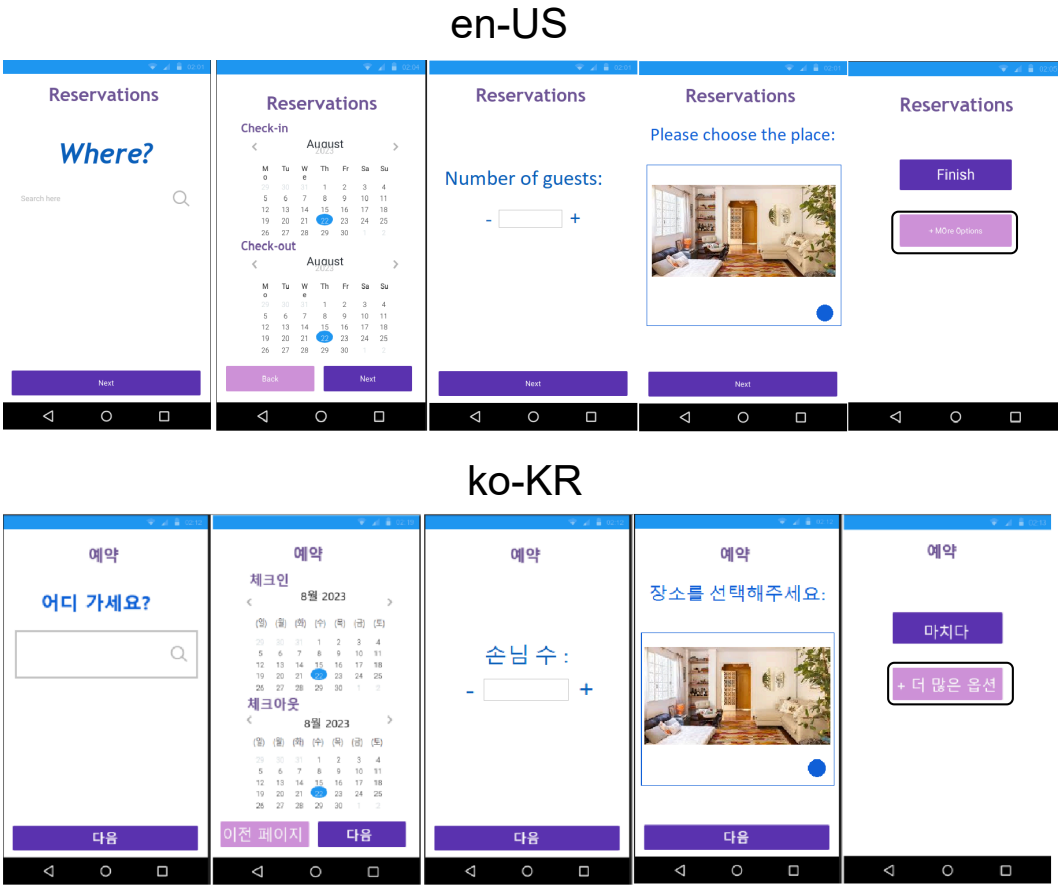


Figure 2 – L10n/i18n testing example. (adapted from (FELIPE et al., 2024))

points of the software life cycle to ensure that the software's use is seamless regardless of the language or its related norms (ARCHANA et al., 2013). Nevertheless, Ramler and Hoschek (RAMLER; HOSCHEK, 2017) state that there aren't many research that offer real results from relevant L10n/i18n testing.

2.3.4 Automated Testing in the Context of Localization/Internationalization Testing

As mentioned above, automated testing helps to reduce manual repetitive tasks and even reduces costs (SCHINDLER et al., 2021). To automate tests, it is crucial to have well-defined steps and a clear outcome to make sure the test implementation is effective. Another benefit of automating is that code can be reused for different types of tests (BERNER et al., 2005).

Localization and internationalization introduce a wide range of variables that can impact an

application's interface and behavior. These include text expansion based on locale, character encoding, date and number formatting, measurement units, and support for RTL layouts (MOLAN, 2008). While some of these behaviors are complex, many are predictable and can be reliably tested through well-crafted automated test cases with clear, step-by-step instructions and defined expected results.

Automated testing for `L10n` and `i18n` can be executed in early stages of development by checking if the code is being correctly localized. As discussed in Section 2.2, specific resource files are loaded for different locales set in the application. Also, it can be possible to identify in the code whether the software is being adapted or not for RTL locales or other locale-specific characteristics (AWWAD; SLANY, 2016).

Through the string's XML file, automated testing can also identify if there are strings that are not translated for the specific locale loaded. Furthermore, it is common to test the GUI, since it will present the user's point of view of the software. By testing the GUI, internationalization issues caused by code error can be detected as well, such as not being mirrored for RTL or incorrect date/time format (AWWAD; SLANY, 2016).

2.4 LOCALIZATION AND INTERNATIONALIZATION FAILURES

Beyond functionality, the `L10n`/`i18n` tester have to verify variations in the text shown, user interface elements on the screen, and discrepancies from the source and the locale's grammar requirements. Felipe et al. pinpoint a few common types of `L10n`/`i18n` failures: overlapping, not localized, truncation, ellipsis, inconsistency, and missing translation (FELIPE et al., 2024) .

- **Ellipsis**: when a string is too long to fit on the screen, it may be shortened and replaced with an ellipsis ("...") to indicate that the content continues, as shown in Figure 3a. While the presence of an ellipsis is not inherently a failure, the visible portion of the shortened string might unintentionally convey offensive or misleading messages, negatively impacting the user experience.

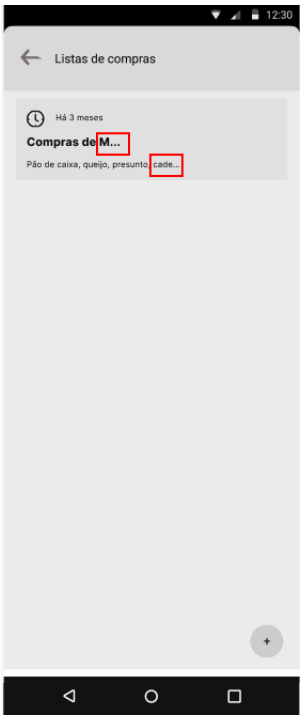
- **Truncation**: when a language uses larger characters or contains words longer than those in the original source, the displayed text may be cut off. In such cases, instead of showing an ellipsis, the word is abruptly shortened, which can negatively affect the user experience, as illustrated in Figure 3b.

- **Overlapping**: may occur when visual elements are not properly spaced or aligned, resulting in components overlapping one another. In some cases, changing the locale can cause the software to mishandle variations in string length, graphical resources, or even mirrored UI elements. An example of this is shown in Figure 3c.

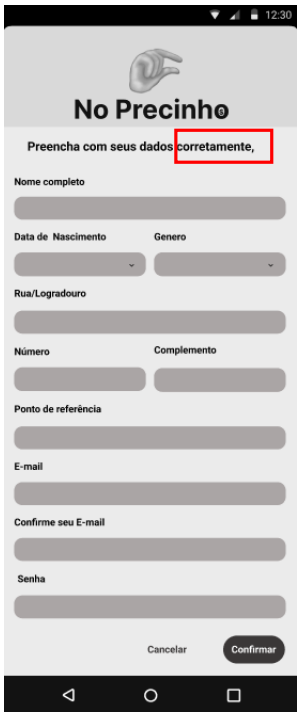
- **Not Localized**: refers to cases where the text has been translated, but the content is not appropriately adapted for the target culture or region. Common behaviors include RTL locales not being properly mirrored, incorrect formatting of dates, units, or currencies, and decimal separators that do not follow local conventions. For instance, as illustrated in Figure 3d, the locale pt-BR uses a comma as the decimal separator, as in "148,51," rather than a dot, as in "148.51."

- **Inconsistency**: can happen during the translation of strings. A single word can have multiple translations within the same language, and strings that share identical wording in the source language may end up with different translations in the target locale. As illustrated in Figure 3e, consider the case where the source uses the word "E-mail" in both "E-mail" and "Confirm your E-mail." When translated into pt-BR, one string retains "E-mail," while the other uses the alternative phrase "endereço eletrônico," leading to inconsistency.

- **Missing Translation** occurs when strings or portions of strings are not translated and still follow the source are displayed on the screen, defying the locale setting for whatever reason. Figure 3f is an example.

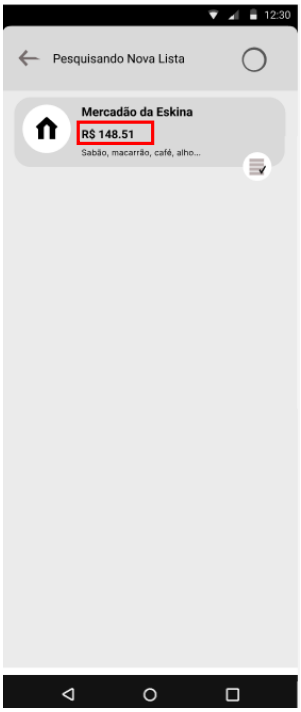


(a) Ellipsis

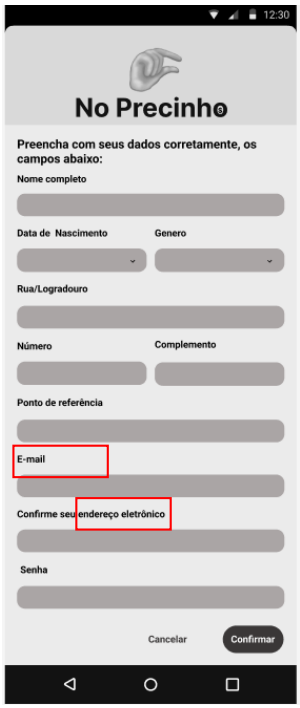


(b) Truncation

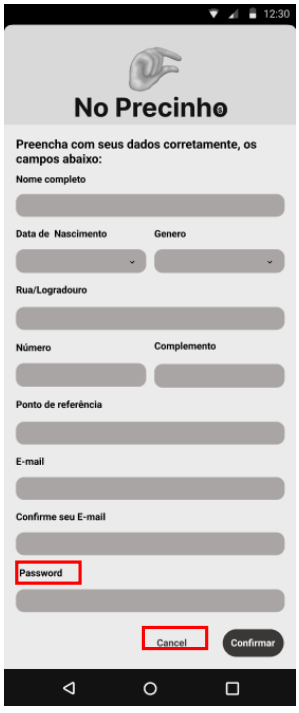
(c) Overlapping



(d) Not Localized



(e) Inconsistency



(f) Missing translation

Figure 3 – Common L10n/i18n failures

2.5 TEST GENERATION TOOLS FOR ANDROID TESTING

There are numerous test generation tools for Android available in the literature. In this section, we focus on five prominent tools: `Droidbot`, Humanoid, Monkey, DroidMate, and Sapienz.

2.5.1 DroidBot

`Droidbot` is an open source portable UI-guided test input generator that works with Android apps (LI et al., 2017). It combines static and dynamic analysis to guide the exploration process, generating targeted interactions based on the app's UI structure. By analyzing both the layout and runtime behavior of the application, `Droidbot` is able to systematically traverse the user interface. This tool allows the user to explore apps by incorporating their own scripts. Droidbot takes screenshots of the app's screen while exploring, as well as of its elements, including the strings on each screen.

2.5.2 Humanoid

Humanoid is a black-box test input generator that applies deep learning techniques to simulate realistic human interactions during Android app testing. Unlike other input generators that rely on random interactions with the UI, Humanoid stands out in its coverage by prioritizing important and meaningful interactions that resemble human behavior, leading to important states faster than randomly generated inputs (LI et al., 2020).

2.5.3 Monkey

Monkey is a tool for testing Android applications commonly used for stress testing. It works by sending a stream of random user events, like touches, gestures, and text inputs, to the app (A. Developers, 2012). It is a very simple tool that does not require configuration. However, because Monkey operates without any awareness of the application's state or UI structure, it is limited in its effectiveness in finding deeper or context-specific issues. While it remains a useful tool for preliminary robustness testing, Monkey is often complemented by more sophisticated techniques.

2.5.4 DroidMate

DroidMate is an automated testing framework for Android applications. It performs a state-aware exploration of the app's interface, recording detailed execution traces that can be used for both automated verification and manual analysis. DroidMate presents a modular architecture, allowing to plug in custom components to adapt the tool to specific testing objectives (JAMROZIK; ZELLER, 2016).

2.5.5 Sapienz

Sapienz is an automated exploratory testing tool for Android applications that leverages search-based optimization techniques. Sapienz is presented as a tool capable of maximizing the app's coverage while exposing many crashes and shortening the test sequences (MAO et al., 2016). The tool is based on the premise that software testing can be formulated as a multi-objective optimization problem, where different aspects of test effectiveness are balanced through optimization.

2.6 CONCLUDING REMARKS

In this chapter we presented the concepts of globalization, localization, and internationalization. The process and some challenges of `L10n` and `i18n` testing were briefly discussed, and some of the most common failures found during this type of testing were introduced and illustrated. Also, some test generation tools were introduced.

3 RELATED WORK

In this chapter, we review studies and approaches related to this research. We begin by presenting prior work focused on L10n and i18n, outlining the foundations and challenges of adapting software for multiple languages and regions. Next, we examine studies related to test generation techniques for Android applications, with an emphasis on automated input generation and GUI exploration. Finally, we discuss existing research specifically addressing L10n and i18n testing, highlighting approaches used to detect and analyze localization and internationalization issues in software.

3.1 LOCALIZATION AND INTERNATIONALIZATION TESTING

Santos et al. (SANTOS et al., 2019) highlights the critical role of internationalization testing in global software development, emphasizing the challenges developers face when adapting software for various locales. The paper shows the importance of L10n and i18n, and pointing that development should go beyond translation and adapt software to locales.

Couto et al. (COUTO et al., 2025) highlights the importance of proper training for testers. The paper introduces a tool that simulates common localization and internationalization failures, allowing trainees to practice identifying, documenting, and reporting these issues in real-world applications. By providing hands-on experience with common testing challenges, the tool helps testers gain proficiency and confidence in managing L10n/i18n tasks. The approach aims to not only equip testers with technical skills but also improve their ability to communicate effectively about localization issues, ensuring that software meets the linguistic, cultural, and functional needs of diverse global markets.

3.2 TEST GENERATION FOR ANDROID TESTING

Ynion (2020) present NEAR, designed to automate most of the tasks in testing UI Localization. By leveraging AI, the tool can intelligently interpret UI layouts, recognize different language scripts, and predict potential problems that might arise during localization. The results show that using the tool helped to reduce manual effort on Regression cycles. It shows the use of automated solutions to help the L10n/i18n testing process and reduce human effort,

which is also aimed in this study.

Felipe et al. (2024) propose a tool to reduce the need for manual search, saving valuable time for testers and allowing them to focus on resolving localization issues more effectively. The tool also use `Droidbot` outputs to find the step-by-step way to reach newly added or recent updated strings that need to be validated. This work shows the use of `Droidbot` to help on `L10n/i18n` testing, which is the same tool used on this study.

3.3 LOCALIZATION AND INTERNATIONALIZATION AUTOMATED TESTING

Ramler and Hoschek (2017) discuss the use of specialized frameworks that can simulate real-world conditions and language-specific issues, such as text expansion or character encoding problems, allowing testers to quickly identify localization issues. Authors strongly recommend automation since the tests performed are repetitive tasks therefore suitable for automation. Automation can accelerate the testing process and also reduce the risk of human error, ultimately enabling software to meet the linguistic and cultural requirements of a global market. This study reinforce that need of tools to reduce manual effort and automatized repetitive tasks.

Archana et al. (2013) presents a framework to automate the detection of localizability issues in internationalized software. The focus is to identify common problems like hard-coded text, layout constraints, and encoding errors that may make localization difficult. By automating this testing, the framework improves efficiency, reduces manual errors, and ensures software is ready for adaptation to different languages and regions. This work also reinforces the need for approaches to help and reduce human efforts.

3.4 CONCLUDING REMARKS

This chapter brought studies related to localization and internationalization, as well as techniques for generating tests for Android applications. The literature highlights the complexity of adapting software for global audiences and the importance of tools and automation to support this process. Reinforcing the motivation for this study. The studies presented in this section are a great contribution for `L10n/ i18n` testing. However, they do not encompasses the diversity of failures that our approach identifies and/or are more focused on white-box testing, while our approach focuses on black-box testing.

4 A SEMI-AUTOMATED APPROACH FOR IDENTIFYING LOCALIZATION AND INTERNATIONALIZATION FAILURES

Given the challenges discussed in Section 2, this study proposes a Semi-Automated Approach to assist the L10n and i18n verification process. The proposed approach uses an automated tool to explore Android applications and capture screenshots of them, and a manual analysis is performed using the images captured during the exploration to identify L10n/i18n failures.

4.1 THE PROPOSED APPROACH

To make sure that the software works properly and displays appropriate content to users from various geographical locations and language settings, the L10n/i18n tests are essential. However, testing Android applications in various locations can be very time-consuming and exhausting. Moreover, it requires the tester to have experience and familiarity with the tested locales. To address this issue, we propose a semi-automated approach that aim to help L10n/i18n testing by combining automated exploration with human review to reduce manual effort and optimize the time spent.

Among existing tools for automated Android app exploration, Droidbot was selected due to its usability, customizable exploration process, and ability to capture visual context. Unlike tools like Monkey, which generate random UI events without considering the current state (as discussed in Chapter 2, Section 2.5), Droidbot employs a UI-guided approach for more structured interactions. This is particularly useful in L10n/i18n testing, where capturing screens and strings helps verify results.

Compared to alternatives like Sapienz, Humanoid, and DroidMate, Droidbot stands out for its lightweight design and lower configuration complexity. While Sapienz and DroidMate offer advanced test generation and coverage, they require extensive setup. In contrast, Droidbot is straightforward to use and supports custom scripts via Command Line Interface (CLI), making it the ideal choice for this study.

Our semi-automated approach leverages automated app exploration to navigate the user interface of Android apps in different language and region settings. We employ Droidbot to navigate on Android open-source apps. During the automated exploration, the Droidbot interacts with Android applications, simulating real user behavior, capturing screenshots of

each state of the user interface, and also capturing strings during the process.

The semi-automated approach that is proposed here was mostly implemented in Python and was implemented as a CLI. An overview of the approach is shown in Figure 4, which highlights its key elements: automated exploration, data consolidation, human verification, and issue reporting.

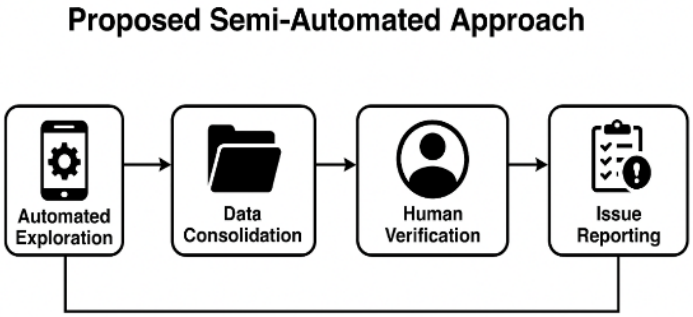


Figure 4 – Approach Overview

During the Automated Exploration, the script checks which app will be downloaded, then downloads the apk. After that, based on the user's input, the devices locale is changed by the script and `Droidbot` is called to explore the app through the interface, capturing textual elements and taking screenshots. The captured screenshots are grouped by locale during Data Consolidation.

After the structured report separated by locale, a Human Verification is conducted, where the tester will review the screenshots in order to identify `L10n/i18n` failures. Each failure identified during the Human Verification is reported back to the developers through the available issue tracker.

In the Issue Reporting, the captured screenshots are added, along with a step-by-step guide to reach the screen and reproduce the failure and the expected behavior for the locale.

To use the semi-automated method proposed in this work, the user needs to run a command line that includes five parameters. It is important to note that an Android device must be connected to the computer for the process to work. This device can be either a physical phone or a virtual emulator.

The first parameter should specify the full path to a text file (with a .txt extension), which must contain the Uniform Resource Locator (URL) of an application hosted on the open-source store. This URL serves as the reference from which the script will automatically retrieve the APK file intended for exploration. The script is capable of handling either a single APK or a

list of APKs provided within the text file, making it possible to run the exploration on multiple apps or a single app.

The second parameter defines the output directory where all artifacts resulting from the execution will be stored. This includes, but is not limited to, the downloaded APK file, logs generated during execution, and any additional data collected during the exploration process. By explicitly defining this directory, it helps keep results organized and easy to access later.

The third parameter corresponds to the unique identifier (ID) of the Android device on which the APK will be installed and the exploration carried out. This is necessary because there might be environments where multiple devices or emulators are simultaneously connected, as it ensures that the correct target device is selected for the execution.

The fourth parameter sets the locale or language settings under which the app will be tested. It can be a single locale or a list of locales separated by commas, allowing the app to be tested in different locales. Finally, the fifth parameter defines how long the app should be explored, in seconds. After this time, the exploration will automatically be interrupted.

The main output of the automated phase is a set of screenshots, organized by locales, which serves as input for the manual validation phase. A human tester, specialized in L10n/i18n, verifies the generated report with the screenshots to identify L10n/i18n failures as the ones shown in the examples in Chapter 2 Section 2.4. For observed failures, the tester will report the issue on the issue tracker of the app's project.

One of the significant advantages we aim for with the use of this approach is the substantial reduction in manual effort and time. Instead of requiring testers to manually explore each application for each locale, they can simply analyze the captured screens. This has the potential to not only accelerate the testing process but also allows for asynchronous and parallel analysis. Automated exploration can be executed without supervision, even overnight, with the possibility of allowing work hours to be optimized.

An additional advantage is that the results are kept as artifacts that can be shared or revisited again. When reporting issues in open-source projects, having artifacts such as screenshots can help to enhance the report since the relevant screenshots can be attached from the exploration artifacts.

4.2 WALKTHROUGH

To illustrate how the proposed semi-automated approach works in practice, this section presents the execution of the approach for testing L10n/i18n failures in an Android app. In this example of use, a tester aims to test an open-source Android app called Petals available on F-Droid. The execution will be performed for two locales: en-US and pt-BR.

- **Device:** An Android emulator with ID emulator-5554
- **App source:** A txt file named *apps.txt* containing the URL: <https://f-droid.org/en/packages/br.com.colman.petals/>
- **Locales:** en-US and pt-BR
- **Exploration time:** 3600 seconds per locale (1 hour).

The user initiates the process via the command line as shown in Listing **??**:

```
1 python run_experiment.py apps.txt ./output_dir emulator-5554 "en-US
    ↪ ,pt-BR" 3600
  \label{list1}
```

Listing 4.1 – CLI command to run the semi-automated approach

Automated Exploration

The script starts the execution with the parameters provided by the user. The first step is to download the APK from the link provided in "apps.txt". For each specified locale, the script performs the following steps:

1. Switches the device's locale to the provided value (e.g., en-US).
2. Calls `Droidbot` to perform an 1-hour exploration using the command shown in Listing 4.2.
3. Installs the APK on the connected device.
4. Explores the app randomly based on the UI elements, capturing screenshots and strings.
5. Saves all outputs to a structured directory.

```
droidbot -d emulator-5554 -a ./output_dir/apk.apk -o ./output_dir/
    ↪ apk_deviceName_locale -policy memory_guided -grant_perm -
    ↪ random -is_emulator -timeout 3600
```

Listing 4.2 – DroidBot CLI command used for exploration

Through the artifacts generated by `Droidbot`, it is also possible to know the path taken to reach the screens, offering a visual map of the exploration as seen in Figure 5, making it easier to reproduce the steps to reach the screens once it is identified `L10n/i18n` failures, without the need to redo the entire application exploration manually for each locale.

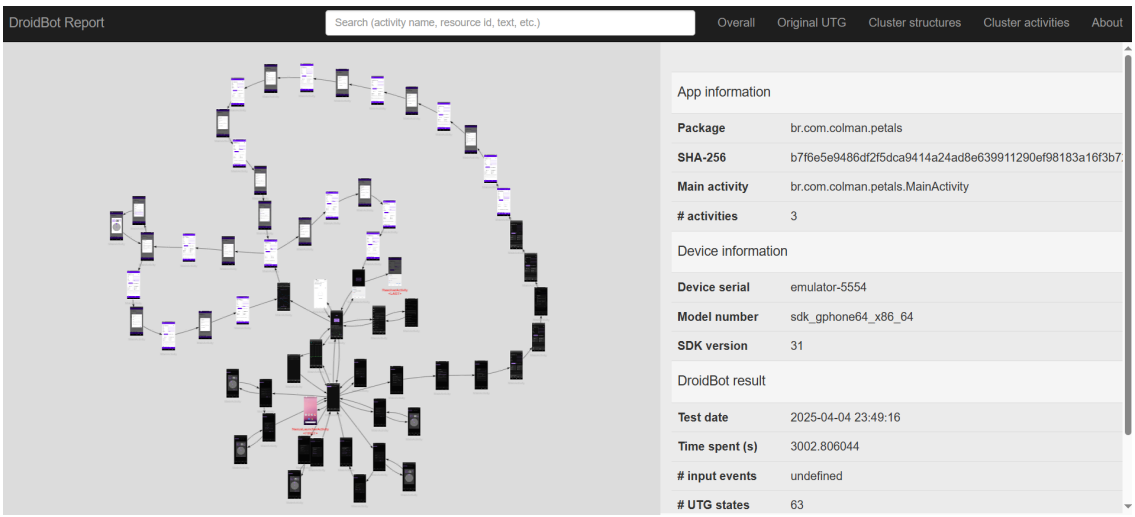


Figure 5 – Droidbot exploration

Data Consolidation

During the exploration carried out in the previous step, everything that the `Droidbot` captured is stored in folders that will be separated by the explored locales, making it easier to retrieve data by locale. After execution, the system saves: Captured screenshots and logs in structured folders and separate output directories per app and per locale as seen in Figure 6.

There is a report generated separating the results by locale. These outputs enable systematic comparison between different language configurations and provide navigation traces for easier reproduction of UI states.

Figure 6 – Data Consolidation

Human Verification

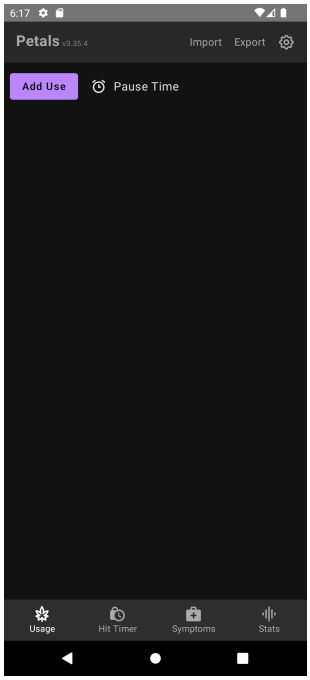
After the automated run completes, the tester verifies the captured screenshots. The human verification will go through each screenshot analyzing to see if there are any failures. Possibly this verification will be done by comparing with the screenshot source captured and stored on its respective folder, which in this case is en-US, as shown in Figure 7.

Issue Reporting

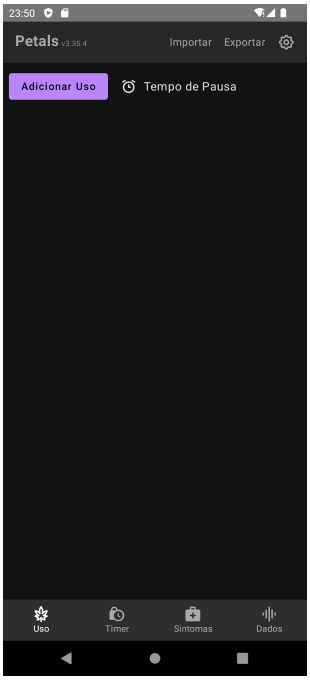
After analyzing the screenshots for each locale, if the tester identifies any failure like the one in Figure 8, where there are incorrect units format for pt-BR, the tester then:

- Selects relevant screenshots from the folder.
- Uses the log files to provide reproduction steps.
- Opens the app's issue tracker (e.g., GitHub).
- Submits a bug report, attaching the screenshots and steps.

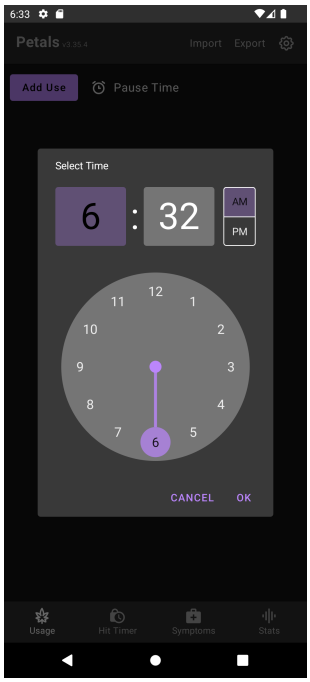
Table 1 summarizes the process using the approach:



en-US



pt-BR



en-US

pt-BR

Figure 7 – Human Verification

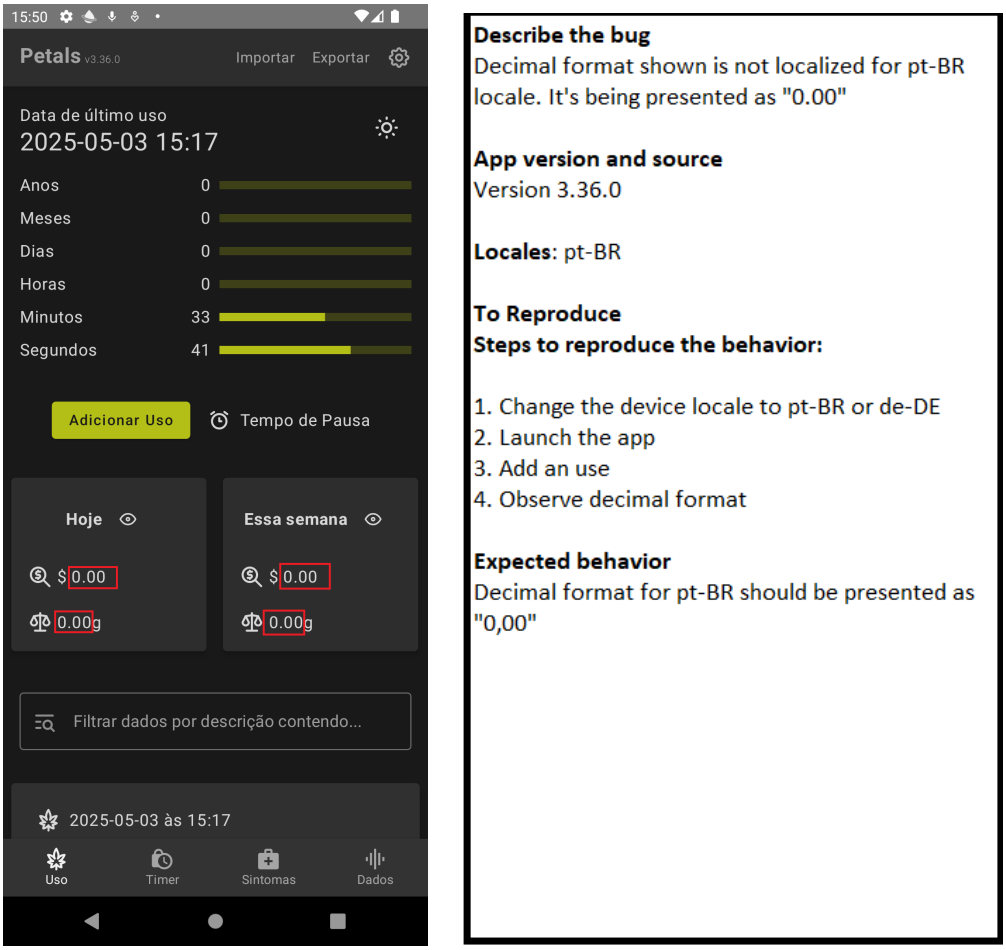


Figure 8 – Issue Reporting

Step	Description
Automated Exploration	Downloads the apk, then changes the device locale to the selected one. Using `Droidbot`, install the apk and explore the app's UI.
Data Consolidation	Compiles the exploration's screenshots into a structured report that is organized by location.
Manual Verification	A human tester analyzes UI screenshots to identify `L10n/i18n` failures.
Issue Reporting	Reports failures found using screenshots captured during Automated Exploration

Table 1 – Summary of the Approach's Use

4.3 CONCLUDING REMARKS

Navigate through applications when the tester doesn't know the language can be really difficult and time consuming to understand exactly what it is being presented on screen and even to reproduce the steps. Our approach, introduced in this chapter, strikes a balance be-

tween automation and human expertise. While it does not replace manual review, it automates repetitive tasks (locale switching, navigation, screenshot capture), allowing testers to focus on analysis and report.

5 EVALUATION

5.1 GOAL, QUESTION AND METRICS

Following the Goal Question Metric (GQM) paradigm presented by (BASILI et al., 1994) we have defined our Goal, Question and Metrics for this study.

Goal

The research goal of this experiment is to **Analyze** the proposed semi-automated approach
For the purpose of finding and reporting L10n/i18n failures
With respect to Android open-source applications
From the viewpoint of L10n/i18n testers
In the context of Android L10n/i18n testing.

Questions

For assessing our study's Goal we defined the following questions:

- **RQ1: What is the usefulness of the proposed semi-automated approach in L10n/i18n for finding and reporting L10n/i18n failures?**
- **RQ2: What types of L10n/i18n failures are the most common in the evaluated open-source apps?**
- **RQ3: How relevant are L10n/i18n issues for developers in the context of open-source apps?**
- **RQ4: How efficient is the proposed approach?**

Through RQ1 we intend to assess the usefulness of the proposed approach so we can understand the contributions of the approach to the L10n and i18n testing. With RQ2 we can understand better the most common failures and improve our approach to be able to identify the most common types of L10n /i18n failures and help on L10n and i18n testing. Through RQ3, we aim to understand how relevant L10n/i18n issues are for developers in the context

of open-source apps. With RQ4, we assess the efficiency of the proposed approach, helping us to understand its potential to support L10n and i18n testers in the verification process to identify and report L10n/i18n failures, reducing human effort.

Metrics

In order to answer RQ1, we need to understand how its use can help out with the verification process to find and report L10n/i18n failures. When the app's exploration stops, there is a report generated by the approach as pointed out in Chapter 4. The report contains all the screenshots captured during the exploration.

A tester manually verifies the set of screenshots in order to identify L10n/i18n failures. To evaluate it, we consider the detection rate of L10n/i18n failures. We can assess the usefulness of the approach through Equation 5.1, where we summarize the issues found. It illustrates how this approach can be used to find actual L10n/i18n failures.

$$M_1 = \text{Total Number of L10n/i18n Failures Identified} \tag{5.1}$$

To address RQ2, we aim to identify and categorize the most frequent types of failures that occur in open-source apps during the L10n/i18n verification. To do this, we analyze the reported failures found through our semi-automated approach, and we focus on the detection frequency of L10n/i18n failures by type of failure.

The metric to assess RQ2 is displayed in Equation 5.2 and it is computed by summarizing the frequency of each type of failure found by the manual verification, such as *Missing Translation*, Ellipsis, etc., across the evaluated apps. By applying this categorization across the evaluated apps, we can better understand the most common failures in L10n/i18n, guiding future improvements and prioritization of future improvements for our approach.

$$M_2 = C_i, \text{ where } C_i \text{ counts the occurrence of the failure type } i \text{ across the explored apps} \tag{5.2}$$

To assess RQ3, we consider a quantitative analysis. For the evaluation, we consider the acceptance ratio of the issues reported in the open-source projects of the explored apps. This metric is displayed in Equation 5.3 and it is computed by dividing the number of issues that were accepted and confirmed by the developers by the total number of issues that were

reported. In addition to that, we complement our analysis by considering the feedback and responses provided by the developers.

$$M_3 = \frac{\text{Number of confirmed issues}}{\text{Number of submitted issues}} \times 100\,(\%) \tag{5.3}$$

To answer RQ4, we focus on measuring the time required by the semi-automated approach as a practical indicator of its efficiency. Specifically, we assess how much human effort is saved by adopting this approach. This involves evaluating both the total time spent (all apps) using the semi-automated approach and the time taken per app.

The time dedicated to exploration by the approach represents a direct saving of human effort since the automated execution can be carried out 24 hours a day in contrast with a human's usual 8 hours a day. This metric is expressed by Equation 5.4.

$$M_4 = T_t = T_e + T_a \tag{5.4}$$

where total time T_t is the sum of execution time T_e and analysis time T_a.

5.2 PLANNING

The planning is essentially the same for all four questions; it involves the selection of applications and the execution of the experiment using our semi-automated approach, which will be described in Sections 5.3 and 5.4.

The planning will differ only for RQ2, which receives support from not only descriptive statistics but also hypothesis tests to address RQ2. For this reason, we describe in the following subsections our hypotheses, treatments, and dependent and independent variables.

5.2.1 Hypotheses

As stated on Section 5.1, to answer *RQ2*, we define our Null Hypothesis that states that all types of `L10n/i18n` failures have no significative difference, which means all failures presents statistically the same frequency. On the other hand, the Alternative Hypothesis states that some types of `L10n/i18n` failures will present statistically a higher frequency than others.

Hypothesis 1 (Null Hypothesis)**.** *All types of* `L10n/i18n` *failures occur with equal frequency in open-source apps.*

Hypothesis 2 (Alternative Hypothesis)**.** *Some types of* `L10n/i18n` *failures occur significantly more frequently than others.*

Formally:

$$H_0 : f_i = f_j = \cdots = f_k, \quad \text{where } f \text{ is the observed failure frequency.} \tag{5.5}$$

$$H_1 : \exists f_i \neq f_j, \text{ for some } i \neq j \tag{5.6}$$

5.2.2 Treatment and Measurement

There is no experimental treatment in this study; instead, this is an observational analysis based on the classification of real-world failures found using a semi-automated verification approach across open-source apps.

- The independent variable is the type of failure.
- The dependent variable is the observed frequency of each failure type.

5.2.3 Statistical Test

Since there is only one categorical variable, and we aim to compare the observed counts (types of `L10n/i18n` failures) to the expected counts (frequency), with each observation being independent and mutually exclusive, the Chi-square goodness-of-fit test was chosen. Since this test does not assume normality and can be used for observed frequencies of mutually exclusive categorical variables, it is suitable for this evaluation (PLACKETT, 1983; COCHRAN, 1952).

Assumptions:

- **Each failure is counted once and classified into one the categories.**
- **Expected frequencies are equal under H_0.**

To calculate the Chi-Square χ^2, the statistical formula is (PANDIS, 2016):

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i} \tag{5.7}$$

(5.8)

Where:

O_i= Observed frequency

E_i= Expected frequency

The formula below can be used to find the expected frequency count, which is equal to the expected frequency counts at each level of the categorical variables (TURHAN, 2020):

$$E_i = \frac{n}{p_i} \tag{5.9}$$

Where:

n= Total of sample size

p_i= Hypothesized rate of the observations at i level

Obtaining degrees of freedom, expected frequency counts, test statistics, and the P value associated with the test statistic are all necessary in data analysis. The categorical variable's degrees of freedom (df) can be found using the equation below (TURHAN, 2020):

$$df = k - 1 \tag{5.10}$$

After calculating the χ^2 value and the df value, the critical value may be obtained using the Table 2. The H_0 can then be rejected or accepted using the following decision rule:

- If $\chi^2_{\text{calculated}} >$ Critical value: Reject H_0
- If $\chi^2_{\text{calculated}} \leq$ Critical value: Fail to reject H_0

Table 2 – Chi-Square Critical Values (Adapted from (PANDIS, 2016))

df	$\alpha = 0.10$	$\alpha = 0.05$	$\alpha = 0.01$	$\alpha = 0.001$
1	2.706	3.841	6.635	10.828
2	4.605	5.991	9.210	13.816
3	6.251	7.815	11.345	16.266
4	7.779	9.488	13.277	18.467
5	9.236	11.070	15.086	20.515

5.3 PREPARATION

Just as there is a great variety of languages spoken around the world, the same is true for Android applications. Considering also the specific language variations depending on the

region, this list becomes even larger. For example, for the Spanish language, we can have a great variety of locales depending on the region, which can be Spain, Chile, Peru, Uruguay, Mexico, etc. [1].

Each of these locales will present specific nuances, despite being the same language, related to their region. The Android system provides support for over 70 different locales; however, not all Android applications developed include all locales. There are some locales that are more commonly supported by Android applications due to the target audience and market reach. Based on this, the selection of locales for this experiment was made based on three criteria: First, five locales were chosen based on development trends (HRESKO, 2025).

Second, we also added the ar-EG locale to represent RTL locales. These locales represent a different disposition of elements in UI, and it's usually challenging to adapt; therefore, we should also consider it in this study. And third, the pt-BR locale was chosen because it is the native language of the authors, helping in the identification of specific nuances and translation suggestions. The locales chosen to be part of this experiment are:

- English US (en-US)
- Chinese Simplified (zh-CN)
- German Germany (de-DE)
- Japanese Japan (ja-JP)
- Portuguese Brazil (pt-BR)
- Arabic Egypt (ar-EG)
- Spanish Spain (es-ES)

On an Android device, the usual way to obtain an application is through the Play Store. However, there are other Android app stores where you can find different applications. Among these available stores, there is Obtainium [2], which is a Free and Open Source Software (FOOS) application store that attempts to install directly from the source, which can be the Play Store itself, GitHub, or F-Droid [3].

[1] https://cldr.unicode.org/translation/getting-started/guide#TOC-Regional-Variants-also-known-as-Sub-locales-
[2] https://obtainium.imranr.dev/
[3] https://f-droid.org/en/about/

Unlike Obtainium, F-Droid has its own application repository where you can download the APK directly from the F-Droid website. Since it is also a FOSS store, it allows access to the project repository, access to the source code, and ease of reporting issues to the developers. In addition, it also provides an easy view of the versions and dates of the latest version updates. For these reasons, the Android application store chosen for this experiment was F-Droid.

F-Droid hosts more than 2,500 open-source projects. The apps chosen for this experiment were selected from those available on F-Droid. For this, some inclusion and exclusion criteria were defined:

- **IC1 (Inclusion Criterion 1): Has at least one locale among the other 6 chosen for this experiment, besides en-US**
- **EC1 (Exclusion Criterion 1): The most recent APK version available was released no more than six months before the start of the experiment.**

With the inclusion and exclusion criteria defined, the selection of apps was then made. A manual and random exploration was conducted in the F-Droid store. Since F-Droid does not indicate whether apps are organized by popularity or any other criteria, the apps were randomly verified according to the order in which F-Droid lists them within each category. Each app was checked for compliance with the inclusion and exclusion criteria.

Upon clicking the link for each app, it was checked which locales were supported; if they had at least one of the other 6 supported locales besides en-US (IC1), and if the date of the most recent apk version release was checked, and if it was less than six months (EC1), the app was chosen. The time of the release should be less than six months because it could indicate that the project was still being maintained. The verification stopped when 50 qualifying apps were identified, resulting in Table 3.

Table 3: Apps selected

ID	APP	Explored Version	Source Code
A1	Recurring Expenses	0.13.1 (34)	RecurringExpenseTracker
A2	Oinkoin	1.0.88 (6081002)	oinkoin
A3	KitchenOwl	0.6.10 (108)	kitchenowl
A4	My Expenses	3.9.5 (779)	MyExpenses

Continued on next page

Table 3: Apps selected (Continued)

ID	APP	Explored Version	Source Code
A5	Currencies: Exchange Rate Calculator	1.22.5 (12205)	currencies
A6	OpenMoneyBox	3.5.1.2 (62)	openmoneybox
A7	Persian Calendar	9.7.0 (970)	persian-calendar
A8	Petals	3.35.4 (3035004)	Petals
A9	Vacation Days	16.0 (16)	VacationDays
A10	Everyday Tasks	1.7.5 (10705)	EverydayTasks
A11	Super Productivity	13.0.4 (1300040000)	super-productivity
A12	MinCal Widget	2.18.1 (87)	min-cal-widget
A13	BetterCounter	5.0.3 (50003)	bettercounter
A14	Chrono	0.6.0 (283)	chrono
A15	Stocks Widget	3.9.835 (300900835)	StockTicker
A16	Fitness Calendar	2025.02.1 (12)	FitnessCalendar
A17	mLauncher - Minimal and Clutter Free launcher	1.7.8 (10708)	mLauncher
A18	Dollphone Icon Pack	1.1.4-hotfix2 (14)	dollphone
A19	ColorBlendr	v1.11.5 (26)	ColorBlendr
A20	Easy Launcher - Minimal launcher	0.3.2 (32)	EasyLauncher
A21	Slideshow Wallpaper	1.2.2 (11)	SlideshowWallpaper
A22	Counter	35 (35)	counter
A23	Rosarium	Judices (11)	Rosarium
A24	TransektCount	4.2.4 (424)	TransektCount
A25	Tournant	2.9.7 (34)	Tournant
A26	UnitsTool	1.0.13 (9030)	UnitsTool
A27	Energize	v0.12.2 (27)	Energizer
A28	Geological Time Scale	0.7.1 (14)	GeologicalTimescale
A29	Rush	3.2.1 (3210)	Rush
A30	Kotatsu	8.1.3 (1009)	Kotatsu
A31	Musify	9.2.0	Musify

Continued on next page

Table 3: Apps selected (Continued)

ID	APP	Explored Version	Source Code
A32	App Manage	4.0.2 (442)	AppManager
A33	Clauncher	v5.2.6 (380)	CLauncher
A34	Save Locally	1.4.3 (29)	SaveLocally
A35	CLT 2025 Schedule	1.69.0-CLT-Editior (107)	CampFahrplan
A36	Tarnhelm	1.8.0 (20250221)	Tarnhelm
A37	FairEmail	1.2277 (2277)	FairEmail
A38	News Reader	1.11 (111)	rssreader
A39	Raccoon	0.4.2 (90)	RaccoonForFriendica
A40	Fennec F-Droid	139.0.4 (1390420)	Fenix
A41	Fedilab	3.32.3 (532)	Fedilab
A42	Tuta Calendar	287.250527.0 (133)	tutanota
A43	FeedFlow - RSS Reader	1.1.6 (1412)	feed-flow
A44	SCEE	61.1 (6102)	SCEE
A45	Bangle.js Gadgetbridge	0.84.0-banglejs (237)	Gadgetbridge
A46	Tridenta	1.5 (60)	tridenta
A47	traced it	0.9.0 (6)	traced-it-android
A48	Ladefuchs	3.1.9 (334)	ladefuchs-react-native
A49	Sky Map	1.10.9 (1560)	stardroid
A50	FOSDEM 2025 Schedule	1.69.1-FOSDEM-Edition (108)	CampFahrplan

5.4 EXECUTION

Before starting the experiment, the necessary inputs for the approach were prepared. With the list of selected apps defined (Table 3), a .txt file was created containing the URLs of all chosen applications, which served as input.

It is important to note that, in a real-world scenario, testers typically do not analyze such a large number of apps simultaneously. However, for the purposes of this study, the file

included 50 app URLs to allow for broader evaluation. Once this initial setup was complete, the command line was used to launch the experiment with the appropriate exploration parameters.

Figure 9 provides an overview of the experiment executed using our semi-automated approach on our set of 50 apps from F-Droid, explored in all supported locales among the 7 locales selected.

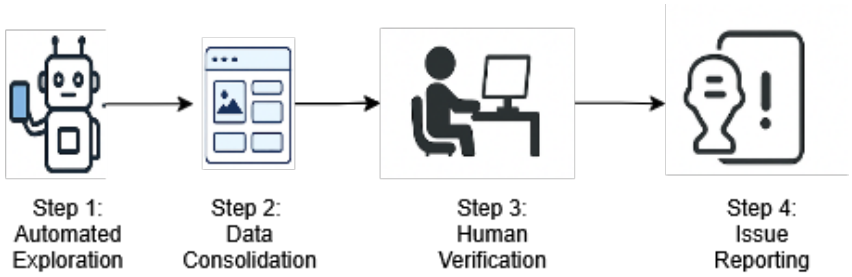


Figure 9 – Experiment Execution

- **Step 1: Automated Exploration**: All the apks were downloaded. The application was then installed and executed under each supported locales. `Droidbot` navigates through several screens, interacting with clickable elements to explore the app's user interface and capture screenshots.
- **Step 2: Data Consolidation**: Once the exploration is complete, all screenshots are organized into a structured report. All the screens captured in the previous step are grouped by locale to help the tester analyze them.
- **Step 3: Manual Verification**: A `L10n/i18n` specialist with three years of experience in g11n testing examines the report to verify the identified issues. During this process, all reported issues are tested in the latest version of the application, which is also necessary when the execution is manual. This step is crucial to ensure that the issues are still reproducible in the most current version, confirming that the problem persists and has not been resolved by recent updates.
- **Step 4: Issue Reporting**: The issues highlighted in the previous step are reported on each app project's "Issue tracker" with the screenshots attached and the description of the issue and locales.

The semi-automated approach proposed was employed to explore 50 apps from F-Droid. Each app was tested for 3 hours per supported locale, across the seven target locales. The

exploration ran continuously, 24/7, throughout the duration of the experiment. To ensure consistency in the results, a standardized emulator configuration was used: Pixel 7, running API 31.

5.5 THREATS TO VALIDITY

Internal Validity: Manual failure classification presents potential threats. While involving screenshot verification across locales, this classification requires locale-specific knowledge and a certain expertise. Misinterpretations may occur when distinguishing actual failures from locale-specific behaviors (e.g., cultural formatting). However, developer validation of reported issues mitigated this concern.

External Validity: Generalizability limitations stem from a limited subset of F-Droid apps and the results might not apply to all Android apps, even though we observed 50 open-source apps. Also, the number of locales chosen was also limited to a subset. Mitigating this threat can only be achieved through additional studies that consider different applications, domains, locales, and additional research questions.

Conclusion Validity: There is also a conclusion validity threat regarding the violation of the assumptions of the statistical methods used. For our study, we consider such a threat to be low, as we used a non-parametric test, the Chi-Square Goodness of Fit Test, which does not rely on assumptions about the distribution of the data. Although the observed significant results in our study are in line with our expectations, additional studies using different subjects should be conducted to minimize this threat.

Construct Validity: To keep conditions under control and consistent throughout the experiment, only one smartphone model and one version of Android were chosen. This made it possible for us to concentrate on finding problems and analyzing how the app behaved in a controlled setting. We do acknowledge, though, that the findings might not apply to other Android versions or devices, highlighting the necessity of more hardware diversity and operating system variations in future studies to provide a more thorough assessment.

5.6 CONCLUDING REMARKS

This chapter presented the planning and execution of the experimental study designed to evaluate the proposed semi-automated approach for identifying and reporting L10n/i18n fail-

ures in Android applications. Following the GQM methodology, we defined the study's objectives, research questions, and evaluation metrics. We detailed the experimental design, including hypotheses, variables, and the statistical method adopted, as well as the criteria used for selecting the apps and locales. Furthermore, we described the step-by-step execution of the experiment and addressed potential threats to validity.

6 RESULTS

In this chapter, we describe the analysis of the results for each of the following research questions:

RQ1 *What is the usefulness of the proposed semi-automated approach in* L10n/i18n *for finding and reporting* L10n/i18n *failures?*

RQ2 *What types of* L10n/i18n *failures are the most common in the evaluated open-source apps?*

RQ3 *How relevant are* L10n/i18n *issues for developers in the context of open-source apps?*

RQ4 *How efficient is the proposed approach?*

6.1 ANSWER TO RQ1: USEFULNESS

Once the execution, with all configured inputs detailed in Section 5.4 finished, a report was generated with all the screens captured during the process, needing only a manual verification executed by a tester with three years of experience. As result, our proposed semi-automated approach successfully identified real-world L10n/i18n failures.

Table 4 presents the failures identified during the study. The "Failure ID" column provides a unique identifier for each issue, while the "App ID" column refers to the application in which the failure was found. The application names are listed in Table 3. The "Failure Type" column categorizes each issue according to the L10n/i18n failure types introduced in Section 2.4. One additional category, *Typo*, is introduced here. This is a failure that occurs when, despite having been translated, the word has some grammatical error.

In total, 41 failures were identified across 12 applications out of the 50 selected. All locales were affected by at least one failure. For the sake of space, we do not detail every failure individually; instead, we summarize the main failures found, grouped by app, to provide a broad view of the failures.

Table 4: Identified L10n/i18n Failures

Failure ID	App ID	Failure Type	Issue Description
F01	A5	Typo	String is misspelled for pt-BR.
F02	A5	Overlapping	UI elements are overlapping for ar-EG.
F03	A8	Not Localized	Decimal format is incorrect for de-DE.
F04	A8	Not Localized	Decimal format is incorrect for pt-BR.
F05	A8	Not Localized	Date format is incorrect for es-ES.
F06	A8	Not Localized	Date format is incorrect for pt-BR.
F07	A8	Missing Translation	String is not translated for de-DE.
F08	A8	Missing Translation	String is not translated for es-ES.
F09	A8	Missing Translation	String is not translated for es-ES.
F10	A8	Missing Translation	String is not translated for de-DE.
F11	A8	Missing Translation	String is not translated for de-DE.
F12	A11	Overlapping	UI elements are overlapping for ar-EG.
F13	A16	Missing Translation	String is not translated for de-DE.
F14	A16	Missing Translation	String is not translated for de-DE.
F15	A17	Not Localized	String misaligned for RTL ar-EG.
F16	A17	Not Localized	String misaligned for RTL ar-EG.
F17	A19	Missing Translation	String is not translated for ja-JP.
F18	A21	Inconsistency	Different values for same word for es-ES
F19	A27	Not Localized	Decimal format is incorrect for pt-BR.
F20	A32	Not Localized	String misaligned for RTL ar-EG
F21	A32	Not Localized	String misaligned for RTL ar-EG.
F22	A32	Not Localized	String misaligned for RTL ar-EG.
F23	A32	Not Localized	String misaligned for RTL ar-EG.
F24	A33	Missing Translation	String is not translated for ar-EG.
F25	A33	Missing Translation	String is not translated for de-DE.
F26	A33	Missing Translation	String is not translated for pt-BR.
F27	A33	Missing Translation	String is not translated for es-ES.
F28	A33	Missing Translation	String is not translated for zh-CN.
F29	A33	Missing Translation	String is not translated for ja-JP.

Continued on next page

Table 4: Identified L10n/i18n Failures (Continued)

Failure ID	App ID	Failure Type	Issue Description
F30	A33	Not Localized	Strings misaligned for RTL r-EG.
F31	A33	Missing Translation	String is not translated for ar-EG.
F32	A33	Missing Translation	String is not translated for de-DE.
F33	A33	Missing Translation	String is not translated for pt-BR.
F34	A33	Missing Translation	String is not translated for es-ES.
F35	A33	Missing Translation	String is not translated for zh-CN.
F36	A33	Missing Translation	String is not translated for ja-JP.
F37	A36	Missing Translation	String is not translated for ja-JP.
F38	A36	Missing Translation	String is not translated for ja-JP.
F39	A48	Missing Translation	String is not translated for en-US.
F40	A48	Missing Translation	String is not translated for en-US.
F41	A48	*Missing Translation*	String is not translated for en-US.

App A5 (Currencies) presented one failure classified as *Typo* for pt-BR, which means that the word was misspelled. The word shown on screen was "Aspeto," but it should be "Aspecto." The app also presented an *Overlapping* failure associated with the locale ar-EG. For app A32 (App Manager), four *Not Localized* failures were identified, and all were related to misalignment in the ar-EG locale. Examples of these failures are presented in Figure 10.

The app A8 (Petals) presents four failures classified as *Not Localized*, involving incorrect formatting of dates and decimal numbers for the locales de-DE, pt-BR, and es-ES. For instance, for pt-BR the date format should follow DD-MM-YYYY, presenting something like "03-05-2025"; however, it is presented as "2025-05-03" on screen.

For de-DE, decimal numbers should be separated by a comma, like in "0,00"; instead, it was presented as "0.00." In addition, five *Missing Translation* failures were detected. Examples of these failures are illustrated in Figure 11.

App A11 (Super Productivity) presented one failure related to *Overlapping*. The failure is reproducible in any locale; however, as it is a failure that occurs in the source, we consider it one issue only. On the other hand, app A19 (Color Blendr) presented one *Missing Translation* failure for ja-JP.

App A21 presented an *Inconsistency* failure. In other words, there is a word that presents different values for es-ES but has the same value for source en-US. The word is presented

as "Antialiasing" in one of the strings, and right above it is presented as "Anti-aliasing" in another string. Examples of these failures are presented in Figure 12.

App A16 (Fitness Calendar) had two strings *Missing Translation*, related to the locale de-DE. In other hand, app A17 (mLauncher) presented two *Not Localized* failures, both involving string misalignment in the ar-EG locale, which is an RTL locale. Such issues directly impact usability for users in RTL locales. The failures for A16 and A17 are illustrated in Figure 13.

App A27 (Energize) showed one *Not Localized* failure, related to an incorrectly formatted decimal value. The pt-BR locale requires decimal values to be separated by a comma (","), but in this case, a dot (".") was used instead. Meanwhile, App A36 (Tarnhelm) had two *Missing Translation* failures, both affecting the ja-JP locale. Figure 14 provides examples of these issues.

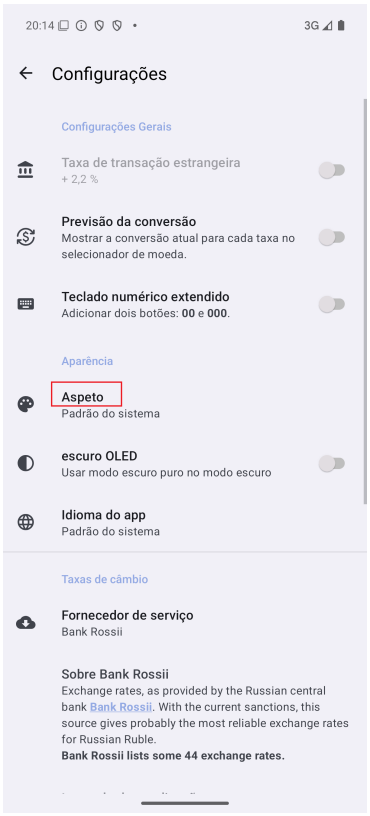
App A33 (Clauncher) had the highest number of failures, totaling 13. Among these, twelve were classified as *Missing Translation* and one as *Not Localized*, presenting a misalignment failure for RTL locale. The problems extend to a range of locales, including ar-EG, de-DE, pt-BR, es-ES, zh-CN, and ja-JP. Representative examples are shown in Figure 15.

App A48 (Ladefuchs) presented three *Missing Translation* failures, all occurring within the en-US locale. This case is particularly interesting given that en-US is commonly used as the default or source locale in most software projects. However, in this instance, the original source content was in de-DE, and the en-US version was expected to be a translated target locale. Figure 16 illustrates examples of the failures identified in this application.

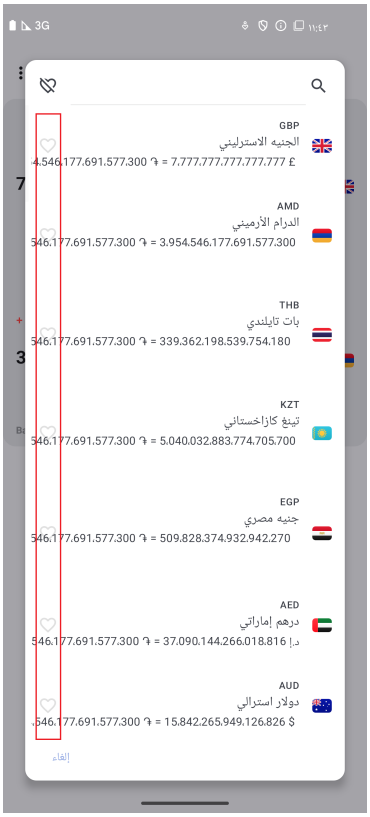
Most of the RTL-related failures were due to misalignment issues. These failures are examples of internationalization problems that became visible to end users. They occurred because the app's design did not fully account for the requirements of RTL locales, particularly the need to mirror UI elements appropriately.

Response to RQ1

A total of 50 apps were explored using our semi-automated approach. Of these apps, there were 237 exploration rounds, considering that each app was explored once for each supported locale. At the end of this experiment's exploration, 47,658 screens were captured and analyzed. After analyzing these screens, a total of 41 L10n/i18n failures were identified.



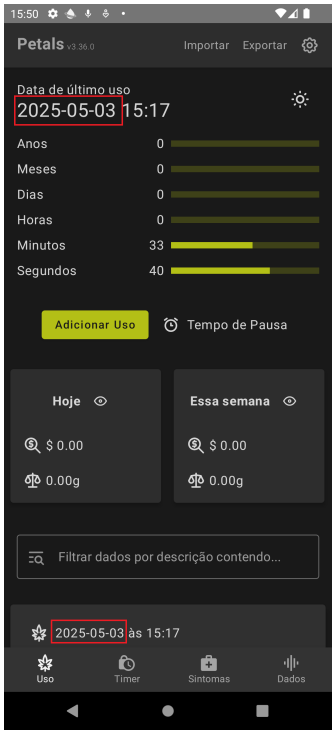
(a) Typo for pt-BR (F01)



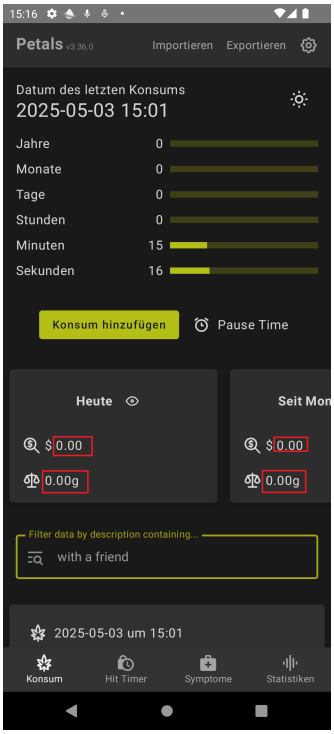
(b) Overlapping for ar-EG (F02)

(c) Not Localized for ar-EG (F22)

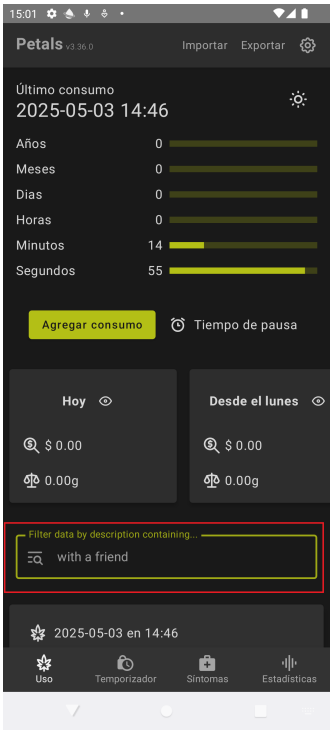
Figure 10 – Currencies and App Manager Failures



(a) Not Localized for pt-BR (F06)

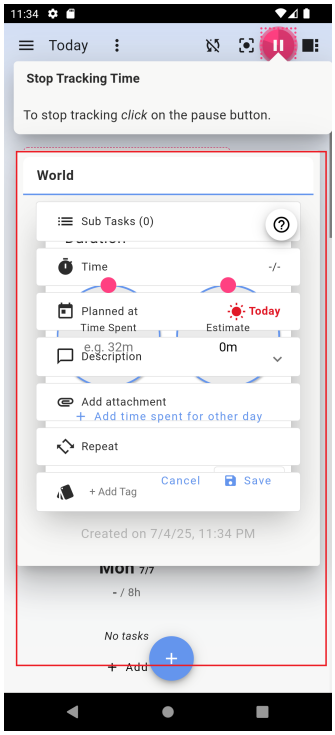


(b) Not Localized for de-DE (F03)

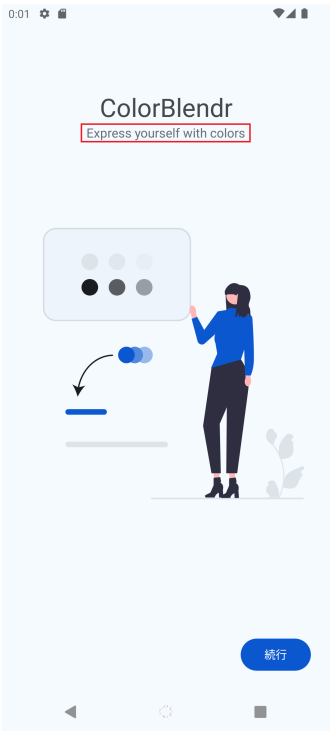


(c) Missing Translation for es-ES (F08)

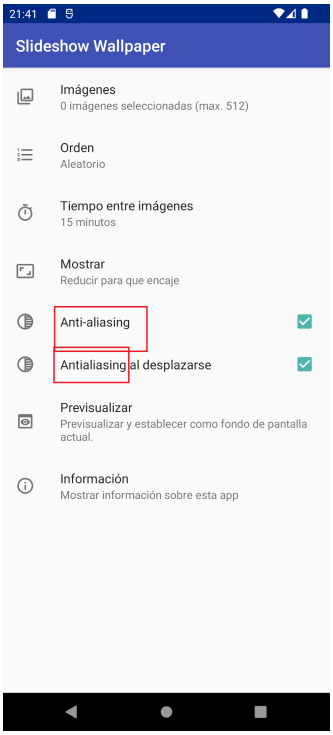
Figure 11 – Petals Failures



(a) Overlapping for en-US (F12)

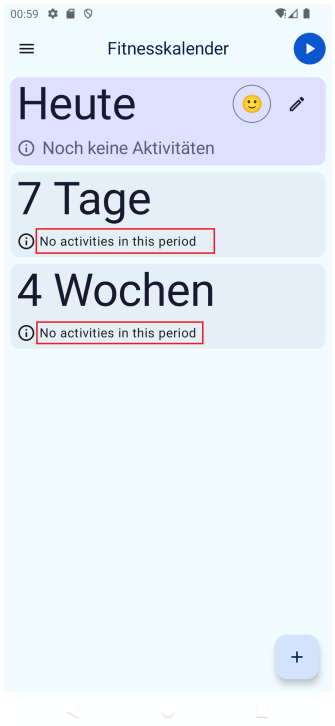


(b) Missing Translation for ja-JP (F17)



(c) Inconsistency for es-ES (F18)

Figure 12 – Super productivity, Color Blendr and SlideShow Wallpaper Failures

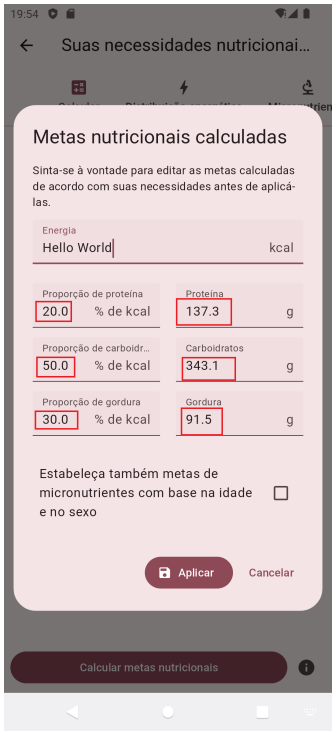


(a) Missing Translation for de-DE (F14)

(b) Not Localized for ar-EG (F15)

(c) Not Localized for ar-EG (F16)

Figure 13 – Fitness Calendar and mLauncher Failures



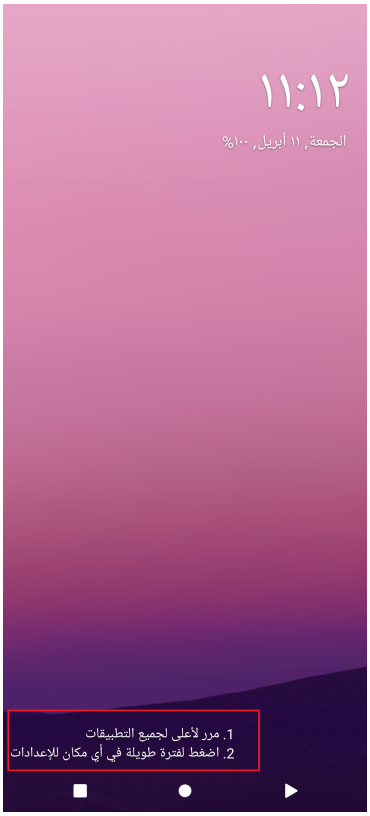
(a) Not Localized for pt-BR (F19)

(b) Missing Translation for ja-JP (F37)

(c) Missing Translation for ja-JP (F38)

Figure 14 – Energize and Tarnhelm Failures

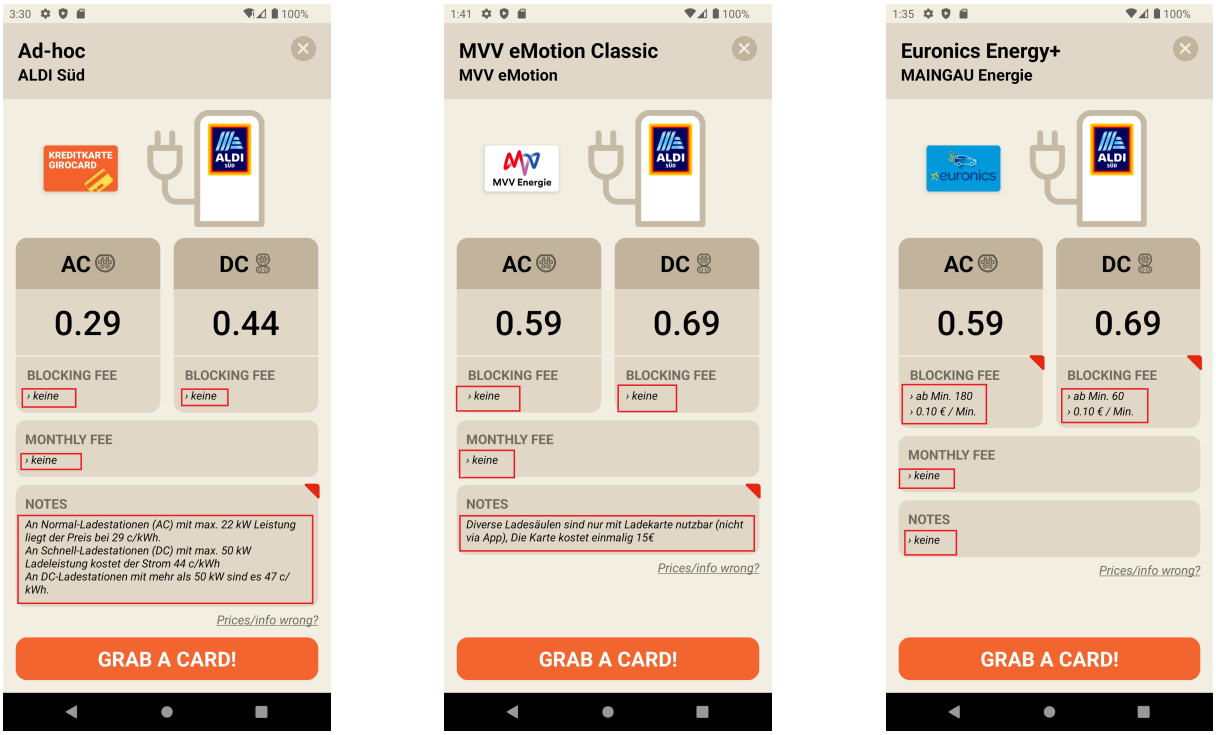
(a) Missing Translation for zh-CN (F28)



(b) Not Localized for ar-EG (F30)

(c) Missing Translation for ja-JP (F36)

Figure 15 – cLauncher Failures



(a) Missing Translation for en-US (F39)

(b) Missing Translation for en-US (F40)

(c) Missing Translation for en-US (F41)

Figure 16 – Ladefuchs Failures

6.2 ANSWER TO RQ2: MOST COMMON TYPES OF L10N/I18N FAILURES

After the automated exploration, a manual verification was performed on the captured screenshots. During the manual verification for the identification of failures, some categories of failures mentioned in Chapter 2 section 2.4 were identified. Among the 41 reported failures, five different categories were identified. They are detailed in Table 5:

Failure Type	**Frequency**
Missing translation	25
Not localized	12
Inconsistency	1
Overlapping	2
Typo	1
Total	**41**

Table 5 – L10n/i18n reported per category

Some categories were found to be more prevalent than others. To find out if there was a statistically significant difference between the failures, we conducted a hypothesis test. The test Chi-Square was selected for this experiment, as stated in Chapter 5.

Our hypotheses are better detailed in Chapter 5, and are summarized below:

- **Null Hypothesis:** All types of `L10n/i18n` failures occur with equal frequency in open-source apps.
- **Alternative Hypothesis:** Some types of `L10n/i18n` failures occur significantly more frequently than others.

The observed frequency must be at least 10 or higher in order to apply the Chi-square (TURHAN, 2020). Therefore, only two categories that met the minimal frequency criteria were taken into consideration for this test, despite the fact that we discovered five categories. Therefore, the categories used are:

Failure Type	**Frequency**	**Percentage**
Missing translation	25	67.57%
Not localized	12	32.43%
Total	**37**	**100%**

Table 6 – Observed frequencies of primary failure types

Considering the frequency of 25 issues for *Missing Translation* and 12 for *Not Localized*, n= 37 (Total of sample size) and p= 2 (Hypothesized rate of the observations). Thus, the expected frequency per category is 18.5 $(37/2)$:

Therefore:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i} \quad = \frac{(25 - 18.5)^2}{18.5} + \frac{(12 - 18.5)^2}{18.5} = \frac{42.25}{18.5} + \frac{42.25}{18.5} \quad = 4.57 \tag{6.1}$$

Degrees of freedom:

$$df = 2 - 1 = 1 \tag{6.2}$$

Critical value at:

$$\alpha = 0.05 : \chi^2_{0.05,1} = 3.84 \tag{6.3}$$

Conclusion: Since $\chi^2 = 4.57 > 3.84$, we reject H_0 at $p < 0.05$.

Response to RQ2

After data analysis from every failure reported in this experiment, we ran statistical tests to see if any particular failure stood out from the rest or if all the failures happened at the same frequency. We rejected H_0 after looking at the statistical test results. We therefore draw the conclusion that certain L10n/i18n failures are statistically more common than others. More precisely, *Missing Translation* failures occur significantly more frequently than other L10n/i18n failures.

6.3 ANSWER TO RQ3: RELEVANCY OF L10N/I18N ISSUES

Among the 50 apps explored using Droidbot, 12 apps presented L10n/i18n failures. A total of 41 failures were found and reported back to developers in their open-source projects. To avoid spam of reports, since some apps presented more than one failure observed, some failures were reported on the same issue report.

A total of 14 issue reports were opened on open-source projects of the apps. There were 33 failures accepted by developers, which means that they agree that they were real failures. Even though the failures were grouped in the same report, the response from the developers were addressed to the failures to respond which they would fix or not, and why. Details of the reported issues are presented in Table 7.

The "App ID" column refers to the unique identifier of each app, while "App Name" lists the corresponding app names. "Issue ID" provides the identifier of each reported issue, and "Issue Description" offers a brief explanation of the failures observed. The "Failures" column shows the number of failures identified, and "Locales" lists the affected locales. Developer feedback is summarized in the "Response" column, followed by the "Status" column, which indicates whether the status of the issue after being accepted or rejected.

To provide context to developers regarding the data L10n/i18n failures, the issue descriptions included an explanation about the type of issue and screenshots attached showing the exact screen where the issue occurred and the strings pointed on screen. After this contextualization, for each failure, the steps to reproduce the failure, the expected result, and, for pt-BR a suggestion for *Missing Translation* cases, were added. There are a total of 5 failures fixed with good feedback from developers (e.g., Figure 17).

At the time of writing, 2 issues have been Rejected by the developers. Next, we will describe

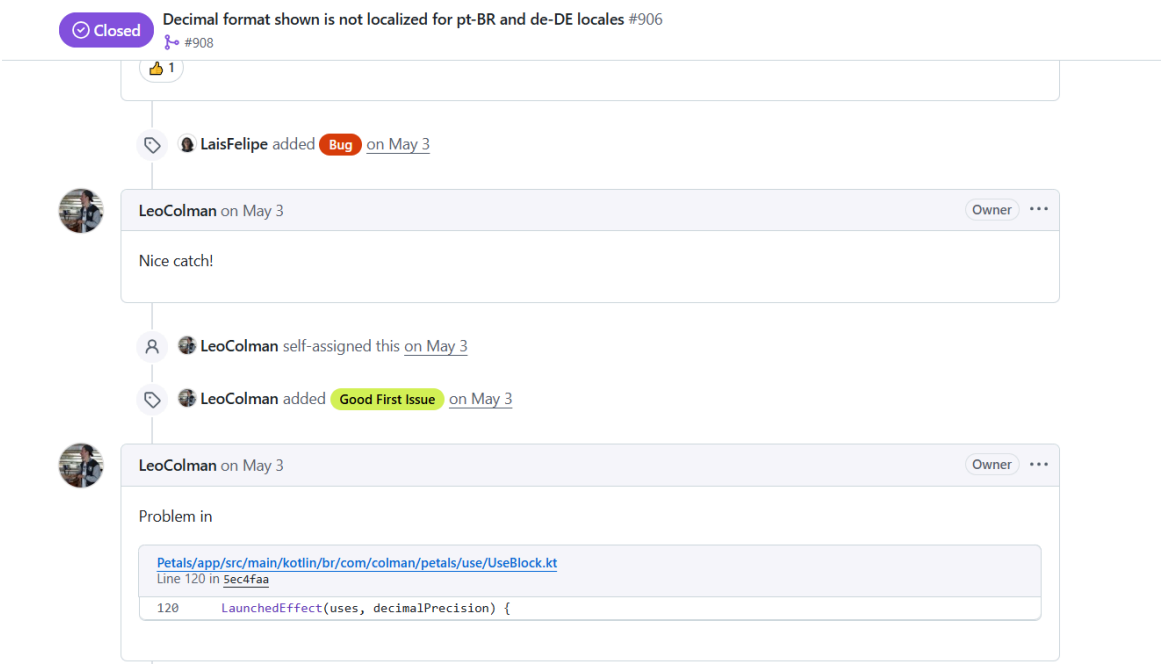


Figure 17 – Feedback Fixed issue

each of them in detail:

- **Issue #47 for app A48(Ladefuchs):** There were strings that were not translated, for en-US locale, on some screens as the example shown in Figure 18. For this issue reported, the developers closed it as LLM spam.
- **Issue #905 for app A8(Petals):** The app shows the date of the user's logged activity; however, when the locale is changed, the date format does not follow the locale set as the example shown in Figure 19 for pt-BR. The issue was reported for pt-BR and es-ES. The developer considered, when rejecting it, that it was a customized field through settings, and they wouldn't adapt it for specific locales.

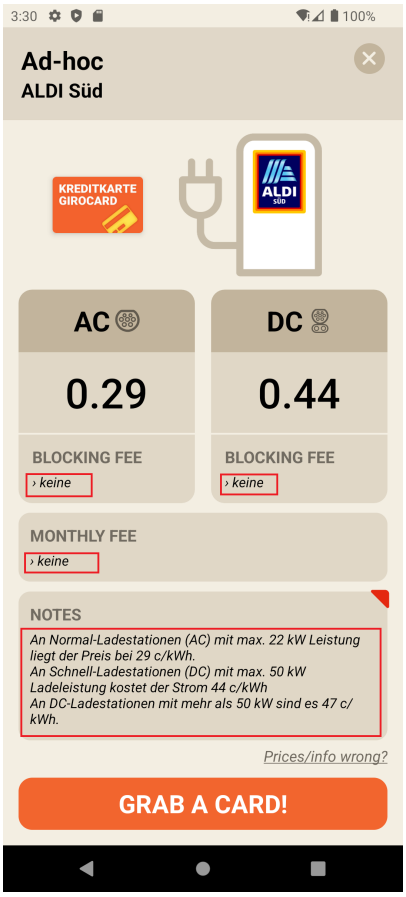


Figure 18 – Ladefuchs

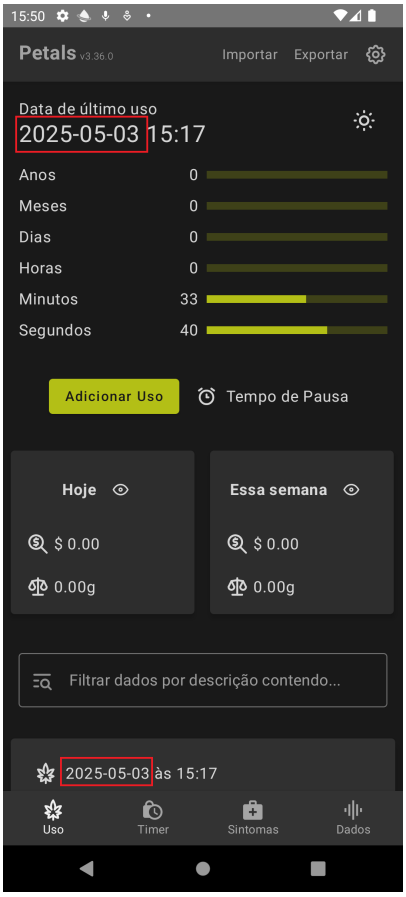


Figure 19 – Petals

Four reported issues were acknowledged by developers as valid but were ultimately closed as Wontfix because they involved *Missing Translation* cases (#907, #221, #48, and #68), and the developers didn't have fluency in the target language to provide an accurate translation.

The type of failure that was reported with more frequency was *Missing Translation*. A reason for that could be because most open-source projects count on contributors to translate to other locales, which sometimes leads to having locales with only some of the strings translated.

During the reports, we noticed that some projects had a percentage indicator of translation for each locale. For these cases, when we found a *Missing Translation* issue, we didn't report it if it was already indicating that the locale wasn't fully translated. But as not all projects had this indicator, we opened issues related to *Missing Translation* when we couldn't find this information.

App ID	App Name	Issue ID	Issue Description	Failures	Locales	Response	Status
A5	Currencies	#69	Typo, Overlapping	2	pt-BR, ar-EG	Waiting reply	-
A8	Petals	#906	Decimal Units Not Localized	2	pt-BR, de-DE	Accepted	Fixed
A8	Petals	#907	Missing translations	5	de-DE, es-ES	Accepted	Wontfix
A8	Petals	#905	Date Format Not Localized	2	pt-BR, es-ES	Rejected	Closed
A11	Super productivity	#4701	Overlapping	1	en-US	Accepted	Analysis
A16	Fitness Calendar	#57	Missing Translation	2	de-DE	Accepted	Fixed
A17	mLauncher	#731	Not Localized: Misalignment in RTL	2	ar-EG	Accepted	Todo
A19	Color Blendr	#221	Missing Translation	1	ja-JP	Accepted	Wontfix
A21	Slideshow Wallpaper	#72	Inconsistency	1	es-ES	Waiting reply	-
A27	Energize	#225	Decimal Format Not Localized	1	pt-BR	Accepted	Todo
A32	App Manager	#1695	Not Localized: Misalignment in RTL	4	ar-EG	Accepted	Analysis
A33	Clauncher	#48	Not Localized: Misalignment in RTL	1	ar-EG	Accepted	Fixed
A33	Clauncher	#48	Missing Translation	12	ar-EG, de-DE, pt-BR, es-ES, zh-CN, ja-JP	Accepted	Wontfix
A36	Tarnhelm	#68	Missing Translation	2	ja-JP	Accepted	Wontfix
A48	Ladefuchs	#47	Missing Translation	3	en-US	Rejected	Closed
Total				41			

Table 7 – Opened issues

Response to RQ3

A total of 41 bugs were reported in 14 issues, with 12 (85.71%) issues receiving responses from the developers. Among those addressed, 10 L10n/i18n issues confirmed. These accepted issues correspond to 33 failures, representing 80.49% of the addressed issues. Thus, it can be inferred that the L10n/i18n failures are relevant to developers.

6.4 ANSWER TO RQ4: EFFICIENCY

This study proposes an approach for the automated exploration of Android apps. A total of 50 open-source applications available on the F-Droid Android app store were tested using the approach. Each application was systematically explored using an automated tool capable of interacting with the app's interface and capturing screenshots throughout the process.

Considering similar studies on generation testing, such as (ADAMO et al., 2018), which suggests that over 2 hours of testing is a reasonable duration, and (SU et al., 2017), which proposes 3 hours as sufficient, this study conducted the exploration for a fixed duration of 3 hours per supported locale using Droidbot.

The semi-automated approach proposed in this study also aims to reduce the manual effort associated with L10n/i18n testing activities by reducing the time and effort spent exploring apps manually. Our results show that it was possible to detect real-world L10n/i18n failures while saving manual exploration time.

Taking app A4 (My Expenses) as an example, it was tested for three hours per locale, and since the app supports all seven target locales, the total exploration time amounted to twenty-one hours. Directories with all of the screenshots and associated artifacts were automatically created for additional examination after the exploration was over.

Given the challenges of L10n/i18n testing, the approach helped optimize the time required to perform this activity by reducing the time needed for testing, as the tester performed the verification on the captured screens instead of having to manually explore the entire application for each locale.

Instead of manually navigating through each application for each supported locale, the testers were able to review pre-collected visual data, reducing the need for redundant exploratory work. For example, instead of reinstalling an app and repeatedly switching the device's language to check if a date format appeared correctly, the tester could simply open the

generated report and compare the screenshots between the languages side by side. Additionally, the automated exploration operated continuously, 24 hours a day, executing in parallel with manual verification tasks, which further contributed to overall time savings.

Table 8 highlights the total exploration time (Te), the total analysis time (Ta) of the screenshots, the total experiment time (Tt), and the number of supported locales (Loc) for each app. In total, the experiment required 44,204 minutes, with 42,660 minutes spent on automated exploration and 1,544 minutes on manual screenshot analysis. On average, this corresponds to approximately 31 minutes of manual analysis and 14 hours of automated exploration per application.

Considering that the manual analysis time will happen with or without our approach, our approach primarily saves the tester's time from the exploration phase. This amounts to a time saving of 42,660 minutes, equivalent to approximately 89 workdays assuming 8 hours of work per day.

Table 8: Time Spent Experiment

App ID	App Name	Loc	Te (min)	Ta (min)	Tt (min)
A1	Recurring Expenses	5	900	66	966
A2	Oinkoin	6	1080	22	1102
A3	KitchenOwl	6	1080	20	1100
A4	My Expenses	7	1260	130	1390
A5	Currencies	6	1080	70	1150
A6	OpenMoneyBox	2	360	9	369
A7	Persian Calendar	6	1080	300	1380
A8	Petals	4	720	25	745
A9	Vacation Days	5	900	15	915
A10	Everyday Tasks	4	720	10	730
A11	Super Productivity	6	1080	90	1170
A12	MinCal Widget	6	1080	21	1101
A13	BetterCounter	4	720	27	747
A14	Chrono	5	900	20	920
A15	Stocks Widget	4	720	11	731
A16	Fitness Calendar	2	360	9	369
A17	mLauncher	6	1080	33	1113
A18	Dollphone Icon Pack	3	540	22	562
A19	ColorBlendr	7	1260	10	1270
A20	Easy Launcher	2	360	17	377
A21	Slideshow Wallpaper	3	540	11	551
A22	Counter	7	1260	65	1325

Continued on next page

Table 8: Time Spent Experiment (Continued)

App ID	App Name	Loc	Te (min)	Ta (min)	Tt (min)
A23	Rosarium	3	540	7	547
A24	TransektCount	2	360	5	365
A25	Tournant	5	900	23	923
A26	UnitsTool	2	360	19	379
A27	Energize	5	900	9	909
A28	Geological Time Scale	3	540	11	551
A29	Rush	6	1080	23	1103
A30	Kotatsu	7	1260	75	1335
A31	Musify	7	1260	10	1270
A32	App Manage	7	1260	26	1286
A33	Clauncher	7	1260	34	1294
A34	Save Locally	2	360	5	365
A35	CLT 2025 Schedule	5	900	7	907
A36	Tarnhelm	3	540	12	552
A37	FairEmail	7	1260	65	1325
A38	News Reader	4	720	32	752
A39	Raccoon	2	360	11	361
A40	Fennec F-Droid	4	720	30	750
A41	Fedilab	7	1260	23	1283
A42	Tuta Calendar	7	1260	9	1269
A43	FeedFlow - RSS Reader	5	900	14	914
A44	SCEE	6	1080	30	1110
A45	Bangle.js Gadgetbridge	7	1260	13	1273
A46	Tridenta	2	360	10	370
A47	traced it	2	360	6	366
A48	Ladefuchs	2	360	8	368
A49	Sky Map	7	1260	13	1273
A50	FOSDEM 2025 Schedule	5	900	11	911
–	**Total**	237	**42,660**	**1,544**	**44,204**

Response to RQ4

Comparing this automated exploration work with the manual, the closest we have would be the ET using session-based testing. So considering that the tester will be conducting an ET with a time-box of the same 3 hours of exploration that we ran in our automated experiment, our semi-automated approach saves all the time of the exploration effort, since the time for analysis and reporting remains the same. In the specific case of our experiment, for our 50 applications and their respective supported locales, the time of automated execution was 711 hours, or if we consider a human workday of 8 hours, it would be 89 days.

6.5 CONCLUDING REMARKS

This Chapter shows the results of the use of our semi-automated approach to help localization and internationalization testers in identifying L10n/i18n issues related in the context of open-source Android apps.

7 CONCLUSION AND FUTURE WORK

This study introduced and evaluated a semi-automated approach to support L10n/i18n testing of Android applications. The approach leverages automated UI exploration using Droidbot, coupled with human verification of screenshots, to detect and report L10n/i18n failures across multiple locales. Through an empirical study involving 50 open-source Android apps from F-Droid, we demonstrated that the approach is both useful and efficient.

Additionally, this study contributes to the industrial context since most of the issues reported were accepted and there were issues fixed. These results highlight how automation can improve the L10n/i18n testing process, increase the user experience in a variety of locales, and assist projects in the industrial context.

7.1 MAIN FINDINGS

- **Usefulness:** The proposed semi-automated approach was applied to 50 open-source Android applications from the F-Droid platform, which were automatically explored in up to seven different locales (en-US, de-DE, es-ES, pt-BR, ja-JP, zh-CN, and ar-EG), resulting in 237 rounds of execution. The manual analysis of the captured screens led to the identification of 41 L10n/i18n failures, demonstrating the practical capability of the approach to support localization and internationalization verification.

- **Relevancy:** In total 14 issues were found using our semi-automated approach were submitted to their respective open-source repositories and 33 received developer responses. Among these, 80.49% of the reported failures were accepted and confirmed by the developers, reinforcing the relevance and applicability of the approach to identify issues. Besides that, a total of 5 failures were fixed, which reinforce the relevancy of our work.

- **Most common types of failures:** A statistical analysis was conducted to determine if some types of failures occurred more frequently than others. The results indicated that *Missing Translation* failures appeared significantly more often than other categories, making it the most common issue type found across the explored apps.

- **Efficiency:** The proposed semi-automated approach explored the apps during 3 hours for each supported locale. The 237 rounds of 3 hours resulted in 711 hours of execution.

This represents the equivalent of 89 working days (based on 8-hour shifts), showing the potential of the approach to significantly reduce human effort in this testing context.

7.2 FUTURE WORK

As future work, we intend to investigate ways to automate not only the exploration but also the report verification stage, which currently requires human intervention. One direction is the adoption of computer vision and machine learning techniques to automatically classify screenshots. For instance, image similarity metrics or deep learning models could learn to detect *Overlapping*, *Truncation*, or *Missing Translation*, reducing the manual burden on testers.

Furthermore, we aim to integrate the approach with existing tools as (ARAUJO, 2024) presented, that utilize the results from `Droidbot` for the automated detection of visual or functional failures, leveraging the same set of captures generated during the exploration. This integration could expand the applicability of the approach and enrich the analysis of the reports, allowing for the detection of various failures more efficiently and with less human effort.

REFERENCES

A. Developers. *UI/Application Exerciser Monkey*. 2012. <https://developer.android.com/studio/test/monkey>. Accessed: 2025-01-22.

ADAMO, D. et al. Reinforcement learning for android gui testing. In: . New York, NY, USA: Association for Computing Machinery, 2018. (A-TEST 2018), p. 2–8. ISBN 9781450360531. Available at: <https://doi.org/10.1145/3278186.3278187>.

ANDROID. *Localize your app*. 2025. Available at: <https://developer.android.com/guide/topics/resources/localization>.

ANDROID. *Support different languages and cultures*. 2025. Available at: <https://developer.android.com/training/basics/supporting-devices/languages>.

ANDROID. *Unicode and internationalization support*. 2025. Available at: <https://developer.android.com/guide/topics/resources/internationalization>.

ARAUJO, G. M. *Detecção Automática de Falhas de Localização a partir de Imagens*. 2024.

ARCHANA, J. et al. Automation framework for localizability testing of internationalized software. In: *2013 International Conference on Human Computer Interactions (ICHCI)*. Chennai, India: Institute of Electrical and Electronics Engineers Conference, 2013. p. 1–6.

ARDIC, B. et al. Hey teachers, teach those kids some software testing. In: *2023 IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG)*. Institute of Electrical and Electronics Engineers Conference: Melbourne, Australia, 2023. p. 9–16.

AUER, S. et al. I18n of semantic web applications. In: *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part II*. Berlin, Heidelberg: Springer-Verlag, 2010. (ISWC'10), p. 1–16. ISBN 3642177484.

AWWAD, A. M. A. et al. Automated bidirectional languages localization testing for android apps with rich gui. *Mobile Information Systems*, v. 2016, p. 13 pages, 2016.

BASILI, V. R. et al. The goal question metric approach. In: MARCINIAK, J. J. (Ed.). *Encyclopedia of Software Engineering*. New York, USA: John Wiley & Sons, 1994. I, p. 528–532.

BELLER, M. et al. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 45, n. 3, p. 261–284, Mar. 2019. ISSN 0098-5589. Available at: <https://doi.org/10.1109/TSE.2017.2776152>.

BERNER, S. et al. Observations and lessons learned from automated testing. In: *Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2005. (ICSE '05), p. 571–579. ISBN 1581139632. Available at: <https://doi.org/10.1145/1062455.1062556>.

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: *2007 Future of Software Engineering*. USA: IEEE Computer Society, 2007. (FOSE '07), p. 85–103. ISBN 0769528295. Available at: <https://doi.org/10.1109/FOSE.2007.25>.

CLDR. *Unicode CLDR Project*. 2025. Available at: <https://cldr.unicode.org/>.

COCHRAN, W. G. The $\chi 2$ test of goodness of fit. *The Annals of mathematical statistics*, JSTOR, p. 315–345, 1952.

COUTO, M. et al. I10n-trainer: a tool to assist in the training of localization (l10n) and internationalization (i18n) testers. In: *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2023. (SBES '23). ISBN 979-8-4007-0787-2/23/09. Available at: <https://doi.org/10.1145/3613372.3613420>.

COUTO, M. et al. A Tool-Assisted Training Approach for Empowering Localization and Internationalization Testing Proficiency . In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, 2025. p. 711–720. ISSN 2159-4848. Available at: <https://doi.ieeecomputersociety.org/10.1109/ICST62969.2025.10988966>.

DUKES, L. et al. A case study on web application security testing with tools and manual testing. In: *2013 Proceedings of IEEE Southeastcon*. USA: Institute of Electrical and Electronics Engineers Conference, 2013. p. 1–6.

FELIPE, L. et al. Tstring: a tool to locate the target string's screen based on automatic exploration. In: *Anais do IX Simpósio Brasileiro de Testes de Software Sistemático e Automatizado*. Porto Alegre, RS, Brasil: SBC, 2024. p. 66–73. ISSN 0000-0000. Available at: <https://sol.sbc.org.br/index.php/sast/article/view/30217>.

GAROUSI, V. et al. What industry wants from academia in software testing? hearing practitioners' opinions. In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (EASE '17), p. 65–69. ISBN 9781450348041. Available at: <https://doi.org/10.1145/3084226.3084264>.

HRESKO, R. *Localization Statistics and Trends*. 2025. Available at: <https://centus.com/blog/localization-statistics-and-trends>.

ITKONEN, J. et al. How do testers do it? an exploratory study on manual testing practices. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. USA: IEEE Computer Society, 2009. (ESEM '09), p. 494–497. ISBN 9781424448425. Available at: <https://doi.org/10.1109/ESEM.2009.5314240>.

JAMROZIK, K. et al. Droidmate: a robust and extensible test generator for android. In: *Proceedings of the International Conference on Mobile Software Engineering and Systems*. New York, NY, USA: Association for Computing Machinery, 2016. (MOBILESoft '16), p. 293–294. ISBN 9781450341783. Available at: <https://doi.org/10.1145/2897073.2897716>.

JAZAYERI, M. The education of a software engineer. In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. USA: IEEE Computer Society, 2004. (ASE '04), p. .18–xxvii. ISBN 0769521312.

LI, Y. et al. Droidbot: a lightweight ui-guided test input generator for android. In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017. (ICSE-C '17), p. 23–26. ISBN 9781538615898. Available at: <https://doi.org/10.1109/ICSE-C.2017.8>.

LI, Y. et al. Humanoid: a deep learning-based approach to automated black-box android app testing. In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2020. (ASE '19), p. 1070–1073. ISBN 9781728125084. Available at: <https://doi.org/10.1109/ASE.2019.00104>.

MAO, K. et al. Sapienz: multi-objective automated testing for android applications. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2016. (ISSTA 2016), p. 94–105. ISBN 9781450343909. Available at: <https://doi.org/10.1145/2931037.2931054>.

MCKETHAN, K. A. et al. Demystifying software globalization. *Translation Journal*, v. 9, n. 2, p. 1–8, 2005.

MICROSOFT. *Locales and languages overview*. 2023. Available at: <https://learn.microsoft.com/en-us/globalization/locale/locale>.

MOLAN, G. *IMPROVEMENTS OF SOFTWARE TESTING FOR LSP The example in the case of internationalization and localization*. 2008.

MYERS, G. J. et al. *The Art of Software Testing*. 3rd. ed. .: Wiley Publishing, 2011. ISBN 1118031962.

PANDIS, N. The chi-square test. *American journal of orthodontics and dentofacial orthopedics*, Elsevier, v. 150, n. 5, p. 898–899, 2016.

PLACKETT, R. L. Karl pearson and the chi-squared test. *International Statistical Review / Revue Internationale de Statistique*, [Wiley, International Statistical Institute (ISI)], v. 51, n. 1, p. 59–72, 1983. ISSN 03067734, 17515823. Available at: <http://www.jstor.org/stable/1402731>.

RAMLER, R. et al. How to test in sixteen languages? automation support for localization testing. In: *International Conference on Product Focused Software Process Improvement*. USA: Institute of Electrical and Electronics Engineers Conference, 2017.

SANTOS, R. E. S. et al. Mind the gap: are practitioners and researchers in software testing speaking the same language? In: . IEEE Press, 2019. (CESSER-IP '19), p. 10–17. Available at: <https://doi.org/10.1109/CESSER-IP.2019.00010>.

SCHINDLER, C. et al. Towards continuous deployment of a multilingual mobile app. *International Journal*, v. 9, n. 7, 2021.

SOUZA, M. et al. On the exploratory testing of mobile apps. In: . New York, NY, USA: Association for Computing Machinery, 2019. (SAST '19), p. 42–51. ISBN 9781450376488. Available at: <https://doi.org/10.1145/3356317.3356322>.

SU, T. et al. Guided, stochastic model-based gui testing of android apps. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 245–256. ISBN 9781450351058. Available at: <https://doi.org/10.1145/3106237.3106298>.

TIJERINO, Y. Cross-cultural and cross-lingual ontology engineering. *CEUR Workshop Proceedings*, v. 687, 01 2010.

TURHAN, N. S. Karl pearson's chi-square tests. *Educational Research and Reviews*, ERIC, v. 16, n. 9, p. 575–580, 2020.

YNION, J. C. Using ai in automated ui localization testing of a mobile app. 2020.

ZHAO, C. et al. Study on international software localization testing. *2010 Second World Congress on Software Engineering*, v. 2, p. 257–260, 2010.