
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# AUTOMATED GENERATION OF TESTS FOR GNOME GUI APPLICATIONS USING AT-SPI METADATA

**AUTOMATICKÉ GENEROVÁNÍ TESTŮ PRO GNOME GUI APLIKACE Z METADAT AT-SPI**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** **Bc. MARTIN KRAJŇÁK**
**AUTOR PRÁCE**

**SUPERVISOR** **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2020**

**Brno University of Technology**
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

# Master's Thesis Specification


Student: **Krajňák Martin, Bc.**

Programme: Information Technology Field of study: Information Technology Security

Title: **Automated Generation of Tests for GNOME GUI Applications Using AT-SPI Metadata**

Category: Software analysis and testing

Assignment:

1. Get acquainted with assistive technologies providing accessibility for applications, in particular with AT-SPI.
2. Study methods for automatic test generation.
3. Design a method for analysing metadata produced by the AT-SPI framework and a method for automatic test generation based on this analysis.
4. Implement the proposed technique in a tool that will be able to generate tests for GNOME applications.
5. Test the created tool on at least 5 open-source GNOME applications.
6. Analyse the obtained results, compare the obtained test coverage with existing test suites, and discuss possible improvements of your tool for the future.

Recommended literature:

- Dadeau, F., Peureux, F., Legeard, B., Tissot, R., Julliand, J., Masson, P.-A., Bouquet, F.: Test Generation Using Symbolic Animation of Models. Model-Based Testing for Embedded Systems. CRC Press, 2011.
- Zander, J., Schieferdecker, I., Mosterman, P.J.: Model-Based Testing for Embedded Systems. CRC Press, 2017.
- Alexander, V., Benson, C., Cameron, B., Haneman, B., O'Briain, P., Snider, S.: GNOME Accessibility Developers Guide. GNOME Documentation Project, 2008.
- Laws, C., Haneman, B.: Accessible Document Navigation Using AT-SPI. Open A11y.org Accessibility Group, 2008.

Requirements for the semestral defence:

- The first two items and at least some work on the third item.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

| | |
|---|---|
| Supervisor: | **Vojnar Tomáš, prof. Ing., Ph.D.** |
| Consultant: | Pelka Tomáš, Ing., RedHatCZ |
| Head of Department: | Hanáček Petr, doc. Dr. Ing. |
| Beginning of work: | November 1, 2019 |
| Submission deadline: | May 20, 2020 |
| Approval date: | October 31, 2019 |

# Abstract

The goal of this work is the development of a tool capable of automatic test generation for GUI applications in the GNOME desktop environment. The tests are generated using metadata provided by the assistive technologies, specifically the AT-SPI. The proposed test generator utilizes the given metadata to create a model of a tested application. The model maps the event sequences that are applied on the tested application during the test generation process. The generation process involves the detection of severe bugs in the tested application. The results of the test generation process are automated test cases suitable for regression testing. The functionality of the implemented test generator was successfully verified by testing 5 open-source applications. The testing of applications performed by the proposed tool has proven the ability to reveal new bugs.



# Abstrakt

Cieľom tejto práce je vývoj nástroja na automatické generovanie testov pre aplikácie s grafickým užívateľským rozhraním v prostredí GNOME. Na generovanie testov sú použité metadáta asistenčných technológií, konrétne AT-SPI. Navrhnutý generátor testov využíva dané metadáta na vytvorenie modelu testovanej aplikácie. Model mapuje sekvencie udalostí, ktoré generátor vykoná na testovanej aplikácii počas generovania testov. Súčasťou procesu generovania je zároveň detekcia závažných chýb v testovaných aplikáciách. Výstupom procesu generovania sú automatizované testy, ktoré sú vhodné na regresné testovanie. Funkčnosť implementovaného generátora testov bola úspešne overená testovaním 5 aplikácií s otvoreným zdrojovým kódom. Počas testovania aplikácií navrhnutým nástrojom sa preukázala schopnosť detekovať nové chyby.



# Keywords

GUI testing, GNOME, AT-SPI, MBT, open-source application testing, test generation, accessibility technologies, model based testing, black-box testing

# Kľúčové slová

testovanie grafických užívateľských rozhraní, testovanie GUI, GNOME, AT-SPI, MBT, testovanie aplikácií s otvorených zdrojovým kódom, generovanie testov, asistenčné technológie, testovanie na základe modelu, black-box testovanie


# Reference

KRAJŇÁK, Martin. *Automated Generation of Tests for GNOME GUI Applications Using AT-SPI Metadata.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

# Rozšírený abstrakt

V dnešnej dobe väčšina sofvérových aplikácií využíva grafické užívateľské rozhranie (GUI). Rozhranie využíva výhody grafického akcelerátora v počítači na zjednodušenie používania softvéru. GUI aplikácie sú vyvýjané pomocou okien a ovládacích prvkov. Ovládací prvok reprezentuje grafický element popisujúci určité správanie alebo funkcionalitu. Interakcia používateľa s ovládacími prvkami generuje rôzne udalosti umožňujúce vykonávať úlohy viacerými spôsobmi.

Aj napriek tomu, že grafické užívateľské rozhrania zlepšujú použiteľnosť a flexibilitu, takisto predstavujú výzvu v testovaní softvéru, keďže testeri musia rozhodnúť, či skontrolujú všetky sekvencie udalostí, alebo len ich časť. Úsilie vynaložené na testovanie GUI aplikácií môže byť zmiernené automatizovaným testovaním softvéru. Aj keď sa za poslednú dekádu nástroje na automatizovné testovanie zlepšili, manuálne testovanie je stále najpoužívanejšou technikou v praxi. Automatizovaný proces testovania GUI aplikácií zabezpečí, že aplikácie budú testované pravidelne a zrýchly sa nájdenie možných chýb. Automatizácia a CI-CD systémy hrajú kľúčovú rolu v regresnom testovaní a to hlavne počas fázy vývoja, keď sa softvér mení častejšie.

Testovanie GUI sofvéru zahŕňa vykonanie udalostí patriacich jednotlivým komponentám GUI a monitorovanie zmien stavu programu. Testy navrhnuté pre GUI sa skladajú zo sekvencií udalostí na vstupe a kontroly zmien stavu programu. Kontrolovať je možné niekoľko indikátorov ako stav GUI, stav pamäte, chybové hlásenia, výstupy aplikácie, alebo akýkoľvek iný indikátor stavu behu programu. GUI testy kontrolujú oveľa viac ako len zdrojový kód súvisiaci len s GUI, keďže vykonané udalosti testujú aj časť zdrojových kódov patriacich mimo GUI. V prípadoch kedy aplikácia nedisponuje iným ako GUI rozhraním je testovanie pomocou GUI rozhrania jedinou možnou formou testovania aplikácie. Z týchto dôvodov je testovanie GUI kritickou súčasťou pre vývoj akéhokoľvek sofvéru s GUI.

Veľkosť a zložitosť moderných grafických užívateľských rozhraní v počte komponent a udalostí, ktoré na nich môžu byť vykonané, presahujú praktické limity analytických prístupov k testovaniu. Počet možných testov pre GUI sa zvyšuje exponenciálne s počtom udalostí a komponent v GUI aplikácii.

V tejto práci prezentujeme naše riešenie navrhnuté pre automatické generovanie testov pre GUI aplikácie v prostredí GNOME. Generátor využíva metadáta asistenčných technológií na vytvorenie modelu, z ktorého sú testy odvodené. Generátor extrahuje z vytvoreného modelu sekvencie udalostí, ktoré je možné na testovanej aplikácii vykonať. Generovanie testov prebieha sekvenčným aplikovaním udalostí na testovanú aplikáciu. Udalosti sú vykonávané pomocou asistenčných technológií, ktoré sú taktiež používané na monitorovanie stavu aplikácie, ako aj rozširovanie modelu o novonájdené stavy v aplikácii počas testovania. Počas generovania testov je zároveň aplikácia monitorovaná kvôli detekcii závažných chýb, ktoré je generátor schopný identifikovať. Navrhnutý nástroj taktiež integruje technológiu OCR, ktorá umožňuje čítanie textu z obrázkov. Táto technológia umožňuje dodatočnú kontrolu stavu testovanej aplikácie.

Implementovaným nástrojom sme otestovali 5 aplikácií. Počas testovania sme dokázali overiť funkcionalitu nami navrhnutého generátora testov, ktorý bol schopný odhaliť niekoľko nových chýb v testovaných aplikáciách. Práca zároveň dokumentuje aj obmedzenia a nedostatky, ktoré sa objavili pri testovaní pomocou navrhnutého nástroja. Testy vygenerované našim nástrojom sú vhodné na automatizované testovanie a boli nasadené v prostredí *Desktop-CI* používaným firmou Red Hat.

# Automated Generation of Tests for GNOME GUI Applications Using AT-SPI Metadata

## Declaration

I hereby declare that this thesis project was prepared as an original work by the author under the supervision of Mr. prof. Ing. Tomáš Vojnar, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Martin Krajňák
June 3, 2020


## Acknowledgements

I would like to thank prof. Ing. Tomáš Vojnar for the guidance provided during the writing of this thesis. My thanks also belongs to Ing. Tomáš Pelka and the DesktopQE team from Red Hat for helpful discussions and feedback dedicated to the implementation of this work. At last, I would like to thank my friends and my family for their support during my studies.

# Contents

# Chapter 1

# Introduction

Nowadays, the majority of software applications feature a graphical user interface (GUI). The interface takes advantage of the computers' graphics capabilities to make software easier to use [25]. Graphical applications are developed using sets of windows and widgets. A widget represents a graphical element describing certain behavior and functionality. User interaction with widgets is generating various events allowing them to perform tasks in different ways while achieving the same goal.

Despite the fact that GUIs improve usability and flexibility, they also represent a challenge for software testing as testers have to decide whether to check all sequences of events or only a subset. The effort required to test the GUIs can be reduced with automated software testing. Even though there was significant progress made in automated testing tools over the last decade, manual testing is still the most common technique in practice. However, with a proper automated GUI testing process, more test cases can be executed regularly and more faults can be found within less time [13]. Automation and CI-CD systems play an essential role in regression testing, especially in the test development phase, when software changes are more frequent. Generally, building GUI test cases involves selecting sequences of events and describing the expected state of the program after the event execution. An indicator of expected state can be a state of GUI, memory state, error log, output log, etc.

The test cases designed for GUIs test much more than the code associated only with GUI, as the events also execute underlying non-GUI code. In cases where an application has only a GUI interface, the GUI testing is the only possible form of testing. The size and complexity of modern GUIs, in terms of components and events that may be executed on them, exceed the practical limits of analytical approaches to testing. The number of possible test cases for GUI increases exponentially with the number of events and components in GUI [14].

In this thesis, we present our solution designed to generate test cases for GUI applications in the GNOME environment. The implemented generator utilizes the metadata provided by the assistive technologies to create a model of a GUI application from which test cases are derived. Further, we discuss the results achieved by testing applications with the implemented tool, the limitations discovered during the development, and the plans for future work as well.

Structure of this thesis continues as follows. Chapter 2 describes GUI testing techniques utilized by this work. Chapter 3 introduces the reader to the architecture of the accessibility technology (AT-SPI) in the GNOME environment, followed by the description of available tools and libraries. We also discuss the limitations that can occur when the AT-SPI is

used for testing as well as the technologies that might be used to cover those limitations, namely the *OpenCV* library and the optical character recognition (OCR) engine *Tesseract*. In chapter 4, we present the implementation of our test generator. Chapter 5 summarises the results we achieved during the testing with our test generator. Chapter 6 contains the evaluation of test coverage and offers the workflow recommendations for our tool. Chapter 7 concludes this thesis.

# Chapter 2

# Testing Graphical User Interfaces

Next several sections are dedicated to various testing techniques used to test GUIs. Variations of both manual and automated testing are discussed, followed by examples of tools using them. Throughout this thesis, a tested application will be referred to as a system under test (SUT).

## 2.1 Random Input Testing

The Random input testing technique is also referred to as stochastic testing or monkey testing. The term monkey is mentioned in any form of automated testing performed without any user bias. This method distinguishes 3 types of monkeys who are testing the application by generating random sequences of events from both a keyboard and a mouse. Dumb monkeys do not have any knowledge about the system, nor its state. They are not aware which actions are legal or illegal. The downside is that they cannot recognize a failure when they encounter one. Their only goal is to crash the SUT. Another group, referred to as semi-smart monkeys, can recognize a bug when they see one. The last group are smart monkeys, who have certain knowledge about the application they are testing, obtained from a state table, or a model of SUT. On the other hand, smart monkeys are the most expensive to develop. Despite the fact that a random testing tool has a weak coverage, Microsoft has reported that 10-20 % bugs in their software were discovered by this method [16].

## 2.2 Manual Testing

High-level GUI and acceptance tests are often being performed manually. Those practices are often inefficient, error-prone, and tedious. Test development tends to be delayed and executed in a hurry during late development stages. Manual tests are pre-defined sets of steps performed on a high level of system abstraction to validate the system against the required specification. However, software is prone to changes, and therefore it needs to be tested regularly against regressions. This leads to excessive costs, since testers have to continuously re-execute test plans throughout development stages [4].

## 2.3 Test Automation and CI/CD

Automated testing solves the major weaknesses of manual testing. The process of automating software testing is similar to a software development process. The goal is to reduce the

need for human involvement in repetitive or redundant tasks. A list of tests that can be automated include [1]:

- functional – testing that operations perform as expected,
- regression – testing that the behavior of the system has not changed,
- exception or negative – forcing error conditions on the system,
- stress – determining the absolute capacities of the application and operational infrastructure.

Implementation of test automation leads to practices like continuous integration (CI) and continuous delivery (CD). Continuous testing goes beyond test automation and brings testing as close to software development as possible.

Continuous integration is a coding philosophy and a set of practices that drives development teams to implement small changes and check version control repositories frequently. The majority of modern applications require code development using different platforms and tools, thus the team needs a mechanism to integrate and validate changes. The goal of the CI is to establish a consistent and automated way to build, package, and test applications. With consistency in the integration process in place, teams are more likely to commit code and changes more frequently. This leads to better collaboration and software quality.

Continuous delivery starts where continuous integration ends. CD automates the delivery of applications to selected infrastructure environments. Therefore it performs necessary calls to predefined sets of services to ensure that applications are deployed.

The common goal for CI/CD is to deliver quality software and code to users. Continuous testing is often implemented as a set of automated regression, performance, and other tests that are executed in CI/CD pipelines. Automated testing frameworks help quality assurance engineers to define, execute and automate various types of tests that can help development teams know whether a software build passes or fails. Most CI/CD tools let developers kick off a build on-demand, triggered by code commit in the version control repository, or on a defined schedule.

Regression tests are an essential part of the CI/CD pipeline that directly informs developers about the effects of their changes on previously tested and stable functions of the application [23].

## 2.4 Black Box Testing

The technique handles the software as a black box. A tester has no knowledge about the implementation of the software. The design of the test cases is only based on the specifications and requirements. Tests usually involve a set of both valid and invalid inputs with predictable outputs. Black box testing plays a significant role in testing as it is evaluating the overall functionality of the software [15].

## 2.5 White Box Testing

The design of test cases depends on the implementation of the software entity. White box testing is focused on internal logic and structure of the code, testing the software from the

developer's perspective. The design of test cases requires full knowledge about softwares' sources, thus allowing one to possibly test every branch in the code. Test cases are usually written as unit tests, system tests or integration tests. White box tests are suitable for execution during the development and also when testing the finished product [15].

## 2.6 Exploratory testing

Exploratory testing is an approach to software testing that is often described as simultaneous learning, test design, and execution. It focuses on discovery and relies on the experience of the tester to find defects that are not in the scope of other tests. The goal is to complement traditional testing to find million-dollar defects that are generally hidden behind the defined workflow [20].

## 2.7 Record/Replay and Scripting Tools

To mitigate the mentioned concerns and increase the quality of software, automated testing has been proposed as a solution. A considerable amount of work has been devoted to high-level test automation, resulting in Record and Replay techniques. Tools are continuously recording the coordinates and properties of GUI components during manual user interaction. Obtained recordings can be played back to emulate user interaction and validate the correct state of the system during regression testing. These techniques have also certain limitations, which is typically sensitivity to GUI layout changes and code changes. Those changes are forcing testers to repeat the recording processes, and therefore they cause additional costs by maintaining automated tests [4].

An example of this category of tools is the open-source project GNU Xnee[1]. The project consists of a library and two applications. Test automation is one of the several use cases for this project. However, the project is limited to X11 display environments [10].

A similar approach for testing is presented by script-based frameworks. These frameworks provide scripting languages to control the GUI. Instead of performing tests manually, testers are writing scripts to automatically interact with the GUI. Scripts contain some assertions to check whether the application executed a sequence of events correctly. A violation of assertions during the test results in a test case failure. These tools are widely used across the industry. JFCUnit[2] is a tool for testing Java Swing applications. Selenium[3] is a project with a range of tools and libraries that enables automation of web applications. Robotium[4] test automation framework allows to write automatic black-box tests for the UI of Android applications. And finally, SOAtest[5] that supports integration testing for web applications by capturing user interactions directly in the browser without requiring any scripting [14].

[1] https://xnee.wordpress.com/
[2] http://jfcunit.sourceforge.net/
[3] https://www.selenium.dev/documentation/en/
[4] https://github.com/RobotiumTech/robotium
[5] https://www.parasoft.com/soatest/web-ui-testing

## 2.8 Random-Walk Tools

Unlike the previously mentioned script-based and capture/replay tools, random-walk tools do not generate test cases. They just randomly walk through the GUI and randomly execute all events they encounter. These tools are easy to use and may find bugs by using unexpected combinations of events. On the contrary, they can reveal only specific tool-supported error events (e.g., crashes, timeouts, permission errors). Tools using this technique are Android Monkey[6] and GUIdancer[7].

## 2.9 Solutions Based on Image Recognition

This category of solutions is often being referred to as Visual GUI Testing. It is an emerging technique combining scripting languages with image recognition. The image recognition allows us to test various systems regardless of their implementation, operating systems, or even platforms. Tools are providing support for emulating user interaction with the bitmap components (images, buttons) shown to a user on the screen. The biggest limitation of solutions based on image recognition is that they are not suitable for highly animated GUIs [4]. There is also a considerable amount of work required for test maintenance, mostly caused by design changes of widgets throughout the development.

There are several examples of tools that use image recognition for testing, including open-source tools Xpresser[8] and Sikuli[9]. Xpresser is a python module that works with a directory of images containing cropped images of widgets. Once the image matching algorithm identifies a location of a cropped image on the screen, an intended action can be performed on the given coordinates [11]. Xpresser is mostly used for building automated test cases for the Linux distribution Ubuntu.

## 2.10 Model-Based Testing

Model-based testing (MBT) is a software testing technique where test cases are generated from a model that describes functional aspects of the SUT. It allows one to check the conformity between the implementation and the model of the SUT, with a more systematic and automatic approach in the testing process. The test generation phase is based on an algorithm that traverses the model and produces test cases suitable for automatic execution [26].

### 2.10.1 Existing Solutions

The TEMA toolset is an MBT framework developed for smartphone applications. Testers have to manually create a two-tier model consisting of two state machines, called the action and keyword machines. Those machines represent the GUI at design and implementation levels. The method generates design-level test cases by traversing the action machine. Afterward the keyword machine is used to transform design test cases into executable ones [12].

[6] https://developer.android.com/studio/test/monkey
[7] https://testing.bredex.de/
[8] https://wiki.ubuntu.com/Xpresser
[9] http://www.sikulix.com/#home1

Another approach was introduced in the GUITAR [14] framework for automated GUI testing. GUITAR can be divided into the following steps:

1. GUI reverse engineering,
2. automated test case generation,
3. automated execution of test cases,
4. support for platform-specific customization,
5. support for addition of new algorithms as plugins,
6. support for integration into other test harnesses and quality assurance workflows.

The first step contains a reverse engineering process. A structural GUI model of an application under test is extracted from the run-time state of the application. This process involves automatic execution of an application, where the tool called Ripper is used to discover as much as possible about the application. The application's window and widgets are discovered in a depth-first manner. The Ripper extracts properties of widgets such as position, color, size, and enabled status, followed by information about events and results of event execution. The depth-first traversal terminates when all GUI windows are covered. The problem with this heuristic is that it would hypothetically contain an infinite number of ways to interact with non-trivial GUI applications. At the end of the process, Ripper stores the extracted structural information about the GUI to a data structure called GUI Tree, in an XML format.

To complete the reverse engineering process, the tool called Graph Converter provides a platform-independent framework to convert the GUI Tree model into a graph, representing relationships between events in the GUI of the application. The result is an Event-flow Graph (EFG) used for test case generation. An EFG is a directed graph representing all possible event interactions on a GUI. Each node represents a GUI event. An edge from a node $v$ to a node $w$ represents a `follows` relationship between $v$ and $w$, indicating that the event $w$ can be performed immediately after the event $v$. An EFG is analogous to a control-flow graph, in which vertices represent program statements and edges represent execution flows between the statements.

In the third step, test cases are automatically generated based on the EFG. Therefore, the GUI test generation problem is reduced to a problem of graph traversal, thus any graph traversal algorithm can be used for test generation.

# Chapter 3

# AT-SPI Architecture

An accessibility is a technology that helps people with disabilities to participate in essential life activities. An accessibility as a part of the GNOME desktop includes libraries and development tools allowing users with disabilities to use other options of interaction with the GNOME desktop environment. Those options include voice interfaces, screen readers, and other alternative input devices [3].

Assistive technologies are receiving information from the Accessibility toolkit (ATK), which offers built-in APIs for all GNOME widgets. ATK provides a set of interfaces that are required to be implemented by GUI components. Therefore, assistive technologies can automatically read most of the labels on screen without any extra efforts made by developers. The interfaces are toolkit-independent, meaning that their implementation could be written for many widgets, including widgets from frameworks such as GTK3[1] and Qt[2].

## 3.1 GNOME Accessibility Implementation Library (GAIL)

Nowadays, the majority of GNOME applications are written in the GTK3 framework. The framework provides a dynamically loadable module named GAIL that implements the ATK interfaces for all GTK3 widgets. Once the module is loaded at runtime, the application is fully capable to cooperate with ATK without any further modifications. The GNOME desktop does not load accessibility support libraries by default. They have to be enabled by setting a special `gsettings` key, which can be achieved either by the `dconf`[3] editor or via the `gsettings` command-line utility using a terminal application (Listing 3.1).

```
gsettings set org.gnome.desktop.interface toolkit-accessibility true
```

Listing 3.1: Enabling accessibility via a `gsettings` command

Additional configurations may be required for applications written in other frameworks such as QT or Java. Compared to the AT-SPI, implementations of other assistive technologies might be too application-specific or use various techniques like OS event snooping, etc. In the GNOME Desktop, all information required by assistive technologies (AT) is passed from the GNOME Accessibility Framework to a toolkit-independent Service Provider In-

[1] https://www.gtk.org/
[2] https://www.qt.io/
[3] https://wiki.gnome.org/Projects/dconf

terface (SPI). The SPI is a key component for providing a stable and consistent API for screen readers, magnifiers, etc. The accessibility support is relying on a per-toolkit implementation (GTK3, QT, Java) and its APIs exported through relevant bridges to unified AT-SPI interface as described in Figure 3.1.

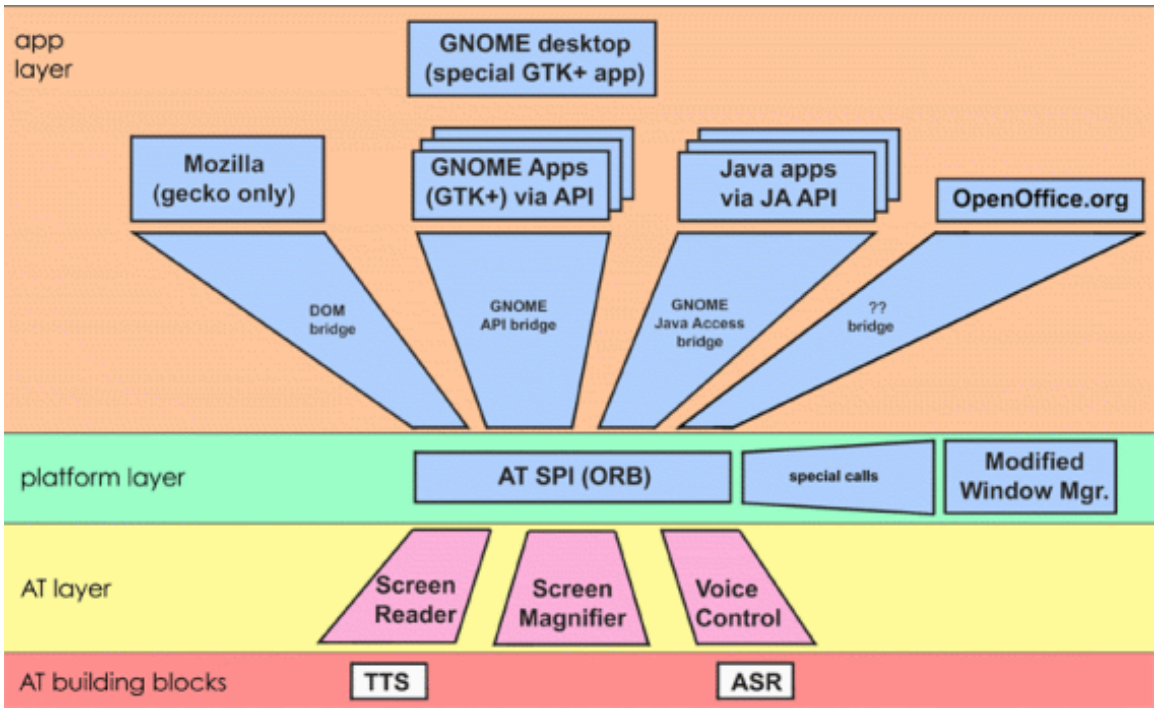


Figure 3.1: GNOME Accessibility Architecture overview[3]

A widget is accessible, if a developer uses any GTK3/GNOME widget and follows the general accessibility guidelines[4] with properly implemented ATK interfaces. A developer can also create a custom widget. A custom widget is accessible when its implementation is based on one of the stock GTK3/GNOME toolkit widgets. The default implementation of the ATK interfaces might be altered by applications. Therefore, a developer can enrich descriptions of widgets and improve the overall user experience in special cases, e.g. when a widget is used for some less expected purposes or the default description is too general. The ATK provides a set of functions to achieve this along with the ability to make any custom component accessible[5] [9].

## 3.2 Libraries and Tools

Currently, there are several tools available for exploration and debugging accessibility features not only on the GNOME desktop.

[4] https://developer.gnome.org/accessibility-devel-guide/stable/gad-coding-guidelines.html.en

[5] https://developer.gnome.org/accessibility-devel-guide/stable/gad-custom.html.en

### 3.2.1 Library Pyatspi

The package pyatspi is a Python wrapper around the AT-SPI's C implementation, which loads the Accessibility typelib and imports the classes implementing AT-SPI interfaces [19].

AT-SPI exposes applications as a tree of widgets that are also accessible in Python through pyatspi. On the top, the root element represents the whole GNOME desktop. Every sub-element represents one running application on the GNOME desktop. Each application has zero or more child elements, each child is distinguishable by its position in the tree and several object properties. Some of these properties are encapsulated inside the accessible object and their values must be obtained through corresponding methods, so-called *getters*.

The library pyatspi is an open-source project available for most Linux distributions via distro specific packaging services (package named `python3-atspi`) or is available to be built from its sources[6].

### 3.2.2 Dogtail

Dogtail is an open-source GUI test framework written in Python and implemented as a library around the pyatspi. Several modules implement a higher level of API to simplify work and interaction with the accessible objects during test development. Dogtail utilizes attributes provided by pyatspi that are required for testing. A small set of attributes is described in the following list:

- *name* – a string value, for most widgets contains a text identical with the text label visible on the widget,
- *role* – a string value, specifies the widget type,
- *childCount* – an integer value, represents a number of sub-elements,
- *actions* – a dictionary that contains available actions which can be performed on a widget by the ATK,
- *visible* – a boolean value, indicates that a widget is visible to the user,
- *showing* – a boolean value, a widget is rendered,
- *text* – a string value, mostly used in input fields or widgets containing plenty of text,
- *description* – a string value, contains a special widget description for users,
- *position* – an integer tuple, x, y coordinates on the screen (might be related to other, component)
- *size* – an integer tuple, shows the height and width of the widget.

Additionally, the elements can be linked together in other useful ways (except the parent-child relationship), where the input widgets (e.g.: text field, check box, combo box, etc.) are linked with the elements that serve as their labels. These labels are making the input widgets easier to identify or interact with. Other advantageous element properties e.g. *showing* or *visible* are used to decide whether the element is hidden from the active screen

[6] https://gitlab.gnome.org/GNOME/pyatspi2

area, thus it is not available for interaction. The *roleName* attribute allows a categorization of widgets that is useful for identification of category-specific methods, e.g. selecting a radio button value, selecting an option in combo boxes, or a click method performed on push buttons.

The library contains methods that can generate user input events. The implementation is focused in a module named *rawinput* that provides methods for generating mouse or keyboard events.

The package dogtail also includes a GUI tool *Sniff* (*AT-SPI Browser* in Figure 3.2), similar to the *Accerciser* application described in the next section. The tool offers less complex functionality, containing a tree view of accessible objects with their basic attributes [7].

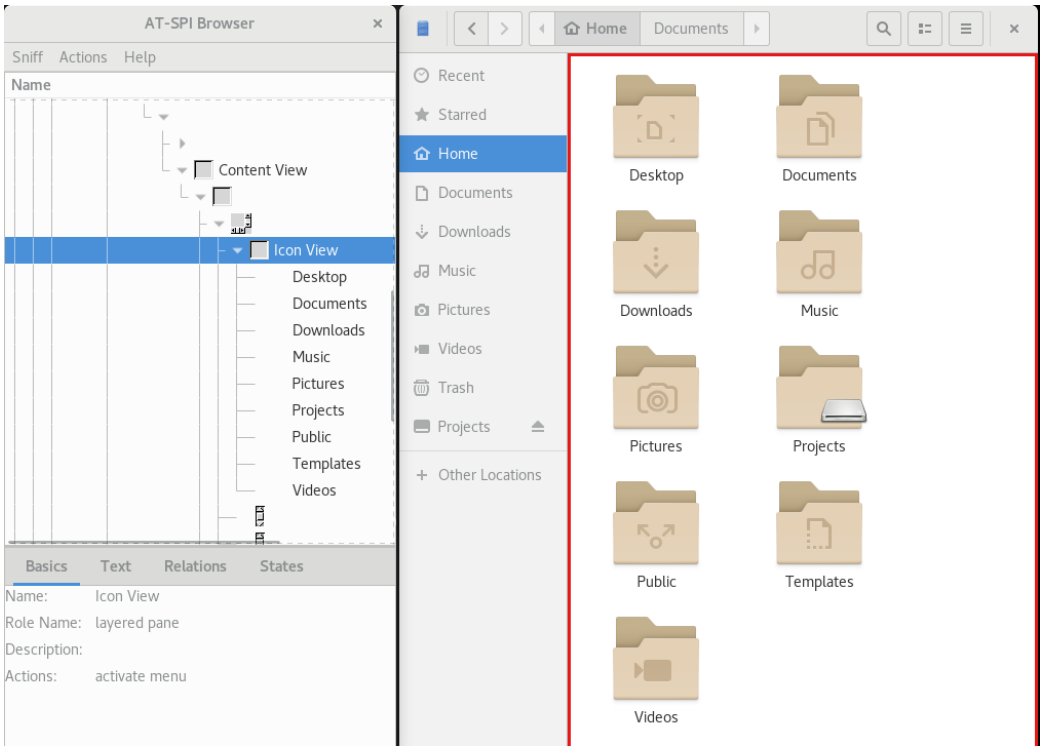


Figure 3.2: The *Sniff* utility (*AT-SPI Browser*), highlighting the *Icon View* area in the *Nautilus* file manager

The module `tree` contains the most important class `Node`, instances of the class represent elements of the desktop user interface. All elements are gathered to the tree structure, representing all applications starting with the root element (desktop). The class is implemented as a mixin for Accessible and various Accessible interfaces and is an important unit for its subclasses, namely `Application`, `Root` and `Window`. The `Node` class also implements methods used for search of nodes in the tree based on certain criteria. A lambda expression can be passed to methods `findChild` and `findChildren` as an argument named `pred`. The lambda expression can contain any properties that uniquely identify nodes, including `name`, `roleName`, `showing` and `visible`. The class also contains action methods that can be performed on nodes without importing other action modules. Verification and identification of shown nodes is easier thanks to the method named `blink`. Once the method is called on a certain element, the element is highlighted on the screen for several seconds. This

functionality is also part of the Sniff tool where an element is highlighted after it is selected in the displayed tree.

The module `dump` contains only one method with the same name. The method returns a string describing the tree of nodes which is useful for python/ipython console debugging.

Finally, the module `rawinput` contains the implementation required for generating events from both a keyboard and a mouse. More complex events simulating keyboard shortcuts, mouse gestures, and drag and drop operations are implemented as well.

Testing dogtail has proven its availability for many Linux distributions through their package repositories, specifically Fedora 32, Red Hat Enterprise Linux 8.2 and Manjaro 18 with GNOME 3.34 (Archlinux). It is also available as a *Pypi* Python package and according to information in it's official Gitlab repository, it should work not only for GTK3 applications but also for applications written in QT and KDE.

During the testing of dogtail, we revealed some minor problems which might occur with test development. There are known cases in which the coordinates of a node were not reported correctly. Most of the elements with the `roleName` value *panel* and *list box* are missing their `name` values. Elements without the `name` value are much harder to identify, although they might not be important for users, as they do not contain any visible text, nor do they offer a way of interaction. The purpose of those elements is to serve as a wrapper that groups other elements together in a tree.

However, there are elements that are available for user interaction but they are not named (e.g. a refresh button in the Disk User Analyzer application[7]). Once an action needs to be dispatched on such an element, the identification has to be done either through a parent element or a sibling element. Additionally, the execution of a mouse event will require an offset calculation to specify the correct element position on the screen.

Another discovered issue is a non-accessible menu which is included in the majority of the GNOME applications. This issue is quite severe, therefore it was reported[8] and resolved by developers.

So to conclude this subsection, dogtail is a powerful tool for the development of automated test cases in the GNOME3 environment. On the other hand, it contains discussed limitations and flaws. Those limitations do not need to come from dogtail itself, they are either accessibility bugs or bugs in the GTK3 framework (non-accessible menu).

### 3.2.3 Accerciser

Accerciser is an interactive accessibility explorer developed in Python. It provides a well-arranged graphical frontend for the AT-SPI library, hence it can inspect, examine and interact with widgets. It also serves as a verification tool for developers, to check that their applications are providing correct information to assistive technologies and automated testing frameworks. Compared to Sniff, Accerciser's interface (Figure 3.3) offers extended features and functions. The default interface has three sections, a tree view with the entire hierarchy of accessible objects and two optional plugin areas. The Accerciser has an extensible, plugin-based architecture. Most of the features available by default are provided by plugins discussed in the next several paragraphs.

The *Interface Viewer* plugin is an explorer of the AT-SPI interfaces provided by each accessible widget of a target application. When a tree element is selected, its interfaces are shown with a list of sensitive methods. The majority of methods are executable. The list

[7] https://wiki.gnome.org/Apps/DiskUsageAnalyzer
[8] https://bugzilla.redhat.com/1723836

contains methods for interaction with an object and various methods for obtaining more information about the object. Accerciser offers an exploration of the following interfaces:

- *Accessible* – shows the number of child widgets, description, states, relations, and other attributes,
- *Application* – if implemented (not mandatory), it shows the application ID, toolkit and version,
- *Component* – shows the element's absolute position with respect to the desktop coordinate system, the relative position with respect to the window coordinate system, size, layer type, MDI-Z-order indicating the stacking order of the component and alpha,
- *Document* – shows document attributes and locale information,
- *Hypertext* – shows a list with all element's hypertext links, including name, URI, start index and end index,
- *Image* – shows the element's description, size, position and locale,
- *Selection* – shows all selectable child items of the selected item,
- *Streamable Content* – shows the selected element's content type and its corresponding URIs,
- *Table* – shows the element's caption, rows, columns, number of selected rows, number of selected columns, and for the selected cell, it shows it's row's and column's header extents,
- *Text* – shows the selected element's text content, that can be editable with attributes including the offset, justification and possibility to show CSS formatting as well,
- *Value* – shows the element's value, minimum value, maximum value, minimal increment for a value.

The *AT-SPI Validator* plugin applies tests to verify the availability of accessibility for a target application. The validator will generate a report of the selected item and all its descendant widgets in the tree hierarchy.

The next plugin is the *Event Monitor*, which displays AT-SPI emitted events including a filter for several different AT-SPI event classes. The plugin has the ability to monitor only events originating from the selected application or a selected accessible (widget). Each event record contains the source and the application.

The *Quick Select* plugin provides global hotkeys for quickly selecting accessible widgets in the Accerciser's Application Tree View, the selected widget is highlighted in the target application.

The *API Browser* plugin shows interfaces, methods and attributes available on each accessible widgets of a target application. By default, it shows only public methods and properties. Private methods and properties are hidden until the checkbox *Hide Private Attributes* is unchecked.

Finally, the plugin *IPython Console* provides a full, interactive Python shell. The console has an immediate access to any selected accessible widgets of a target application. The currently selected object in the tree view is available in the IPython Console under the symbol *acc*. The plugin provides an easy way to test and debug code used in test cases.

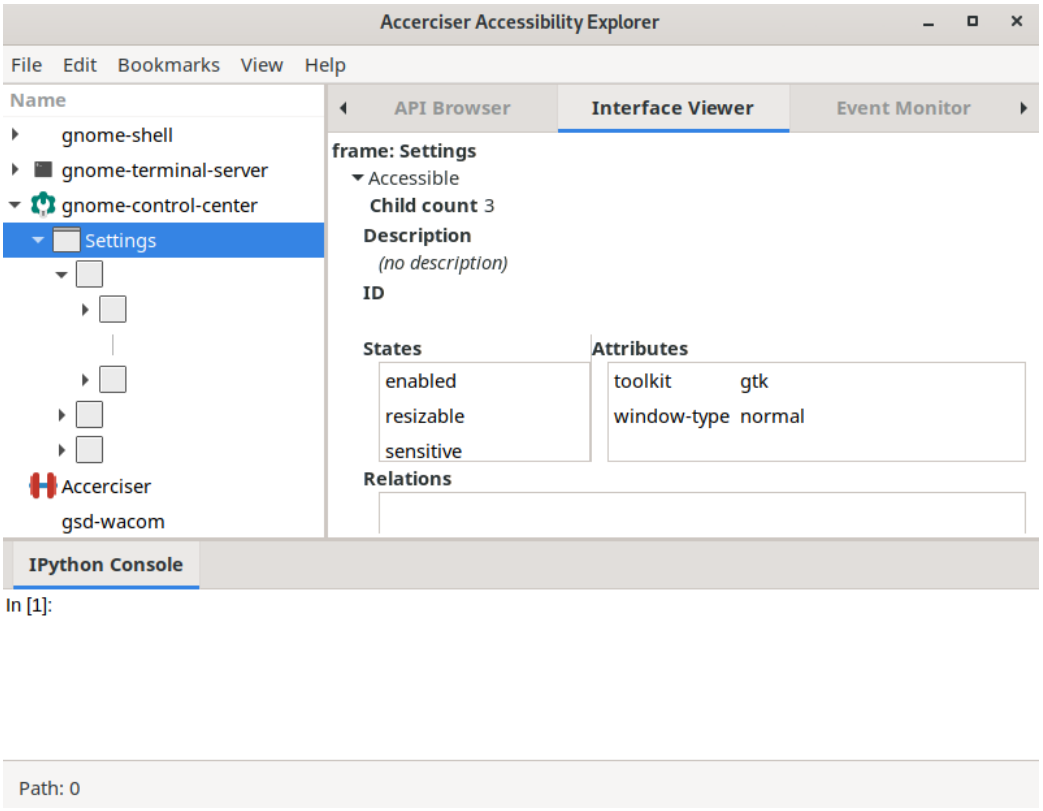


Figure 3.3: Accerciser's default configuration

## 3.3 Covering Limitations of Accessibility and Verification

As discussed in the aforementioned sections, the information provided by the AT-SPI is not flawless. Therefore, the next couple of sections is dedicated to an exploration of technologies that might be used to support the accessibility in such cases.

### 3.3.1 OpenCV and Image Matching Techniques

OpenCV or Open Source Computer Vision Library is a software library that provides optimized algorithms for computer vision and machine learning. According to the official OpenCV webpage [18], the library contains more than 2,500 algorithms and it is being developed by a vast community of contributors around the world. The library is used extensively by government institutions, research groups, and companies including Microsoft, Google, IBM, etc. One of the biggest advantages is its native C++ implementation with bindings making the library available in Python, Java, and Matlab, and the fact that supports Linux, Android, Mac OSX, and Windows. Similarly to dogtail, OpenCV can be installed easily via the Python3 package manager (`pip`), regardless of the Linux distribution.

OpenCV offers many algorithms, including image recognition that can be used to either locate or verify the presence of an element on the screen. This approach would require to have a set of images containing elements prepared in advance, then it can be used to find the image location on the screenshot taken during a test run. Compared to verification of the node via the AT-SPI only, this approach would also verify that the element is properly rendered on the screen. An additional benefit is a possibility of verification of text format-

ting and colors. On the contrary, this process requires additional manual work where one would need to capture images, label them, and associate them with certain test scenarios. The number of elements displayed on the screen multiple times creates another parameter that would require manual maintenance. The most common example of such cases are buttons labeled either *OK* or *Cancel* as they are used in many applications.

Another possible approach is to use the shape recognition algorithm which can locate shapes like circles, rectangles, and many other common shapes. From the development perspective this would be easier to maintain, as there is no requirement for images prepared in advance. Frequent application changes during the software development may also cause that tests based on image matching can be easily outdated. This factor forces testers to revisit test suites, therefore the efficiency of automated tests deteriorates. It can also help with the widget location in cases where accessibility is reporting wrong coordinates. On the other hand, locating the right widget in cases when several similarly shaped ones are located on the screen at the same time will yield very inconsistent results.

### 3.3.2 Optical Character Recognition

The Optical Character Recognition (OCR) is a method of extracting text from images. One of the available open-source tools is a tool called Tesseract.

Initially, Tesseract development started in 1985 at Hewlett Packard Laboratories but the major breakthrough was achieved in 2006 when the project was open-sourced in cooperation with the University of Nevada in Las Vegas. Since then, the project has been developed under the sponsorship of Google [24].

Usability of Tesseract was increased in version 3.x, supporting a wide range of image formats and gaining the ability to be used in a larger number of scripting languages. While Tesseract 3.x is based on traditional computer vision algorithms; in the past few years, methods based on Deep Learning have surpassed traditional machine learning techniques by a vast margin, especially in terms of accuracy in several areas of Computer Vision. Remarkable results were achieved in handwriting recognition. Tesseract has implemented a recognition engine based on Long Short Term Memory (LTSM) which is a kind of Recurrent Neural Network (RNN). While this kind of RNN is used to recognize texts of random length, a Convolutional Neural Network is used just for recognition of a single character. Version 4 provides both a legacy OCR engine and a new LSTM engine which is enabled by default [8].

Tesseract can be used as a command-line tool, and its integration into software being developed is possible via the Tesseract's API available in Python3 or C++. Setting Tesseract up on Linux or other platforms may differ, but the process is accurately described in the Tesseract's wiki[9], with the last resort solution – building it from its sources. The setup process includes installation of the `tesseract-ocr` package itself, `pytesseract` Python3 bindings installable via the package manager `pip`, and the Tesseract's language pack with trained data for the English language (version 4.x supports 130 languages[10]).

The Tesseract's OCR engine works best when used with images containing black text on a white background in a common font. The text should be approximately horizontal with the height of at least 20 pixels. With possibilities of image processing provided by OpenCV, the image quality in some cases needs to be improved before applying text detec-

[9] https://github.com/tesseract-ocr/tesseract/wiki

[10] https://github.com/tesseract-ocr/tesseract/wiki/Data-Files#data-files-for-version-400-november-29-2016

tion methods. The most common image preprocessing methods include inverting images, rescaling, binarisation, noise removal, rotation, border removal, and page segmentation[11].

The Tesseract's API for Python3 is bundled in a module named `pytesseract`. The module provides several methods, the most important ones for the purpose of this work being `image_to_string` and `image_to_data`. Both methods have one compulsory parameter which is an image intended for text extraction. The image has to be in a certain format, one of the options is to load the image through OpenCV's `imread` method. Additional parameters may be applied including the language, timeout, and engine configuration[12]. The `image_to_string` method returns all recognized strings including all whitespaces and other special characters. The `image_to_data` method provides additional metadata about all recognized strings in a form of dictionary-like object. The returned dictionary contains the following lists of properties:

- text – string value, may contain a string, special character, one word or line of text,
- left – integer value, specifies the number of pixels from the left side of the image,
- top – integer value, specifies the number of pixels from the top of the image,
- width – integer value, specifies the width of the recognized string,
- height – integer value, specifies the height of the recognized string,
- the rest are less important values for this work: `level, page_num, block_num par_num, line_num, word_num, conf`.

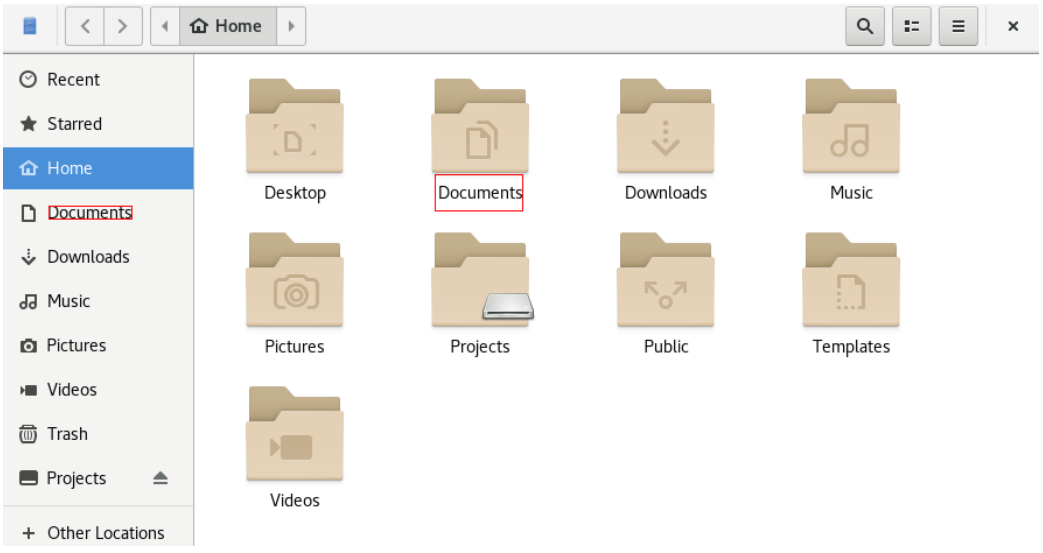


Figure 3.4: Demonstration of the OCR engine detection for the string Documents in the Nautilus File Manager window

Figure 3.4 contains a demonstration of the Tesseract engine capabilities. The task was to locate the string *Documents* in the screenshot of the GNOME file manager application

[11] https://github.com/tesseract-ocr/tesseract/wiki/ImproveQuality
[12] https://pypi.org/project/pytesseract/

*Nautilus*. The engine successfully found both strings located in the image and provided coordinates and dimensions that were used by OpenCV to highlight the strings in the image.

The demonstration has proven that the Tesseract's OCR engine can be used as an alternative tool for location or verification of widgets that contain text. This method also verifies that the text content was properly rendered and is readable for the user. OCR systems have limitations and work with a certain margin of error which is a fact that also applies to Tesseract. Various applications can use different color schemes including background colors and font colors, input fields, and labels. Highlighting elements to perform actions on them can also lead to changes in color conditions. Image preprocessing methods provided by OpenCV can aid in avoiding problems associated with those cases, namely color inversion and binarisation. Those methods would supply the Tesseract's engine with an image containing black text and a white background for the evaluation.

## 3.4 Conclusion

This chapter has been dedicated to the accessibility technologies in the GNOME desktop with a deeper look at implementation, libraries, and tools for debugging. Furthermore, technologies that may be able to cover limitations and bugs in accessibility have been evaluated as well. Both OpenCV and Tesseract may help with identification, location, and verification of non-accessible elements in applications. A possible disadvantage is a delay caused by taking and processing screenshots of applications that have to be taken at the right time. The OpenCV's image matching algorithm can reliably locate prearranged images of icons, labels, or whole application windows on the screen. Considering a stable application environment with a black text on a white background in most applications, Tesseract can detect and reliably locate most of the text content on the screen. Other cases can be covered by image preprocessing done again in OpenCV. Both technologies are working with the actual application content rendered to users, possibly bringing an additional level of verification. However, the goal of this work is to generate test cases dynamically and preparation of a set of screenshots to verify a proper rendering of icons would violate this effort. A solution for such a situation, would be to take screenshots during the test generation process. However, an icon would need to be cropped out from the screenshot, thus relying on the position of the icon reported by the AT-SPI. Therefore, an integration of the Tesseract's OCR is more beneficial for this project.

# Chapter 4

# Design of the Proposed Test Generator

In this chapter, we present the design of our test generator. Section 4.1 explains reasons behind the implementation of a custom representation of an application model. In Section 4.2, we discuss a setup and a configuration of the test environment required for the tested applications. Section 4.3 describes the structure of the generated test cases. Then in Section 4.4, we describe the algorithm for test case generation. Finally, Section 4.5 addresses the integration of the Tesseract's OCR engine in our test generator.

A goal of this work is to develop a tool capable of generating automated test cases for GUI applications. The proposed test generator works with AT-SPI metadata provided by applications and converts them to test cases. The required metadata should be available for a lot of applications, assuming they are developed in one of the common frameworks (GTK3, QT). However, this tool is focused on testing of GUI applications developed for the GNOME desktop[1]. A tool is developed in Python3, version 3.6.

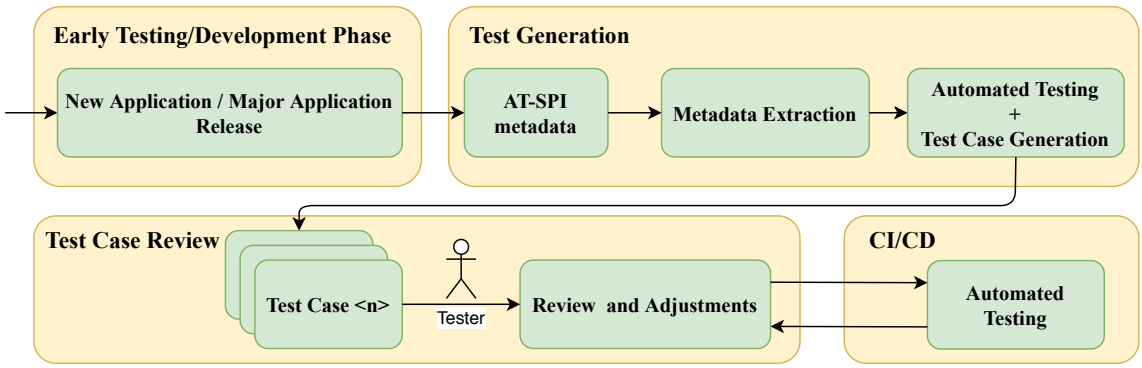


Figure 4.1: A workflow overview when testing with the proposed test generator

The GNOME applications are open source, and they are being developed by community of enthusiasts around the GNOME project. Anyone from the community can fix bugs in applications by sending a merge request with fixes, request a new feature, or propose newly developed features. This development model does not contain a planning phase or a phase where one can design an abstract model of an application, from which the test cases could be derived. Therefore, our solution is based on deriving the model of a GUI application

[1] https://wiki.gnome.org/Apps

from the AT-SPI metadata. The extracted data about widgets and relationships between them provide the foundation which gives the test generator the ability to interact with an application. Therefore, the test generator can use the extracted information to perform exploratory testing. The testing includes execution of scenarios (so-called *event sequences*), monitoring the behavior of the SUT, and detection of certain errors and crashes of the SUT. Test cases are generated as a by-product of this process.

Figure 4.1 describes an overview of the workflow when testing with the proposed test generator. The beginning of the scenario starts with either a newly developed application or a version of an application that contains major changes. The tool performs an initial exploratory testing of the application based on the extracted model and exports those scenarios in the form of *behave* test scenarios.

From the testing perspective, the proposed tool combines several testing techniques. The extracted AT-SPI metadata create a foundation for a simplified model of the application by partially adapting the model-based testing technique. Since the test cases are derived from the model without any knowledge about the implementation of the tested application, the tool resides in the category of black-box testing. The knowledge about the SUT provided by the model allows the tool to benefit from the approaches described in the random input testing and random walk tools in a more deterministic way. The tool can be characterized as a semi-smart tool as it can detect certain crashes of the SUT during the test generation and immediately report them with a reproducer. The results of the test generation process are test cases that may be adjusted and executed again. The tests are executable in the CI/CD pipeline that can be triggered at any stage of the application development and reports the results without manual retesting.

## 4.1 Model Extraction

In this section, we justify and present the implementation of a custom accessibility tree that serves as a model of a tested application for our test generator.

**Custom Model Justification** The model extraction process relies on the AT-SPI metadata that is provided after the start of an application. As mentioned in Section 3.2.1, once the application is running, a tree of widgets is exposed and available for interaction. The provided representation of the tree itself is not suitable to be directly used as a model of an application because the implementation contains several restrictions for the purposes of this work.

The first restriction stems from nodes/widgets with no functionality nor a way of interaction for the user, e.g.: filler, separator, panel, etc. Theoretically, a copy of the tree could be created with those nodes filtered out, although in that case the parent-child relationship in the tree needs to be restored accordingly. However, this is not possible, since the attributes `children` and `parent` in `Atspi.Accessible` object instances are read-only.

The accessibility tree also contains references to properties and methods which are available only during the application runtime. If the application crashes or it is terminated, the aforementioned methods and properties can not be accessed. Furthermore, the test generator must start the execution of every test scenario (event sequence) from the default state which is achieved by obtaining a fresh instance of a tested application.

Additionally, the custom implementation of the model allows us to track the progress of the test generation process. The model consists of objects gathered in event sequences, each

object has its unique identity that lasts throughout the test generation process. This implementation allows us to measure the event coverage and ensures that an already executed event sequence is never repeated.

**Model Implementation** A solution for previously mentioned restrictions is a custom implementation of the accessibility tree that also serves as a model of applications for our test generator. Test case generation requires that a custom tree is derived from the accessibility tree provided by the *dogtail*. The unusable nodes are filtered out, while the parent-child relationships of the nodes are preserved. The custom tree can be used as a model, that will map the possibilities of interactions available for users working with an application. The model includes every node from the original tree with available actions that are also executable by the AT-SPI.

Our development revealed that not all rendered widgets are properly labeled with actions by AT-SPI. The affected nodes are the ones with role names `page tab` and `list item`. The former has to be clicked to gain access to additional nodes, the latter can be placed on the same level as a push button. Therefore, our generator relies on records in the file `roleNames.py` where the role names of actionless nodes are enumerated. If a node is not associated with any action, the default action for the node is `click`.

However, the model does not allow to execute the associated actions directly, as the generation process requires one to run several instances of an application. The instances of accessible objects and some of their properties are valid only for one application runtime. Therefore, several important values are extracted in the process, making them available even after the termination of an application instance. The properties `name, roleName, parent_name, parent_roleName` are used as the unique identifier because some nodes might share the same `name` and `roleName` (e.g. OK, push button). The properties give the test generator the ability to match each node from the model to the current application instance exposed by the accessibility layer. The implementation of the model is presented in the class diagram in Figure 4.2.

Starting from the lowest level, an instance of the class `GNode` represents one node from the tree. Several attributes are copied from the original `dogtail.tree.Node` instance, including attributes storing the pieces of information about the parent node, the data describing the state of the node, the list of children, and if available, the name of the action method. The list of children is also composed of instances of the `GNode` class, so the tree is recreated recursively. Therefore, the model can hold all information about tested applications, without relying on their state. An instance of the `GTree` can represent either a whole application or a smaller part of the application e.g.: a dialog or a menu. As discussed previously, this offline model of the application tree also contains a lot of nodes without the ability of interaction, which needs to be filtered out. Those nodes are identified by the list of `RoleNames` that are gathered in the separate file `rolenames.py`. Finally, the class `TestTree` serves as a wrapper that filters those nodes and preserves the parent-child relationship. The result of this process is an instance of the `TestTree` object and it contains only nodes required to generate test cases.

## 4.2 Test Environment

This section describes conditions that need to be achieved in the GNOME environment for our test generator. Subsection 4.2.1 addresses phenomenons that can occur in the GNOME

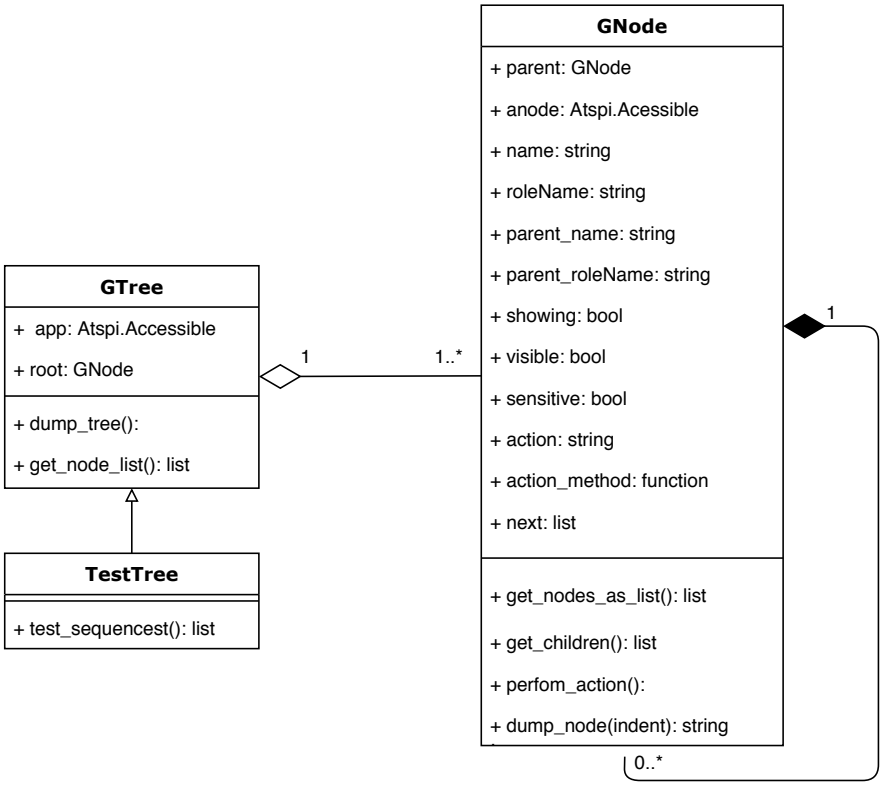


Figure 4.2: The class diagram of the custom application tree that serves as an application model for our test generator

Shell environment and need to be suppressed during the test generation. It also introduces the *qecore* library designed to handle this kind of issues. Subsection 4.2.2 describes a method and parameters used to configure our test generator for various applications. Subsection 4.2.3 offers a brief introduction to *Flatpak* applications, then explains why our test generator needs to acquire support to test such applications as well as contributions to the *qecore* library that were submitted and approved. Subsection 4.2.4 concludes this section with a description of how applications are executed and monitored during the test generation.

### 4.2.1 Test Environment Setup

Our test generator is designed to test GUI applications that are developed to work in the GNOME Shell environment. The environment contains various features[2] like workspaces, notifications, the application grid, the activities overview and menus. Some of these features may change a state of the environment, and therefore negatively affect tested applications during the test generation process. Execution of an action that brings the environment to some of those states steals the focus from the tested application back to GNOME Shell, thus blocking any further interaction. A notification might collide with the user interface of the tested application and blocks the execution of an action during the test case. These factors need to be avoided to ensure stability during the test generation and the test execution

[2] https://help.gnome.org/users/gnome-help/stable/shell-introduction.html.en

as well. The setup must also be able to recover the environment from potential test case failures and will not influence the execution of the subsequent test cases.

The required setup for the test execution is implemented in the module *qecore*. The module is designed for test automation of GNOME desktop applications and contains various measures designed to avoid occurrences of unintentional environment events and focus on a tested application. The module is bound to *dogtail* and it is intended to be used with *behave* framework [17]. The module is actively developed by quality engineers from Red Hat.

The test generation process relies on the environment setup provided by the *qecore* module. However, the module is designed for the test execution and our test generator utilizes different approaches for execution and monitoring of tested applications. For this reason, we developed a custom subclass `App` based on the *qecore's* `Application` class. The relationship between classes is shown in Figure 4.3.

During the development of our test generator, we also contributed to the implementation of the *qecore* module. We introduced a `desktop_file_path` property for the `Application` class to solve the problems with location of `.desktop` files required to test LibreOffice applications. We also submitted a couple of smaller fixes[3,4]. The mentioned proposals were approved and merged.

```
1  libreoffice-startcenter:
2    a11y_app_name: soffice
3    app_process_name: soffice.bin
4    desktop_file_path: /usr/share/applications/libreoffice-startcenter.desktop
5    kill_command: "pkill soffice"
6    params: "--norestore" # required to avoid unwanted file restore dialogs
7    cleanup_cmds:
8      - "pkill soffice" # LO required a custom kill cmd
9      - "rm -rf .config/libreoffice/*"
10   packages:
11     - libreoffice
12   flatpak: False
```

Listing 4.1: An example of the apps.yaml entry for LibreOffice StartCenter

### 4.2.2 Test Generator Configuration

Assurance of compatibility with various applications across the GNOME ecosystem requires that some metadata describing the tested application has to be provided before the test generation process.

The metadata is gathered in a configuration file written in the YAML[5] language. The reasons behind choosing YAML is syntax simplicity and human readability in comparison with JSON[6] or XML[7], followed by the reliable support in Python provided by the library *pyyaml* [22].

[3] https://gitlab.com/dogtail/qecore/-/merge_requests/24
[4] https://gitlab.com/dogtail/qecore/-/merge_requests/26
[5] https://yaml.org/
[6] https://www.json.org/json-en.html
[7] https://www.w3.org/XML/

Our generator uses the configuration file `apps.yaml` where we store records about all tested applications. An example for an application record can be seen in Listing 4.1.

An application record starts with a name of application on the top level. The application name should be unique, as the name is used as a folder name of the generated project. The metadata are stored as values with keys. A large group of keys matches the names of `App/Application` class properties. Some of the items are not necessary for the test generation, although they are required for the test execution. Required keys/values may vary per tested application. The list of keys/values that can be defined for each application includes:

- `a11y_app_name` – This is the only compulsory item. It defines a name of the application in the accessibility tree. The value can be found in GUI tools Sniff or Accerciser as previously discussed in Section 3.2.2. The value can match with the name of the application.
- `app_process_name` – The value is required if the name of the application process differs from the application name. The value is used during the cleanup in between the executions to make sure that an instance of the application has been killed and a next test will use a new one.
- `desktop_file_path` – This is required if default *qecore's* method fails to find the desktop file of an application. The desktop file contains useful data about applications, including a command required to run an application from the command line.
- `params` – The value is required if the application needs to be run with custom command line parameters. All parameters should be entered in one string, separated with a space. This also allow us to run an application with a test file.
- `cleanup_cmds` – The value may contain a list of commands that will be executed after the generation of each test case. Executed commands should always restore an application to its default settings. The commands are used during the execution of generated test cases, at the end of every test case.
- `packages` – The value is required for execution in the CI environment, contains a list of rpm[8] packages required to be installed to both generate and execute tests.
- `flatpak` – The key/value is required if a tested application is a flatpak.

### 4.2.3 Flatpak Applications Setup

This subsection is dedicated to a brief introduction to Flatpak applications, followed by explanation why Flatpaks needed to be integrated in our test generator. Further, we discuss an effort that has been done to support Flatpak applications in our test generator.

**Flatpak** Desktop applications on Linux are being distributed through various distribution-specific package managers. Flatpak[9] is a technology for building and distributing desktop applications on Linux that aims to solve the problem with a cross-platform distribution of

[8] https://rpm.org/
[9] https://flatpak.org/


Figure 4.3: A class diagram providing an overview over the implemented test generator

packages on Linux. Applications, or so-called flatpaks, are delivered to users regardless of the lifecycle of the underlying Linux distribution. The system implements a set of sandboxing technologies, to isolate Flatpaks from each other and the system, thus providing security benefits to users [2].

The majority of GNOME applications are also available through flatpak. A dedicated flatpak repository *Nightly GNOME Apps* contains the latest development versions of GNOME applications. With flatpak, those applications are installed alongside their stable versions. This gives us the potential to test the application much sooner before it is released to distributions. This is a benefit behind the integration of flatpak support to this

work. The main repository for flatpak applications called Flathub[10] contains hundreds of applications developed in various frameworks and programming languages. However, the effort done by this work only supports applications developed in GTK3 as they obtain the accessibility support by design.

**Flatpak Integration** There are several differences in the execution process between flatpaks and non-flatpak applications. Every flatpak application has a unique name, e.g.: `org.gnome.gedit`. A unique name is required for every operation executable through the `flatpak` command-line utility. The utility not only serves as a package manager able to install, remove, downgrade and update flatpaks, it also provides a sandbox to run flatpaks. Those differences demand certain changes in the runtime used for the test execution and the test generation.

Considering the test execution, the approach used in the *qecore's* `Application` class should be suitable for testing flatpaks. However, the initial testing emphasized the previously mentioned differences, and therefore we developed the subclass `Flatpak` that inherits the methods from the `Application` class and reimplements some of them do address those differences. The most important changes are:

- `__init__` – the constructor performs a validity check on inserted flatpak ID, the format requires two dots, e.g. `org.gnome.gedit`,
- `start_via_command` – runs a flatpak via only via command and with the flatpak command-line utility, e.g. `flatpak run <id>`,
- `kill_application` – terminates a flatpak via command e.g. `flatpak kill <id>`,
- `get_desktop_file_path` – performs a recursive search for flatpak's `.desktop` file in two possible locations:
  - `~/.local/share/flatpak/app/` – flatpak installed per-user
  - `/var/lib/flatpak/app/` – flatpak installed system-wide
- `is_running` – performs a check if a flatpak is running, this is again done with the flatpak command-line utility (e.g. `flatpak ps <id>`) and the presence of an instance in the accessibility tree

Additionally, the invocation of some of the inherited methods does not make sense for flatpak applications. The invocation of those methods with an instance of the `Flatpak` class raises an exception. The exception contains a message with an explanation that the methods are not available for Flatpak objects. We proposed the developed module `Flatpak.py` to the *qecore* project, the module was accepted[11].

As discussed in Section 4.2.1, the *qecore* library is designed to handle the test execution. Additionally, there are other requirements to handle Flatpaks during the test generation. Therefore, we developed the subclass `FlatpakApp` that works on the test generator level to fullfil those requirements. As described in the class diagram shown in Figure 4.3, the `FlatpakApp` serves as a wrapper for the `Flatpak` class in the same manner as the `App` class wraps the `Application` class. `FlatpakApp` and `App` are customized classes for the test generator, while `Flatpak` and `Application` are used during the text execution.

[10] https://flathub.org/home
[11] https://gitlab.com/dogtail/qecore/-/blob/master/qecore/flatpak.py

### 4.2.4 Execution and Monitoring of an Application

This subsection describes the method used by our test generator to execute and monitor tested applications. The implementation for the non-flatpak applications is encapsulated in the class `App`. The class `FlatpakApp` achieves the same goals for the flatpak applications.

**Execution** The test generation process performs various actions available in an application that might change settings or layout of the application. Therefore, every test case must start from the same state which should satisfy the following conditions:

1. an application is not running, if so, force the application to stop;
2. reset the applications' settings to the default state by performing a predefined custom cleanup;
3. start a new instance of the application with the default settings;
4. make sure that application is ready for an interaction.

**Monitoring** Several indicators can be monitored while the application is being tested. The most essential one is to be able to safely determine if the application is running at the moment or not. This can be done either by examination of the *pid* (process id) belonging to the application process or by relying on AT-SPI. If the application tree is not available, it can be certainly assumed that the application instance is not running. This statement also applies vice versa, so an assertion that an application has started is achievable in the same way. The implementation takes advantage of *dogtail's* `Tree.Node.Applications()` call, returning a list of applications currently exposed to the accessibility bus.

Furthermore, it is also necessary to perform certain checks during the time an application is being interacted with. Therefore, every tested application will be run as a sub-process, which enables us to capture the output generated by tested applications to standard streams (*stdout, stderr*). Once an application has been terminated, it also allows us to check the return codes. The implementation relies on the Python's standard library `subprocess`.

The output generated to the standard stream is checked for errors defined in the designated configuration file. In case of error throughout the generation process, an error message is printed immediately to warn about the possible bug in tested applications. The warning contains the number identifying the test in which the error occurred, a full error message, and a return code. All other captured messages, e.g., warnings or deprecation messages from the GTK framework are saved to one log file, in a folder where the tests are generated. The messages are being appended, so the log file can be checked at any time during the generation process. Every line contains the test number, so it can be easily determined when the message occurred and match it with the reproducer from the given test case.

## 4.3 Generating an Environment for the Test Execution

This section describes an output of our test generator along with description of individual files that are required for execution of generated test cases. The input of the test generator is provided by the application metadata located in `apps.yaml`. The output is a project structure containing generated test cases and other files required for the test execution.

Initially, the test generator checks the availability of the entry for a tested application in the configuration file `apps.yaml`. Subsequently, it creates a sub-folder with the name of the application where generated content will be placed. Predefined source files with the implementation of *steps* (used by *behave* framework) are copied to the folder structure along with scripts and other files that are necessary for the test execution. Figure 4.4 demonstrates the structure generated for the application *GNOME Terminal*.

```
gnome-terminal
├── features
│   ├── generated.feature
│   ├── environment.py
│   └── steps
│       ├── ocr_steps.py
│       └── steps.py
├── gnome-terminal.log
├── mapper.yaml
├── requirements.txt
├── runtesh.sh
└── cleanup.sh
```

Figure 4.4: The generated project structure for the application *GNOME Terminal*

The sub-folder named `features` contains files with the *behave* test cases. Test cases generated by the test generator are in the file `generated.feature`. The file contains a single so-called *Feature* that contains all generated test cases. Test cases are composed of a tag, a brief description of the test case, and so-called *steps*. The tag is a unique identifier of the test case and thus allows single test case execution, if required. The description should briefly define what should be done with the SUT, when the test case is executed. The steps are one-line statements, each of them describes either an execution of an action or an assertion described in a human-readable language. Successful execution of all steps evaluates the test case as passed. Otherwise, the result of the test case is a fail.

The file `environment.py` contains the setup required for the test execution (see Appendix E). It contains 3 functions used by the *behave* framework to set up or restore the required environment during the test execution.

The `before_all` function is run once before the execution of the test cases. It initiates the GNOME environment setup from the *qecore* library and creates an instance of either the `Flatpak` or the `Application` class. The type of the application and the parameters for the class instance are extracted from the entry in the configuration file `apps.yaml`.

The `before_scenario` function is executed before every test case (scenario). It contains an invocation of the method from the *qecore* that should set the testing environment to the default state and other preparations for the testing. Additionally, it executes the `cleanup.sh` script with a custom per-application cleanup defined in `apps.yaml` (discussed in Section 4.2.2).

The `after_scenario` function is called after the execution of every test case, regardless of its results. The result is then submitted to the generated test report.

The folder `steps` contains source files with the implementation of the *steps* used in the *behave* scenarios. The implementation of steps is divided into two files. The module `ocr_steps.py` contains only one *behave* step which encapsulates the implementation and optimization used for the verification of the string on the screen. The module `steps.py`

contains general implementation of steps. The steps are functions implemented in Python with the `step` decorator from the *behave* framework. The decorators serve as a wrapper to call the Python functions from the `.feature` files. So in the case of this project, the steps written in the test cases are function calls, the functions are defined in these modules. The definition of the `@step` decorator (Listing 4.2) contains variables, thus allowing us to keep the code base as minimal as possible.

```
1 @step('State: "{roleName}" "{name}" "{prop}" is "{state}"')
2 def assert_state(ctx, name, roleName, prop, state):
3     node =ctx.app.instance.child(name, roleName)
4     focus_node(node)
5     assert hasattr(node, prop), f'Obj: {node} is missing attribute {prop}'
6     prop_value =f'{getattr(node, prop)}'
7     assert state ==prop_value, f'Expected: {state}, Got: {prop_value}'
```

Listing 4.2: The implementation of the step that is used to perform an assertion on any of the properties belonging to an accessible node

The file `gnome-terminal.log` aggregates log messages produced by a tested application throughout the test generation process. The log file name is derived from the application name defined in the `apps.yaml` file. The `mapper.yaml` file contains a list of test cases with other data required for the CI execution. The file `requirements.py` contains all Python dependencies that need to be installed to execute the test cases. Finally, the `runtest.sh` is a wrapper script for execution of test cases.

## 4.4 Test Case Generation

In this section, we describe the test generation process. The Subsection 4.4.1 describes the extraction of event sequences from the model of an application. Subsection 4.4.2 describes the execution of the extracted event sequences. In Subsection 4.4.3, we discuss the node expansion process performed by the test generator during the test generation.

The implementation of the test generator is encapsulated in the class `TestGen` (class diagram in Figure 4.3). The behavior of the generator can be also influenced by several command-line arguments that will be described later in this work. Based on the parameters (`flatpak` item in `apps.yaml`), the instance of either the `App` or the `FlatpakApp` is created.

The generator then creates a copy of the default project structure and injects the files inside the project structure with values that correspond to the application that is going to be tested (Figure 4.4). Namely, the files `environment.py`, `mapper.yaml`, and `cleanup.sh` contain special placeholders (tags) that are replaced by values defined in `apps.yaml`. Just note that no Python code is being generated during the process. The default project already contains all the predefined *behave* steps required to execute generated test cases. The next sections are dedicated to the details of the generation algorithm. Algorithm 4.1 contains a shorter version written in a pseudocode.

### 4.4.1 Derivation of Event Sequences

This subsection describes how our test generator extracts event sequences and explains lines 1–4 from Algorithm 4.1.

The generator begins with a first start of a tested application and extracts the AT-SPI tree of the application instance through the `dogtail`. Then, a writable copy of the tree is

**Algorithm 4.1:** Test generation algorithm pseudocode

```
Data: Running application exposed to the accessibility bus, apps.yaml
Result: Test Cases
start the application;
scan the application tree, generate the test tree;
derive the event sequences;
terminate the application;
foreach event_sequence in event_sequences do
    application cleanup, if required;
    start the application;
    foreach action in event_sequence do
        save the state before the action is executed;
        execute the action;
        add the action step to the test case;
        if application is not running then
            check return code and the logs;
            if application crashed then
                print reproducer and log;
            else
                add the quit assertion to the test case;
            end
        else
            evaluate the tree changes through the symmetric difference;
            if action started new application then
                generate the assertion;
            else if action generated new window/s then
                foreach window in windows do
                    append new event sequences for the window;
                end
            else
                append new event sequences for the remaining nodes;
            end
        end
    end
end
```

created through the `GTree` instance. The action-less nodes are then removed by creating a new instance of the `TestTree` class.

**Event Sequence** The core of our test cases is derived from the `TestTree` through the class method named `test_sequences`. The method returns a list[12] of event sequences. An event sequence contains a list of nodes associated with actions that will be executed for every test case. Listing 4.4.1 shows an output produced by the `print_sequences` method used for debugging. For each `GNode` instance, we print a `name`, a `roleName` and an action separated by `=>`.

[12] https://docs.python.org/3/tutorial/datastructures.html

```
1 LibreOffice:frame: => File:menu:click => New:menu:click => Text
    Document:menu item:click
```

Listing 4.3: An example of the event sequence extracted from LibreOffice StartCenter.

The `test_sequences` method iterates through the list of leaves and calculates the path from a leaf to the root of the tree. The result is a list of paths or as mentioned earlier a list of event sequences.

An event sequence does not represent a whole test case. The whole test case is created by applying the event sequence on the live instance of the application. While applying the event sequence, the generator appends assertions steps and OCR checks. The checks are generated either before or after the execution of actions and they will be used as a verification in a generated test case to confirm that the application reached the intended state. An event sequence can be used multiple times or it can be extended, if the generator discovers that the applied sequence led to a discovery of new nodes. Those new nodes are evaluated and the generator creates new event sequences, each of those sequences start with a sequence that led to their discovery.

### 4.4.2 Execution of Event Sequences

In this subsection, we describe how we used the extracted event sequence to create a test case. From this point the generator works with 3 instances of the tree: the currently running application instance obtained through *dogtail*, the instance of the class `Gtree`, the instance of the class `TestTree` or so-called model used to derive the tests.

The test generator then starts to iterate over the extracted event sequences, monitors the application, executes actions, and generates steps and assertions that are then put together in *behave* scenarios. The implementation of the steps used in generated test cases is discussed in Section 4.2.

Every iteration of the event sequences (see line 5 in Algorithm 4.1) works with a newly started instance of the application, so every scenario begins with a step that starts the application. The step internally contains an assertion to make sure that the application has started and is ready for the interaction. The generator stores a shallow copy of the list containing the applications that are currently available through AT-SPI. It also saves a copy of the currently available nodes in the tested application.

The generator selects the first node from the sequence and locates the node within the currently running instance of the application. In case that application contains too many nodes (widgets), some of them might be hidden. The generator tries to avoid that by using `grabFocus` method on the node. The method does not work for menus, where the `select` method has to be used instead. Additionally, the `node.sensitive` property is checked. If the value of the property is `False`, the generator prints a warning as the value indicates that the action might not be executable in the current state. Then the action associated with the node is executed. If the execution of the action was successful, the event coverage is increased and a step with the description of the node and executed action is added to the test scenario. The execution of the action is followed by several checks performed on the current instance.

Initially, the generator checks whether the application is still running by retrieving the application instance from the accessibility tree. If the application instance is no longer present, there are two possibilities. The application was intentionally terminated by the executed action or the application crashed. The decision is made by the examination of

the generated logs (*stderr, stdout*) and the value of the return code retrieved after the termination.

If the application was terminated with the return code value of 0, the test generator appends a new step to the test case. The step contains an assertion that the application is no longer running (Quit/Exit button).

If the return code does not contain the value 0, the generator raises the error, prints the reproducer, the return code value, and the content of the obtained logs. The generator proceeds to the next test case.

In some cases, the occurrence of an error does not mean that the application crashes. Therefore, the generator checks the log for known errors after every executed action regardless of the state of the application. The list of the messages that are being checked is stored in the file `errors.py`. A new error message can be appended to the list at any time. The list currently contains messages that previously occurred in bugs related to *GNOME* applications.

### 4.4.3 Model Expansion

This subsection provides a description of the model expansion during the test generation process. The description contains an explanation how our test generator scans an application for new nodes (widgets) as well as creation of test cases for these nodes. We also explain the remaining part of the Algorithm 4.1 that starts from line 20.

Successful execution of the action, followed by no errors detected in the log and application still being run, indicates that the action could have changed the state of the application.

Initially, the generator checks whether the action triggered the execution of a new application. The detection is achieved through the symmetrical difference computed on two sets. The first one contains the list nodes representing running applications before the action was executed, the second one holds the list of applications available after the execution. Both lists are shallow copies, so the generator does not need to compare an entire tree for each application. If that is a case, the generator appends the assertion implying that the applied sequence led to the start of a new application. The test generator does not expand the nodes of a newly spawned application to the current test tree as they do not belong to the application that is currently being tested. This solution has limitations, an application that is not exposed to the accessibility bus will not be detected.

If the previously described effort failed, the generator proceeds to search for the changes within the tree of the tested application. The implementation takes advantage of the method `get_node_list` from the class `GTree`. The method returns all nodes from the tree instance in one *list*. The list is converted to a set, and similarly to the process of detection of a new application, it calculates the symmetric difference between sets captured before and after the executed action.

The generator distinguishes between several roles of the discovered nodes. The appearance of a new window or a dialog causes the generation of an assertion to the current test case. Regardless of a role, the generator creates a `TestTree` instance with new nodes (a subtree) and retrieves event sequences derived from the subtree. The new event sequences are prepended with the sequence that led to their discovery and then added to the list of the event sequences that will be executed in the next iterations.

The expansion during the test case generation can significantly increase the execution time. The test generator implements an option `--shallow` that disables the expansion and

generates test cases only from the model obtained after the start of the application. The option gives testers the ability to obtain fundamental test cases that can be reviewed and updated in a shorter time.

## 4.5 OCR Integration

In this section, we present the integration of Tesseract's OCR engine in our work. Subsection 4.5.1 is dedicated to the implementation of image preprocessing methods in our test generator. Subsection 4.5.2 described integration of the OCR in executable test cases.

The main goal of the OCR integration in this work is to provide an additional level of verification of string values presented by applications and thus not rely purely on AT-SPI. However, the integration of OCR into the generated test cases has to be reliable to avoid false-positive test results. For the reasons mentioned in Section 3.4, the implementation has to contain image preprocessing optimizations and configuration to achieve stable results. Tesseract offers several options that allow to optimize string detection and text analysis. One of them is the definition of the recognized language. It is assumed that most of the tested applications will use the English language and therefore, the dataset trained for the English language is used.

### 4.5.1 Screenshot Preprocessing and Optimizations

In this subsection, we discuss the implemented optimizations that were required to achieve reliable results with the Tesseract's OCR engine in our test generator.

As discussed in 3.3.2, Tesseract is less prone to errors when operating with images containing black text on a white background. Therefore, we used a thresholding method to convert screenshots to binary colors (black and white). However, some applications use darker color themes or contain parts with different color schemes. If we used the thresholding method with such application it would provide us with an opposite result. Therefore, our solution always extracts strings from two images. The first one is a binarized copy of the original image, the second one is a copy of the binarized image with inverted colors. This ensures that the Tesseract's OCR engine has the best possible conditions to obtain the string from the screen. Given that the string is present on the screen, it should be found regardless of a theme set in an application. A demonstration of the image conversions

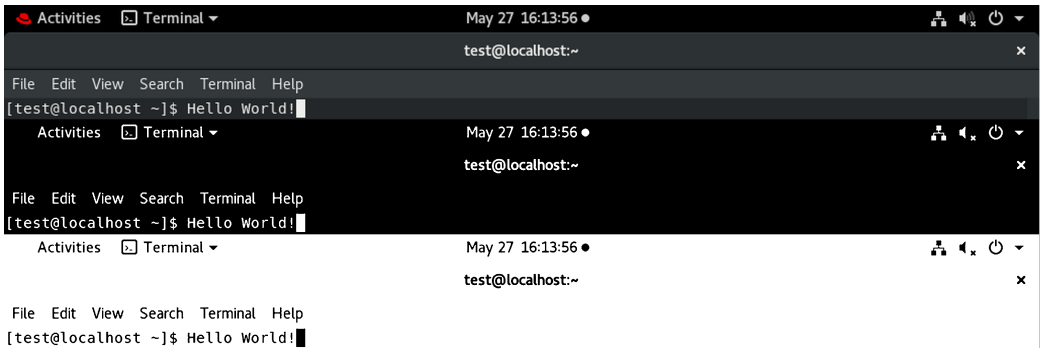


Figure 4.5: Steps of image preprocessing for the OCR, from the top: the original image, the binarized image, the inverted binarized image

is shown in Figure 4.5, containing 3 images, ordered from the top: the original image, the binarized image, and the inverted binarized image.

```
1 s
2 ALY
3 O BTEE T
4 ERM
5 test@localhost:
6 8
7 Edit View Search Terminal Help
8 [test@localhost ~1$ Hello world![]
```

Listing 4.4: Text extracted from the original image in Figure 4.5 without optimizations

Listing 4.4 demonstrates the results obtained from the original image without any optimizations. The Tesseract's OCR engine manages to extract certain strings from the screen, although the results are not reliable and thus may lead to false-positive reports during the test execution. Further experiments have shown additional issues with text formatting as well as difficulties with recognition of similarly looking letters. An example can be seen on line 8 (see Listing 4.4), where character `]` was misinterpreted as character `1`. These issues were suppressed by upscaling the resolution of the original image from 1,024 x 768 pixels to 3,200 x 2,400 pixels.

```
1 Activities
2 Terminal ~
3 a4
4 v
5 May 27 16:13:56 @
6 test@localhost:~
7 File Edit View Search Terminal Help
8 [test@localhost ~]$ Hello World![}
```

Listing 4.5: Text extracted from Figure 4.5 with all implemented optimizations

Listing 4.5 shows results achieved with image preprocessing methods. When compared to the results shown on Listing 4.4, it proves an increase of the efficiency achieved with the implemented optimization. The result contains all important strings shown on the screen. Additionally, the OCR engine reports some random characters which are probably caused by a misinterpretation of a group of smaller icons in the picture. The image upscaling is achieved through the *Pillow*[13] library, the image conversions are implemented through methods from the *OpenCV* library.

The time consumed by taking screenshots during the generation process is significant. Therefore, the developed tool has the ability to disable the generation of the OCR steps during the generation of test cases through the command line parameter `--disable-OCR`. If the generated test cases already contain steps performing OCR checks and are intended to be executed without them, the tests can be executed with the shell variable `OCR=False`. The defined variable will cause skipping of the OCR checks, although they will still be shown in the test logs as executed. This is caused by the limitation of the behave framework as

[13] https://pypi.org/project/Pillow/

```
1  @1_Spreadsheet
2  Scenario: libreoffice-startcenter: Spreadsheet
3    * Start: "libreoffice-startcenter" via command "libreoffice --norestore"
         in session
4    * Action: "click" "File" "menu"
5    * Action: "click" "New" "menu"
6    * State: "menu item" "Spreadsheet" "showing" is "True"
7    * OCR: "Spreadsheet" is shown on the screen
8    * Action: "click" "Spreadsheet" "menu item"
9    * State: "frame" "Untitled 1 - LibreOffice Calc" is shown
10   * OCR: "Untitled 1 - LibreOffice Calc" is shown on the screen
```

Listing 4.6: A test case demonstrating the integration of OCR into test cases

it only allows one to skip whole test scenarios. Tests executed with the variable set to skip the OCR steps will contain a warning message.

### 4.5.2 Implemented Steps

This subsection demonstrates a *behave* step that was implemented to achieve the OCR integration in our tests.

The results obtained from experiments with the OCR were implemented to a single *behave* step. The step contains a string variable that should be found on the screen at a specific moment during the test execution. The process involves taking a screenshot via the `gnome-screenshot` utility. It continues with the aforementioned image preprocessing and extraction of the text from two variants of images. Finally, an assertion is made to confirm the presence of the string on the screen.

Listing 4.6 contains a test case generated for *LibreOffice StartCenter* with two OCR steps. During the test, the OCR engine confirms the presence of the string `Spreadsheet` on line 7. Another OCR step is on line 10, where the OCR confirms a presence of the window title `Untitled 1 - LibreOffice Calc` on the screen.

OCR steps are not added to test cases automatically. Our test generator may encounter strings that contain various characters and have various length. An OCR step is generated to a test case only if the test generator performs a successful OCR check on a given string. Failed OCR checks are reported immediately during the test generation process. The OCR check performed in advance by the test generator and the optimizations discussed in Subsection 4.5.1 should prevent an occurence of false-positive results in test cases caused by the OCR.

## 4.6 Generated Test Cases

In this section, we describe the *behave* test cases that are generated by our test generator as well as logs that are generated during the test runs.

The result of the generation process is available in a folder structure named after a test application and contains generated test cases, configuration files, and scripts for execution in the CI environment. Generated *behave* test scenarios are located in the file named `generated.feature`. The file contains all the test cases divided into so-called scenarios.

Each scenario (see Listing 4.6) has a unique name starting with character `@` that allows single test execution if required. All tests are executable by issuing a command `behave` in the generated project folder. The tests are also respecting the cleanup commands which are set in `apps.yaml`. The cleanup is always executed after the finish of the test, regardless of the result of the executed tests. Execution of `behave` either prints steps from a test scenario to standard output or can generate an HTML log for every scenario (see Figure 4.6). This log format is more suitable for examination of the results executed in the CI environment accessible through the web interface. Furthermore, the *qecore* library embeds data collected during the test runs to the *behave* reports of failed test cases. Therefore, the reports contains videos captured during the test runs, screenshots taken in a moment when a test case failed and additional logs. The additional data helps with identification of potential flaws in tests and false-positive results.

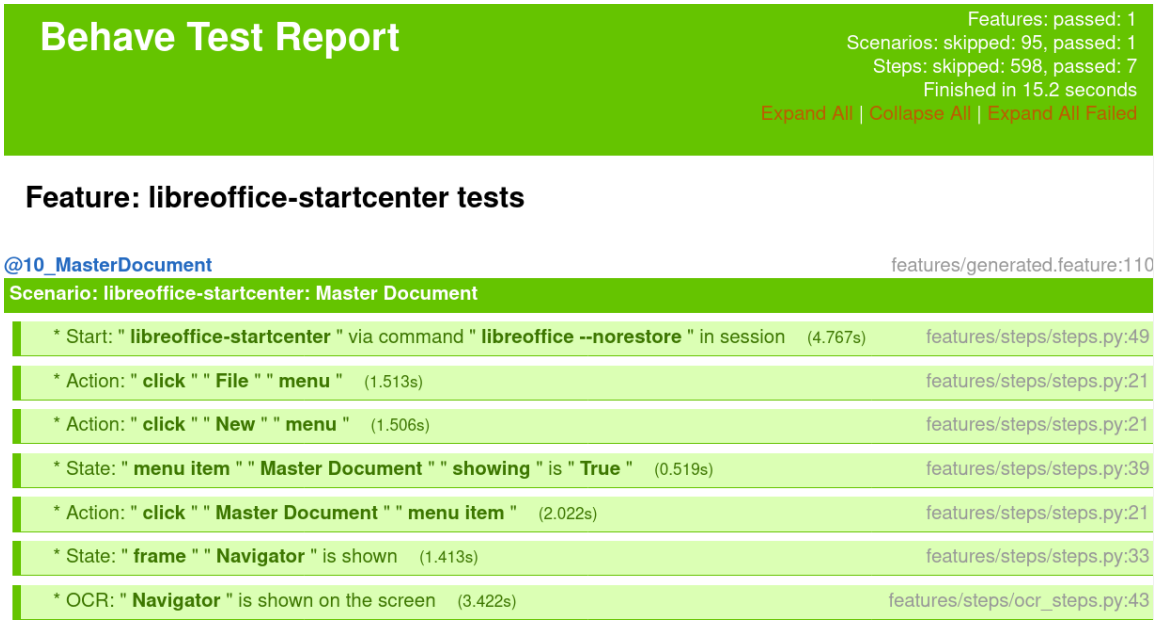


Figure 4.6: An example of the test report generated by the *behave* framework during the execution of the tests generated for LibreOffice StartCenter

# Chapter 5

# Testing and Results

In this chapter, we discuss the results of our test generator. Section 5.1 describes the coverage measurement techniques we obtained for the test cases created by our test generator. In Sections 5.2 – 5.6 we present 5 open-source GUI applications as well as the results that were achieved when we tested them with our test generator.

All performed testing was done on virtual machines preloaded with distributions *Red Hat Enterprise Linux 8.2/8.3*. Virtual machines were assigned with 4 GiB of RAM and two logical CPU cores (*Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz*). The execution of the test generator on production workstations should be avoided as the performed actions may potentially lead to alteration of the system or a data loss. The same approach applies to the generated test cases.

## 5.1 Coverage Evaluation

This section is dedicated to the coverage measuring techniques that were implemented within our test generator, namely a model coverage and an event coverage. Further, we discuss the code coverage analysis performed with the `gcov` tool.

**Event Coverage** The event coverage measures the number of executed events on a tested application during the test generation. It is a common technique used in GUI testing [14]. An execution of an event is counted as successful one, if the event was executed by the accessibility layer without any errors. The event coverage approach implemented by our test generator also has a disadvantage. The event coverage report contains only events reported by the accessibility technology, other events like drag and drop, keyboard shortcuts and mouse scrolling are not included.

**Model Coverage** Another coverage measurement performed by our test generator is the model coverage. The coverage is measured as the number of nodes (widgets) involved in test cases from the overall number of nodes included in the model.

As discussed in Section 4.1, the model contains only nodes that offer an action that can be executed by users. It is expected that this coverage will always cover 100 % of the nodes. However, with applications that contain a richer GUI, some of the nodes might be hidden or the generator will not be able to derive an event sequence that will be able to access those nodes. This especially applies to cases when the generator will be used on a new application that contains some special layout or an action on the given node which

could not be executed by the accessibility technology. In such cases, our generator skips the whole test case, generates an error message, and proceeds to the next test case.

Nodes that are not covered by tests, along with event sequences that are involved, are printed in a report after the test generator finishes. The nodes or event sequences reported as failed must be evaluated manually with several possible outcomes:

1. a node is not available for interaction
2. a bug in a tested application,
3. a bug in the accessibility technology (e.g. incorrectly reported coordinates),
4. an imperfection or a bug in the test generator,
5. the tested application is affected by the previous test case (change in the settings/layout), and additional cleanup must be added to `apps.yaml`.

Both the event coverage and the model coverage are evaluated in the end of the test generation process. The measurements are a part of a final test generator report. The report also contains a list of unexecuted event sequences. The list can be used to retest test cases that were not sucessfully tested by our test generator.

**Code Coverage** We also obtained code coverage measurements from tests runs generated by our test generator. The code coverage measurements were obtained with the `gcov` tool that comes as a part of the GNU development tools. The purpose of the tool is to perform the code coverage analysis and to find dead or unexecuted code. Coverage-driven testing can be characterized by the following steps:

1. Find the areas of a program not exercised by test suite.
2. Create additional test cases to exercise so far not exercised code, thereby increasing code coverage.
3. Determine a quantitative measure of code coverage, which is an indirect measure of quality.

To obtain a measurement of the code coverage, it is required to compile the application with `gcc/g++` and two extra parameters `-fprofile-arcs` and `-ftestcoverage`. Running the compiled binary with the `gcov` tool yields a percentage of the executed code located in source files. Measurements can be obtained for any software written in C/C++ [6].

To obtain measurements, tests must be executed with a custom binary, compiled with the mentioned parameters. Once the custom binary is executed, files with extensions `.gcda` and `.gcno` should appear in the directory where the binary is located. Measurements are aggregated throughout the test execution and the code coverage is reported to the special files with the mentioned extensions. Then, the `lcov` tool is used to aggregate the measurements and generate a report in two steps (Listing 5.1). The first command takes the `.gcda` and `.gcno` files and generates a `.info` file with coverage information. The second command takes the `info` file and generates a detailed HTML report. The report contains every source file (`.c` file) along with the percentage of covered functions and lines.

```
$ lcov -c -d . -o app.info
$ genhtml -o lcov_report -s --legend app.info --ignore-errors
```

Listing 5.1: Shell commands used to generate an HTML report with the `lcov` tool

## 5.2 GNOME Terminal

GNOME Terminal (Figure 5.1) is one of the most important applications from the GNOME application stack. The application serves as a terminal emulator for accessing a UNIX shell environment. The application can be used to run programs available on the system[1].

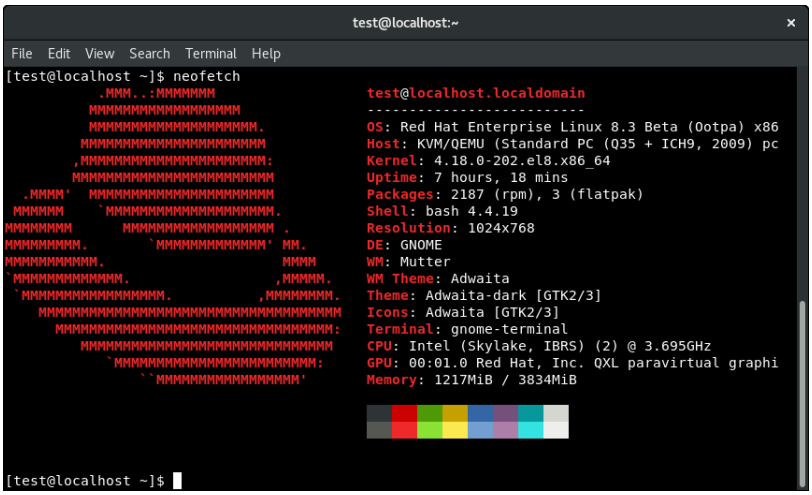


Figure 5.1: GNOME Terminal application UI

### Setup and Cleanup

A large part of test cases generated for GNOME Terminal performs some changes of settings either via the *Preferences* dialog or through menus located at the top of the window. Preferences can change various aspects of the application, including the text encoding, layout of widgets, and color schemes. Those changes need to be set back to default values to make sure that test cases will not affect subsequent test cases. In *Terminal*, this is achieved through 2 cleanup commands (see Listing 5.2) that are executed at the end of every test case. Testing was performed with the rpm package `gnome-terminal-3.28.3-1.el8.x86_64`.

```
dconf reset /org/gtk/Settings/Debug/enable-inspector-keybinding
dconf reset -f /org/gnome/terminal/legacy/
```

Listing 5.2: The cleanup commands required to reset GNOME Terminal to its default settings

[1] https://help.gnome.org/users/gnome-terminal/stable/introduction.html.en

### Test Generation and Results

Listing 5.3 contains the final test generator report and summarizes the testing performed on *Terminal*. The developed test generator was able to generate 485 test cases while covering 1,700 events in the application. Tests are covering menus, several smaller dialogs, and a *Preferences* window.

```
1    Test Generator Report for Component: Terminal
2    Covered Events: 1700/1702 (99.88 %)
3    Number of Covered Nodes: 516
4    Number of Generated Test Cases: 485
5    Nodes without the Coverage:
6        Edit:menu:click => Preferences:menu item:click => :list item:
7            => Menu:toggle button:click
8    No errors found!
9    Generation Time: 1:53:52.676497s
```

Listing 5.3: Final test generator report for GNOME Terminal

## 5.3 GNOME Help

GNOME Help (Yelp) is a help viewer for GNOME[2]. The application natively renders documents in various formats including HTML documents. The UI (see Figure 5.2) of the application is filled with links to navigate between documents.

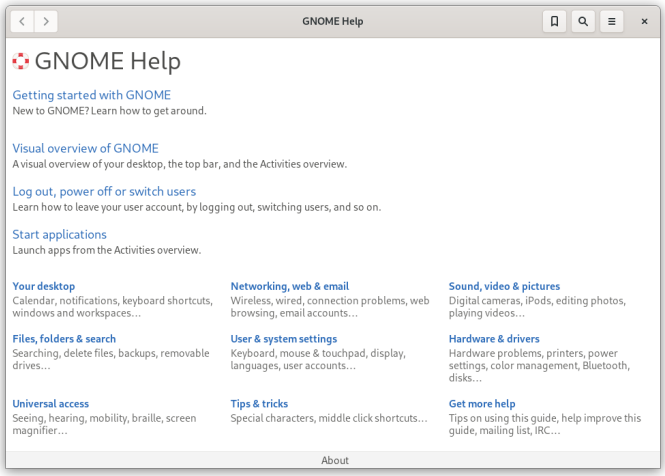


Figure 5.2: GNOME Help (Yelp) application UI

### Test Generation and Results

Listing 5.4 contains the final report and summarizes the testing performed on *GNOME Help*. The developed test generator was able to generate 2,412 test cases while covering the

[2] https://wiki.gnome.org/Apps/Yelp

4,690 events in the application. The application did not require any individual setup, the testing was performed with the rpm package `yelp-3.28.1-3.el8.x86_64`.

```
1    Test Generator Report for Component: Help
2    Covered Events: 4690/4696 (99.87%)
3    Number of Covered Nodes: 2415
4    Number of Generated Test Cases: 2412
5    Nodes without the Coverage:
6    GNOME Help:frame: => Digital cameras:link:jump
7        => More Information:heading:Click
8    ...
9    Generation Time: 4:57:13.370368s
```

Listing 5.4: The final test generator report for GNOME Help

The report also contains 6 failed event sequences. The reason behind these failures is partially the way we implemented our test generator. As discussed in Section 4.1, we introduced the file `rolenames.py` where we enumerated all role names for nodes without an action. The mentioned failures contain node with the role name `heading`. Since the majority of headings are assigned with an action called `jump`, the role name `heading` is not blacklisted in the file `rolenames.py`. However, there are nodes with the role name `heading` that are assigned with an empty action (an empty string). This is how the accessibility system labels links in Help that are not available for interaction. Our test generator handles a node with an empty action by replacing the empty action with a default one – Click. Since the default action is not the one assigned to a node, its execution might fail which is what most probably happened in the failed event sequences.

The second part of the report is dedicated to captured error messages. Overall, there were 42 warnings about occurrences of error messages during the test generation along with their reproducers. The examination has revealed that error messages were found in test cases where an external link leads to a web page. External links are not handled by the Help application, they are forwarded to a default web browser for the GNOME session. In our case, the default web browser was Firefox. A manual application of reported reproducers has shown, that the captured error messages were not originated from the Help process, but they were generated by the Firefox process. Initially, we were not able to reproduce the issue because the messages did not appear until the Firefox window was closed. To support the claim that the error messages are caused by the bug in Firefox, we successfully reproduced the same issue with the LibreOffice StartCenter, where our test generator found the same error messages in a different test scenario. Our findings were submitted in a bug report[3].

*GNOME HELP* also offered a good demonstration of how nodes are expanded during the test generation. To visualize the node expansion performed by our test generator during the testing, we integrated an automatic generation of event flow graphs implemented by the `networkx` library written in Python3. An initial graph is generated after the scan of the SUT performed in the beginning of testing. A final graph is generated at the end of the test generation process. All graphs that were generated during the testing with our generator are available in Appendix F.

[3] https://bugzilla.redhat.com/show_bug.cgi?id=1837978

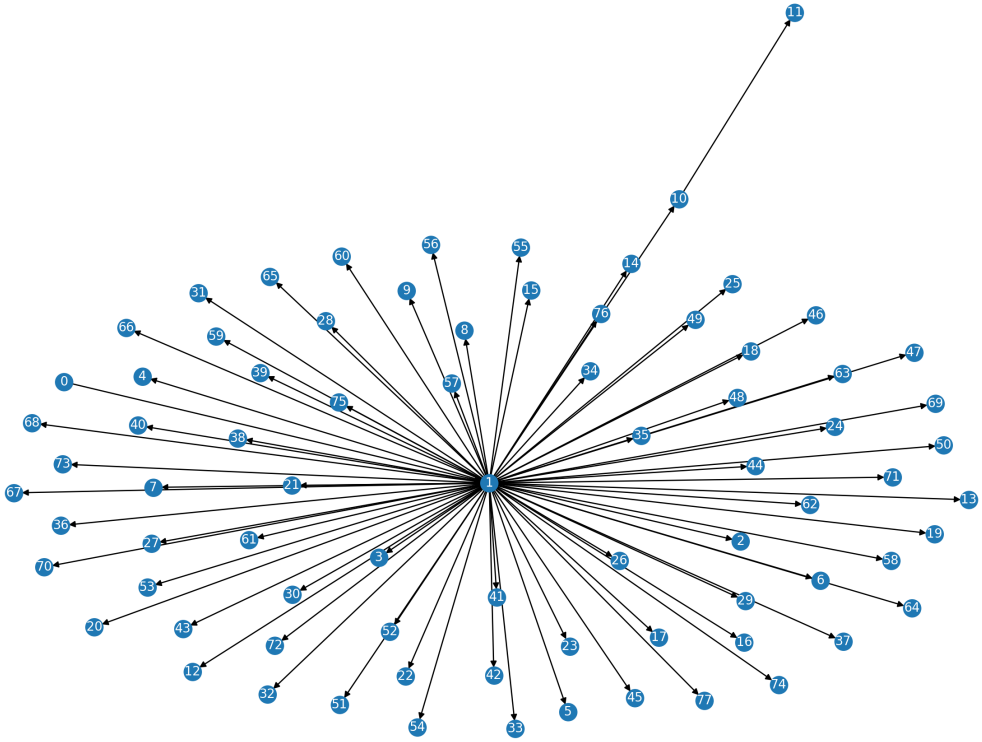

Figure 5.3: The initial event flow graph of event sequences for GNOME Help

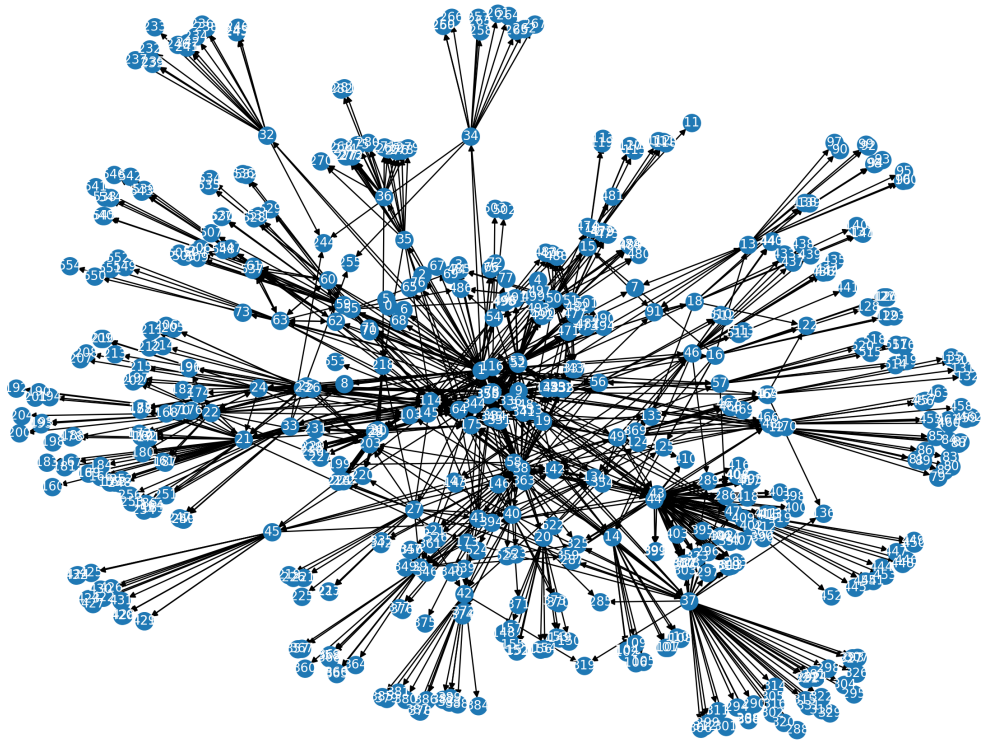

Figure 5.4: The expanded event flow graph of event sequences for GNOME Help

Initially, the generator scans the tree for the available nodes and builds a model from 78 available nodes (widgets) and derives 96 event sequences (Figure 5.3). The Generator proceeds with the application of event sequences and continuously expands the model to the final number of 2,415 nodes and 4,696 event sequences (Figure 5.4). The expanded graph shows which event lead to the discovery of new nodes associated with an action that can be executed by the accessibility system. When compared to manual testing, the effort required to achieve this coverage would be very tedious and time-consuming.

## 5.4 LibreOffice StartCenter

In this section, we describe results achieved by our test generator when we tested LibreOffice applications. Moreover, we discuss the alterations that were required to be implement within our test generator due to issues caused by the accessibility layer. LibreOffice StartCenter (Figure 5.5) is a document management application, it connects 7 other applications from the document suite. Each application can open and edit a different format of documents[4]. LibreOffice is not a part of the GNOME application stack, although it is being shipped as a default document suite in a lot of Linux distributions and has the required accessibility layer support.

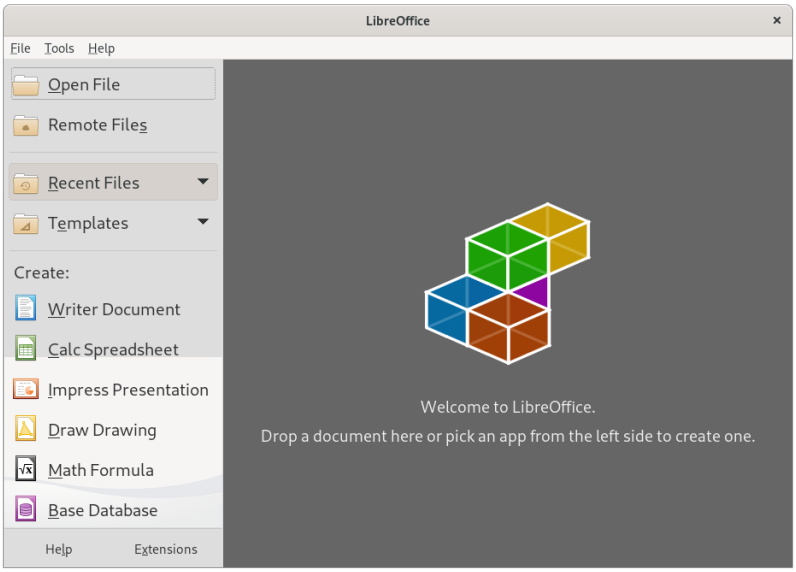


Figure 5.5: LibreOffice StartCenter UI

### Required Implementation Changes and Limitations

During the testing of LibreOffice StartCenter we encountered issues with LibreOffice Calc which forced us to implement application-specific changes to our test generator.

Tests generated for LibreOffice StartCenter are executing all applications from the LibreOffice suite. LibreOffice Calc is a spreadsheet editor that contains theoretically an infinite number of editable cells that are created on demand. Therefore, the application cannot be recursively explored for new nodes. A recursive search causes that the application will generate new cells until the RAM on a virtual machine is depleted. The virtual machine is

[4] https://help.libreoffice.org/3.3/Common/Start_Center

```
1    Test Generator Report for Component: LibreOffice StartCenter
2    Covered Events: 7795/7853 (99.26 %)
3    Number of Covered Nodes: 2516
4    Number of Generated Test Cases: 2315
5    Nodes without the Coverage:
6    ...
7    Generation Time: 11:48:49.071478s
```

Listing 5.6: The final test generator report for LibreOffice StartCenter

unresponsive and even refuses to execute commands through a remote shell. Therefore, the implementation of the test generator was altered to avoid the execution of recursive search querries on LibreOffice Calc.

Another accessibility-related issue has started to occur when the test generation process reached certain types of dialogs (a letter wizard, a fax wizard, etc.). The behavior was quite similar to the previous issue, the test generator hanged on a recursive search query for a while and then failed with the error message shown in Listing 5.5.

```
1 Failed to handle new nodes atspi_error:The application no longer exists (0)
```

Listing 5.5: The accessibility layer error that prevents the generator from node expansion when testing LibreOffice StartCenter

The investigation has shown the application spawns 2 windows, the first one is the previously mentioned dialog, the second one is a generic LibreOffice window that contains only menus. The blank window probably spawned because those wizard dialogs are not standalone applications, they belong to other applications from the LibreOffice suite. This claim is supported by the fact that the blank window only lasts as long as the dialog is opened. The blank window contains widgets but they are not available for interaction as the focus can only be placed on the dialog window in the front. Therefore, the blank window is an issue from the implementation perspective of the generator as it tries to perform node expansion on a window that never becomes available. Since the accessibility system throws the mentioned error, it is handled as an exception and the generator continues without expansion to a next test case. The cancellation of the node expansion process also means that widgets from the affected dialog window are not tested by our generator.

### Testing and Results

Testing LibreOffice required some setup in the `apps.yaml` file. The application instance has to be started with the `--norestore` parameter to avoid the restoration of unsaved documents from previous sessions. Furthermore, the user configurations files located in `~/.config/libreoffice/` are removed during the cleanup process. After the previously discussed implementation changes, the generator has been able to perform a full run on the application and produces a test report that is partially shown in Listing 5.6. A subset of tests generated for LibreOffice Calc is shown in Appendix D.

The report shows that the achieved event coverage was not as successful as with the previous components. The majority of the failed sequences contained nodes that have the action available, but the action could not be executed at a given time.

The report also contained several crashes and errors. A couple of those errors confirmed the Firefox issue mentioned in Subsection 5.3 that we reported. Furthermore, another severe issue was discovered by our solution when the StartCenter crashed after clicking on the Help button. We submitted a bug report[5] for the issue and it has been fixed by developers.

The remaining group of crashes was caused by quite an interesting phenomenon. The test generator reported 6 crashes that appeared to be quite similar. Each of those crashes was triggered by an event associated with a button that was not available for an interaction. A manual application of the reproducers was not possible because the event can only be sent through the accessibility system. Nevertheless, this proves that the proposed test generator can reveal this kind of flaws in GUI software. The testing was performed with the `libreoffice-core-6.3.6.2-1.el8.x86_64` rpm.

## 5.5 Evince

This section introduces us to a document viewer application Evince[6], along with the results that we achieved by testing the application with our test generator.

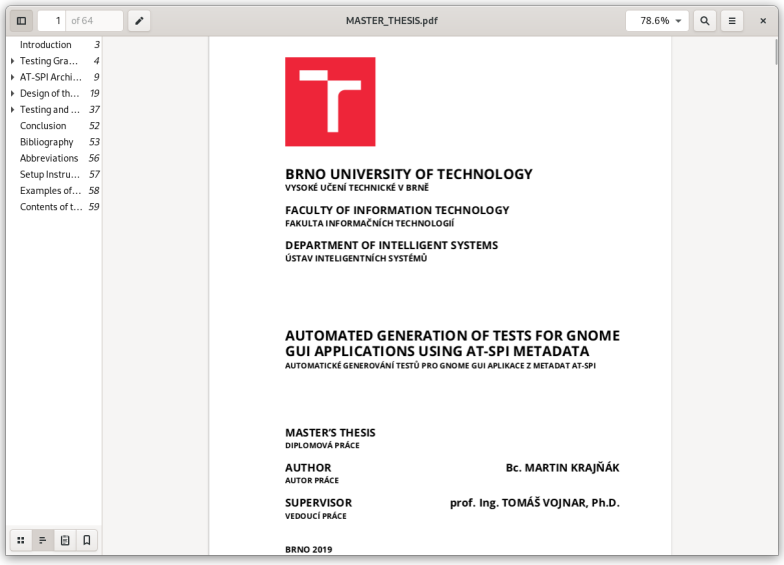


Figure 5.6: Evince document viewer UI

### Testing and Results

Initially, we tested a blank Evince instance without a test file. However, results have shown that a lot of widgets are not available, or are disabled. Therefore, we decided to add a test file in the `.pdf` format as shown in Figure 5.6. A name and a path of the test file is configurable via the `apps.yaml` file. The test file is used during the test case generation process as well as the test case execution process. An inclusion of the test file has unlocked the majority of disabled widgets and allowed us to use the full potential of our test generator.

[5] https://bugzilla.redhat.com/show_bug.cgi?id=1819798
[6] https://wiki.gnome.org/Apps/Evince

```
1 Test Generator Report for Component: Document Viewer
2 Covered Events: 373/398 (93.71 %)
3 Number of Covered Nodes: 217
4 Number of Generated Test Cases: 189
5 Nodes without the Coverage:
6 ...
7 Generation time: 0:29:20.398814s
```

Listing 5.7: Final test generator report for Evince

Listing 5.7 contains the final test generator report. The report section that starts on line 5 contains a list of unexecuted event sequences. The manual reproduction of sequences has shown that the reported event sequences lead to widgets that were not available for an interaction. In two cases, our test generator crashed the application by executing an action on an unavailable widget through the accessibility layer. This is the same issue that we encountered when we tested LibreOffice StartCenter (see Subsection 5.4).

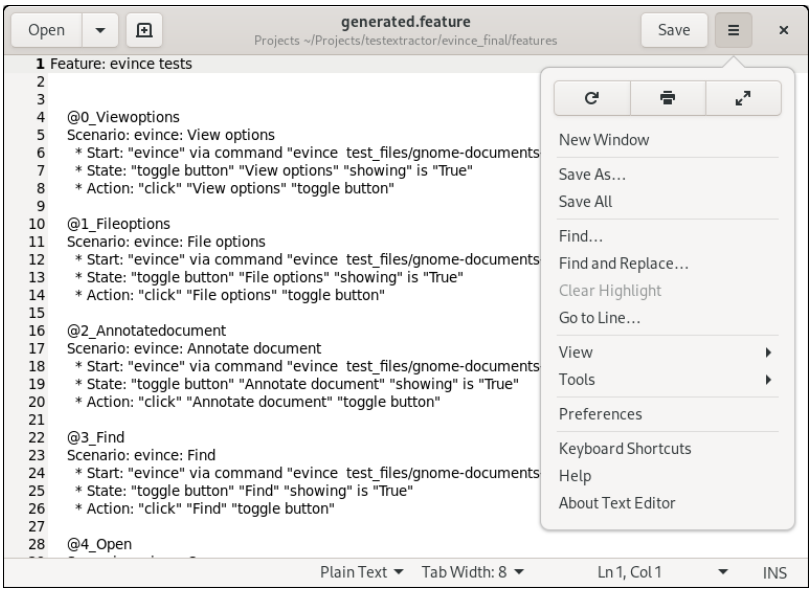


Figure 5.7: Gedit text editor UI

Our test generator revealed an unknown issue with the `Gtk-CRITICAL` error message. We were able to report the issue from the reproducer provided by our test generator (see Appendix C) and we created a bug report[7]. The issue occurs in a test case when a document is being opened in the *Presentation Mode*. The error message is not that severe, however this kind of messages are printed when a critical failure occurs within the application and there are numerous bug reports for the GNOME applications for similar issues. Furthermore, the issues like the one we reported are harder to find, because they can be only detected by checking the *stderr* of a GUI application.

Since the *RHEL 8.3* distribution contains an older version of the application that we tested (`gedit-3.28.1-3.el8.x86_64`), we also decided to test a newer flatpak version of the

[7] https://bugzilla.redhat.com/show_bug.cgi?id=1842017

application (`org.gnome.Evince, Version:3.36.1`). The testing was performed without confirmation of the bug in the newest version. By testing the flatpak version of Evince we confirmed the compatibility of our test generator with flatpak applications.

## 5.6 Gedit

This section presents the results achieved with our test generator when testing Gedit application (see Figure 5.7), which is a text editor of the GNOME desktop environment[8].

### Testing and Results

After the issues we experienced with testing Evince without a test file, we included one for Gedit tests from the beginning. The achieved results are summarized in the final test generator report shown in Listing 5.8. The report contains a lot of unexecuted event sequences that are not displayed in the listing. The issues were quite similar to issues that occured with applications we tested before. Our generator failed to derive an event sequence for a group of widgets in a *Print* dialog and a *Preferences* dialog.

Furthermore, the report contained 9 occurrences of the `Gtk-CRITICAL` error. However, when we applied reproducers from the report manually, the errors did not appear. The errors could only be achieved by execution of actions through the accessibility system. During the investigation, we found out that all 9 test cases have one common denominator, which is a so-called *hamburger menu* shown in Figure 5.7. We examined the menu through the *Sniff* utility to discover that the accessibility system reports a wrong hierarchy of widgets associated with this menu. The menu button is in this case an end node, which means that this is where the event sequence derived for the menu ends. Since the group of widgets associated with this menu is reported by the accessibility layer on a wrong place, our test generator derives a group of event sequences for them without an important middle step – opening a menu. Therefore, the reported errors are not bugs, as the generator triggers an event on a node which is not present on the screen. In conclusion, this is another discovered imperfection of the accessibility system.

```
1 Test Generator Report for Component: Text Editor
2 Covered Events: 1200/1301 (92.23 %)
3 Number of Covered Nodes: 643
4 Number of Generated Test Cases: 565
5 Nodes without the coverage:
6 ...
7 Generation time: 2:48:58.877205
```

Listing 5.8: The final test generator report for Gedit

[8] https://wiki.gnome.org/Apps/Gedit

# Chapter 6

# Evaluation and Future Work

This chapter presents an overall evaluation of our test generator. Section 6.1 discusses the results of a code coverage analysis. In Section 6, we compare the proposed test generator to existing test suites for the GNOME aplications. Section 6.3 describes a recommended workflow for our test generator and concludes this chapter with plans for a future work.

As discussed in Chapter 4, the approach to application testing done by this work combines several testing techniques and takes advantage of existing testing frameworks and libraries. We are not aware of any currently available solutions designed to generate test cases from the AT-SPI metadata in the GNOME environment. The closest related solutions were described in Subsection 2.10.1. However, that solution is designed for applications with different development cycles, where a model is being developed before a prototype of the application is available. Other solutions (e.g. references [21],[5]) rely on static analysis and build a model from the byte code of the GUI applications written in Java.

## 6.1 Code Coverage Evaluation

In this section, we compare our test generator to another GUI test generator based on static analysis. Further, we compare the tests generated by our test generator to the tests written with the script based tools.

Our test generation tool works on black-box testing principles. However, we were able to perform a code coverage analysis for *Evince*, *Gedit*, and *Terminal* to evaluate the amount of source code covered by our solution. To put the results of our code coverage analysis in perspective with similar tools, we compared our results to results acquired by a solution introduced by Artl et al. [5]. However, their solution targets a different platform and it also takes advantage of white-box testing principles. Table 6.1 compares the amount of coverage achieved by our test generator with a range of coverage acquired by their solution on 4 open-source applications written in Java. The code coverage achieved with our test generator did not reach the same results, however there are several factors that must be considered in evaluation.

The analysis performed on the tests generated for *Terminal* has shown that several parts of the GUI code were not executed by the tests. A *gcov* code coverage report provides a detailed analysis of source code. An examination of data provided by the analysis identified a set of functions designed to handle keyboard shortcuts that are not covered by our tests. It is also expected that a large portion of source code is related to non-GUI operations which are not covered by our test cases at all.

| | Our Solution | | | Solution [5] |
|---|---|---|---|---|
| **Application** | Evince | Gedit | Terminal | Selected Java Applications |
| Lines Coverage (%) | 22.4 | 37.2 | 33.2 | 54-62 |
| Functions Coverage (%) | 33.6 | 46 | 41.0 | — |
| Branches Coverage (%) | 16.0 | 23.7 | 20.8 | 26-37 |

Table 6.1: Coverage results achieved with our test generator based on AT-SPI metadata (black-box approach) compared to the test generation tool [5] with a white-box approach

Furthermore, we performed the code coverage analysis of tests generated for *Evince* twice. Our findings have shown, that by including a `.pdf` test file, we raised *lines coverage* in *Evince* by 11.4 %, *functions coverage* by 15.6 %, and *branches coverage* by *5.9* %. With consideration that *Evince* supports 8 other document formats, we could possibly grow the code coverage by including more test files. The similar approach may be applied in *Gedit* as it supports syntax highlighting for various programming languages.

## 6.2 Comparison with Existing Solutions and Test Suites

When it comes to solutions used for test automation for GNOME applications, several Record-and-Replay tools were described in Section 2.7. The proposed solution utilizes *dogtail* combined with the *behave* framework, so the generated test cases are executable even after the generation process is finished.

Scripted test cases are written by humans (quality engineers). Their goal is either to automate scenarios that cover key features in applications or to create scenarios based on previously discovered bugs and defects. However, the proposed test generator is a semi-smart tool. Errors and crashes that occur during the generation process are recognized and reported with a reproducer. The potential problem is with the semantics of the test cases. The test generator can apply a sequence of actions to the application, although it cannot decide whether the outcome is expected. Therefore, the proposed tool should aid testers with development of test automation right after the executable version of an application is available. The main advantage is in the exploratory testing performed during the test generation. The test generator can sequentially execute available events (event sequences), detect errors and crashes, and thus help testers to avoid drawbacks of manual testing. A report from the test generation process will also point out the widgets that were not covered by the exploratory testing and therefore they are not covered by the test cases. An additional benefit is provided by the fairly quick availability of the working test automation. Testers can push either all or a subset of test cases in the CI environment. Any test case can be reviewed and updated, new test cases could be written with available *behave* steps or a new step definition can be added. Generated test cases can be merged with test automation available from the previous versions, if it is not too obsolete.

In conclusion, generated test cases are not comparable with the currently available test automation developed by testers with the script-based tools. The goal of generated test cases is to cover as many events in applications as possible, whereas the currently available test suites are focused on automation of the most essential tasks performed by users in which bugs and defects occurred in the past. This comparison does not include unit tests or any other white-box tests performed on the library level.

The test generator itself is a piece of software as well. It is designed to work with as many applications as possible, therefore, the implementation is as general as possible. If a tested application reveals flaws within the test generator, the adjustments must maintain the general approach to ensure that the generation process for other applications will not be affected. Therefore, if the adjustment is too application-specific, the effort that needs to be done to include the adjustment in the test generator should not be greater than developing a custom test case.

## 6.3 Recommended Usage and Future Work

In Chapter 4, we introduced a recommended workflow with our test generator (Figure 4.1). The workflow contains a review of generated test cases that could be in some cases as time consuming as manual testing. When we tested LibreOffice StartCenter or GNOME Help, the amount of time required for test generation went up to 12 hours. Therefore, we introduced the command-line option `--shallow` that prevents the test generator from the expansion of newly discovered nodes. The option provides the ability to obtain the most essential test automation for an application in a reasonable time (up to 10 minutes), and with the reasonable amount of test cases (up to 100). This so-called *shallow* automation could be quickly reviewed and pushed to a CI environment to perform regression testing when a new version of an application is built. Then, a non-shallow run can be performed to let our generator go through all extracted scenarios and report reproducers for potential issues.

Furthermore, another option could be implemented to restrict the depth of executed event sequences. However, the option can lead to generation of unreasonable test cases, e.g. opening a menu without clicking on menu items, triggering a dialog without clicking a button, etc.

Lastly, we could narrow an amount of generated test cases by definition of a window, a dialog, or a menu we want to test. The implementation of this feature would require us to handle an additional input, specifically a *name* and a *roleName* of a widget as well as an event sequence that is required to navigate to the widget.

# Chapter 7

# Conclusion

In this thesis, we presented our test generator for GNOME GUI applications. The generator utilizes the metadata of accessibility technologies to create an abstract model of an application. The model is then used for identification of event sequences that are executed on a tested application. The extracted event sequences are applied on a live instance of the tested application. The state of the application is monitored for severe issues that could appear during the interaction with the application. Additionally, the generator can discover widgets that appear during the testing and include them to the model. The generator also creates additional assertions based on metadata from the accessibility layer as well as assertions that are performed by the OCR engine *Tesseract*. The event sequences along with assertions are put together in a set of executable test cases written in a *behave* framework. The generated test cases are suitable for regression testing performed by the CI pipeline.

Futhermore, we used our solution to test 5 GNOME GUI applications. For 2 of those applications, we extended the testing on their flatpak versions. The testing performed with our test generator has proven the ability to identify unknown bugs in multiple applications which were reported to developers. We also verified the deployment of the tests generated by our solution by performing several successful test runs with a selected group of tests (*shallow* tests) in the *Desktop-CI* environment used by Red Hat.

We have also described the limitations caused by the accessibility layer that we encountered during the testing. These limitations partially changed our approach to keep the implementation as general as possible and forced us to integrate some application-specific changes in our solution. The majority of issues we encountered with the accessibility in GTK3 applications need to be fixed within the affected applications. However, the accessibility bugs are usually not a priority, unless they are critical.

A plan for future work includes the integration of new parameters that would allow us to test only a selected part of the application. The evaluation of code coverage results achieved by our test generator has shown, that we can possibly increase the level of the code coverage by including more test files supported by applications. Therefore, we might implement a mechanism that will exchange multiple files during the test generation.

# Bibliography


[1] *Automated Testing Advantages, Disadvantages and Guidelines* [online]. 2005 [cit. 2020-04-25]. Available at: http://www.exforsys.com/tutorials/testing/automated-testing-advantages-disadvantages-and-guidelines.html.

[2] *Flatpak* [online]. 2020-03-16 [cit. 2020-04-20]. Available at: https://wiki.debian.org/FlatPak.

[3] Alexander, V., Benson, C., Cameron, B., Haneman, B., O'Briain, P. et al. *GNOME Accessibility Developers Guide* [online]. GNOME Documentation Project, 2008 [cit. 2019-6-11]. Available at: https://developer.gnome.org/accessibility-devel-guide/stable/index.html.en.

[4] Alégroth, F. R. R. L. Visual GUI testing in practice: challenges, problemsand limitations. *Empirical Software Engineering*. New York: Springer US. 2015, vol. 20, no. 3, p. 694–744. ISSN 1382-3256.

[5] Arlt, S., Podelski, A., Bertolini, C., Schaf, M., Banerjee, I. et al. Lightweight Static Analysis for GUI Testing. In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, p. 301–310. ISBN 9781467346382.

[6] Best, S. *Analyzing Code Coverage with gcov* [online]. 2019-03-09 [cit. 2020-01-16]. Available at: https://www.linuxtoday.com/blog/analyzing-code-coverage-with-gcov.html.

[7] Cerza, Z., Rousseau, E., Malcolm, D. and Humpa, V. *Package dogtail* [online]. Red Hat, Inc., 2014 [cit. 2019-12-20]. Available at: https://fedorapeople.org/~vhumpa/dogtail/epydoc/.

[8] Chandel, V. S. *Deep Learning based Text Recognition (OCR) using Tesseract and OpenCV* [online]. Big Vision LLC, 2018 [cit. 2019-12-26]. Available at: https://www.learnopencv.com/deep-learning-based-text-recognition-ocr-using-tesseract-and-opencv/.

[9] Diggs, J. *GTK+ and ATK - A Foundation for GNOME Accessibility* [online]. GNOME Documentation Project, 2011 [cit. 2019-6-11]. Available at: https://wiki.gnome.org/Accessibility/Documentation/GNOME2/AtkGuide/Gtk.

[10] Free Software Foundation, Inc. *Introduction to GNU Xnee* [online]. 2012 [cit. 2020-01-16]. Available at: https://xnee.wordpress.com/.

[11] Gagnon, C. *Xpresser* [online]. 2012-12-22 [cit. 2020-01-16]. Available at: https://wiki.ubuntu.com/Xpresser.

[12] Jaaskelainen, A., Katara, M., Kervinen, A., Maunumaa, M., Paakkonen, T. et al. Automatic GUI test generation for smartphone applications - an evaluation. In: *2009 31st International Conference on Software Engineering - Companion Volume*. IEEE, 2009, p. 112–122. ISBN 9781424434954.

[13] Moreira, P. A. C. N. M. M. A. Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability*. 2017, vol. 27, no. 3, p. n/a–n/a. ISSN 0960-0833.

[14] Nguyen, R. B. B. I. and Memon, A. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*. Boston: Springer US. 2014, vol. 21, no. 1, p. 65–105. ISSN 0928-8910.

[15] Nidhra, S. Black Box and White Box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Applications*. june 2012, vol. 2, p. 29–50.

[16] Nyman, N. In Defense of Monkey Testing. *Software Testing and Quality Engineering Magazine*. 2000-01, p. 18–21.

[17] Odehnal, M. *Automation of Desktop Applications* [online]. 2020 [cit. 2020-04-20]. Available at: https://dogtail.gitlab.io/qecore/doc_basic_automation.html.

[18] OpenCV team. *About Open Source Computer Vision Library* [online]. OpenCV, 2019 [cit. 2019-12-26]. Available at: https://opencv.org/about/.

[19] Parente, P. *Package pyatspi* [online]. IBM Corporation, 2007 [cit. 2019-12-8]. Available at: https://people.gnome.org/~parente/pyatspi/doc/.

[20] Parmar, D. *Exploratory testing* [online]. [cit. 2020-04-20]. Available at: https://www.atlassian.com/continuous-delivery/software-testing/exploratory-testing.

[21] Reis, J. and Mota, A. Aiding exploratory testing with pruned GUI models. *Information Processing Letters*. Elsevier B.V. 2018, vol. 133, p. 49–55. ISSN 0020-0190.

[22] Russo, J. D. *5 Reasons To Use YAML Files In Your Machine Learning Projects* [online]. towardsdatascience.com, 2019 [cit. 2020-4-21]. Available at: https://towardsdatascience.com/5-reasons-to-use-yaml-files-in-your-machine-learning-projects-d4c7b9650f27.

[23] Sacolick, I. What is CI/CD? Continuous integration and continuous delivery explained. *InfoWorld.com*. May 10 2018. Copyright - Copyright Infoworld Media Group May 10, 2018; Last updated - 2020-03-30.

[24] Vincent, L. *Announcing Tesseract OCR* [online]. Google Developers Blog, 2006 [cit. 2019-12-26]. Available at: http://googlecode.blogspot.com/2006/08/announcing-tesseract-ocr.html.

[25] Yuan, X., Cohen, M. B. and Memon, A. M. GUI Interaction Testing: Incorporating Event Context. *IEEE Transactions on Software Engineering*. July 2011, vol. 37, no. 4, p. 559–574. ISSN 2326-3881.

[26] Zander, J., Schieferdecker, I. and Mosterman, P. J. *Model-Based Testing for Embedded Systems*. CRC Press, 2011. ISBN 9781439818473.

# Appendix A

# Abbreviations

| | |
|---|---|
| **ATK** | Accessibility Toolkit |
| **AT-SPI** | Assistive Technology Service Provider Interface |
| **CI** | Continuous Integration |
| **CD** | Continuous Delivery |
| **GAIL** | GNOME Accessibility Implementation Library |
| **GCC** | GNU Compiler Collection |
| **GCOV** | GNU Coverage Testing Tool |
| **GNU** | GNU's Not Unix |
| **GNOME** | GNU Network Object Model Environment |
| **GTK** | The GNOME Toolkit |
| **GUI** | Graphical user interface |
| **LTSM** | Long Short Term Memory |
| **RHEL** | Red Hat Enterprise Linux |
| **pid** | process identification number |
| **rpm** | RPM package manager |
| **RRN** | Recurrent Neural Network |
| **OCR** | Optical Image Recognition |
| **OpenCV** | Open Source Computer Vision Library |
| **UI** | User Interface |

# Appendix B

# Setup Instructions and User Manual

Testing with our test generator requires the following setup:

1. Open a terminal,
2. Copy the contents of the attached medium or clone a git repository by executing
   `$ git clone https://github.com/mkrajnak/testextractor`
3. Based on the underlying Linux distribution (*Fedora* or *RHEL*), install the dependencies located in the *install* folder:
   `$ cd install && sudo dnf -y install ./*.rpm`
4. Enable the AT-SPI:
   `$ gsettings set org.gnome.desktop.interface toolkit-accessibility true`
5. Setup Tesseract (requires external software repositories):
   ```
   $ dnf config-manager -add-repo
   https://download.opensuse.org/repositories/home:
   /Alexander_Pozdnyakov/CentOS_8/
   $ rpm -import https://build.opensuse.org/projects/home:
   Alexander_Pozdnyakov/public_key
   $ dnf -y install tesseract
   ```
6. In the *testextractor* directory, create a virtual environment and install the Python3 dependencies:
   ```
   $ python3 -m venv .venv
   $ source .venv/bin/activate
   $ pip install -r requirements.txt
   ```
7. Intruction to use the test generator are shown Listing B.1.

```
Usage: testgen.py [OPTIONS]

  Accessibility test generation tool for GTK+ applications

Options:
  --app TEXT Name of the application entry in apps.yaml
                        (compulsory)
  --generate-project-only Generates only the project folder for --app
  --disable-ocr Disables OCR
  --shallow Disables the model expansion (test only nodes
                        available after start)
  --verbose Enables verbose logging
  --test INTEGER Regenerates only defined test, expected to be used
                        with --shallow
  --help Show this message and exit.
```

Listing B.1: output of `./testgen.py --help`

# Appendix C

# Test Generator Bug Report

```
1  ...
2  WARNING:
3  TEST:0 contains CRITICAL:
4
5  (evince:15299): glib-critical **: 02:20:18.264: g_variant_new_string:
       assertion 'string != null' failed
6  Steps to Reproduce:
7        * Start: "evince" via command "evince test_files/gnome-documents-
             getting-started.pdf" in session
8        * State: "check box" "Presentation" "showing" is "False"
9        * Action: "click" "Presentation" "check box"
10 ...
```

Listing C.1: A demonstration of the bug found by our test generator

# Appendix D

# Examples of Generated Test Cases

```
1  Feature: libreoffice-startcenter tests
2  ...
3      @10_MasterDocument
4      Scenario: libreoffice-startcenter: Master Document
5        * Start: "libreoffice-startcenter" via command "libreoffice --
             norestore" in session
6        * Action: "click" "File" "menu"
7        * Action: "click" "New" "menu"
8        * State: "menu item" "Master Document" "showing" is "True"
9        * Action: "click" "Master Document" "menu item"
10       * State: "frame" "Navigator" is shown
11       * OCR: "Navigator" is shown on the screen
12
13     @11_Templates
14     Scenario: libreoffice-startcenter: Templates...
15       * Start: "libreoffice-startcenter" via command "libreoffice --
             norestore" in session
16       * Action: "click" "File" "menu"
17       * Action: "click" "New" "menu"
18       * State: "menu item" "Templates..." "showing" is "True"
19       * Action: "click" "Templates..." "menu item"
20       * State: "dialog" "Templates" is shown
21       * OCR: "Templates" is shown on the screen
22
23     @12_Open
24     Scenario: libreoffice-startcenter: Open...
25       * Start: "libreoffice-startcenter" via command "libreoffice --
             norestore" in session
26       * Action: "click" "File" "menu"
27       * State: "menu item" "Open..." "showing" is "True"
28       * Action: "click" "Open..." "menu item"
29       * State: "file chooser" "Open" is shown
30       * OCR: "Open" is shown on the screen
31
```

```
32    @13_OpenRemote
33     Scenario: libreoffice-startcenter: Open Remote...
34       * Start: "libreoffice-startcenter" via command "libreoffice --
             norestore" in session
35       * Action: "click" "File" "menu"
36       * State: "menu item" "Open Remote..." "showing" is "True"
37       * Action: "click" "Open Remote..." "menu item"
38       * State: "frame" "Remote Files" is shown
39       * OCR: "Remote Files" is shown on the screen
40       * State: "dialog" "Remote Files" is shown
41       * OCR: "Remote Files" is shown on the screen
42
43     @14_NoDocuments
44     Scenario: libreoffice-startcenter: No Documents
45       * Start: "libreoffice-startcenter" via command "libreoffice --
             norestore" in session
46       * Action: "click" "File" "menu"
47       * Action: "click" "Recent Documents" "menu"
48       * State: "menu item" "No Documents" "showing" is "True"
49       * Action: "click" "No Documents" "menu item"
50
51     @15_Letter
52     Scenario: libreoffice-startcenter: Letter...
53       * Start: "libreoffice-startcenter" via command "libreoffice --
             norestore" in session
54       * Action: "click" "File" "menu"
55       * Action: "click" "Wizards" "menu"
56       * State: "menu item" "Letter..." "showing" is "True"
57       * Action: "click" "Letter..." "menu item"
58       * State: "dialog" "Letter Wizard" is shown
59       * OCR: "Letter Wizard" is shown on the screen
60
61     @16_Fax
62     Scenario: libreoffice-startcenter: Fax...
63       * Start: "libreoffice-startcenter" via command "libreoffice --
             norestore" in session
64       * Action: "click" "File" "menu"
65       * Action: "click" "Wizards" "menu"
66       * State: "menu item" "Fax..." "showing" is "True"
67       * Action: "click" "Fax..." "menu item"
68       * State: "frame" "<Empty>" is shown
```

Listing D.1: Test cases generated for LibreOffice StartCenter

# Appendix E

# Example of a Generated Project Environment File

```
#!/usr/bin/env python3
import sys
import traceback
from os import system

from qecore.sandbox import TestSandbox

def before_all(ctx):
    try:
        ctx.sandbox =TestSandbox("libreoffice-startcenter")
        ctx.app =ctx.sandbox.get_application("libreoffice-startcenter"
            , a11y_app_name="soffice"
      , app_process_name="soffice.bin"
      , desktop_file_path="/usr/share/applications/libreoffice-startcenter.desktop")
    except Exception as e:
        print(f"Environment error: before_all: {e}")
        traceback.print_exc(file=sys.stdout)
        sys.exit(1)

def before_scenario(ctx, scenario):
    try:
        system("bash cleanup.sh")
        # TODO: Add a custom cleanup before runnnig the test
        ctx.sandbox.before_scenario(ctx, scenario)
    except Exception as e:
        print(f"Environment error: before_scenario: {e}")
        traceback.print_exc(file=sys.stdout)
        sys.exit(1)

def after_scenario(ctx, scenario):
    try:
        ctx.sandbox.after_scenario(ctx, scenario)
    except Exception as e:
        print(f"Environment error: after_scenario: {e}")
        traceback.print_exc(file=sys.stdout)
```

Listing E.1: The `environment.py` file generated by our test generator for LibreOffice StartCenter

# Appendix F

# Event Flow Graphs

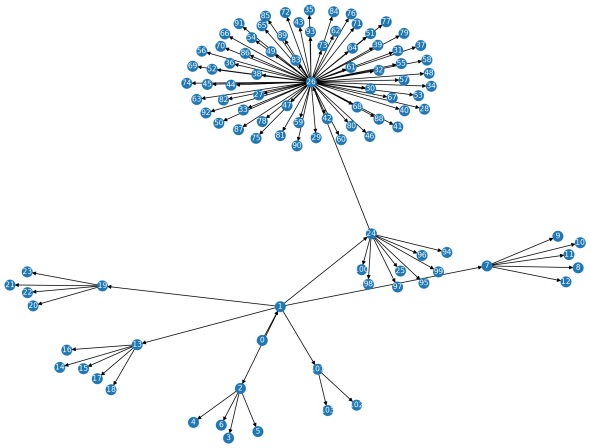

Figure F.1: Initial event flow graph for GNOME Terminal

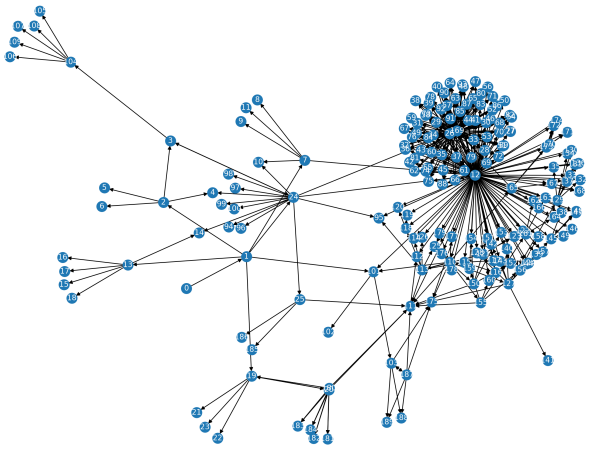

Figure F.2: Expanded event flow graph for GNOME Terminal

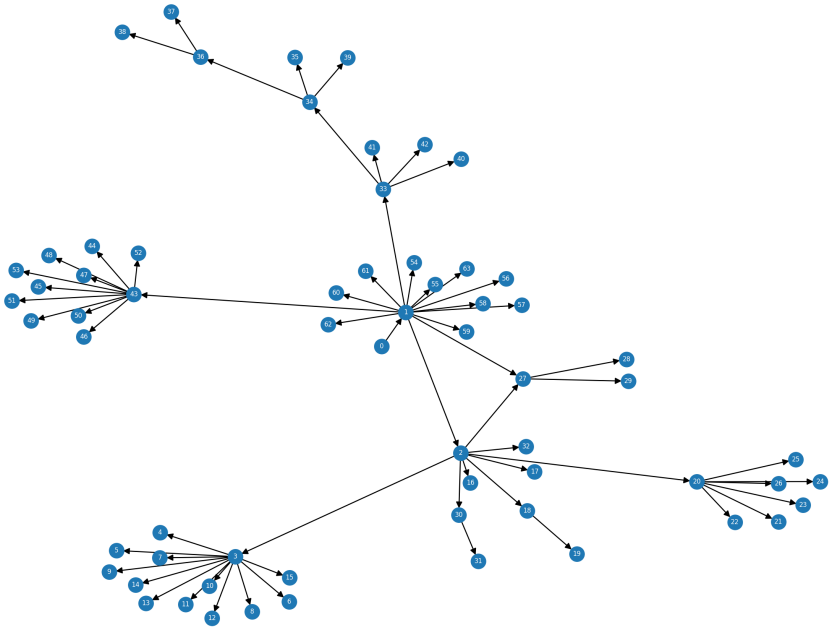

Figure F.3: Initial event flow graph after the start of LibreOffice StartCenter

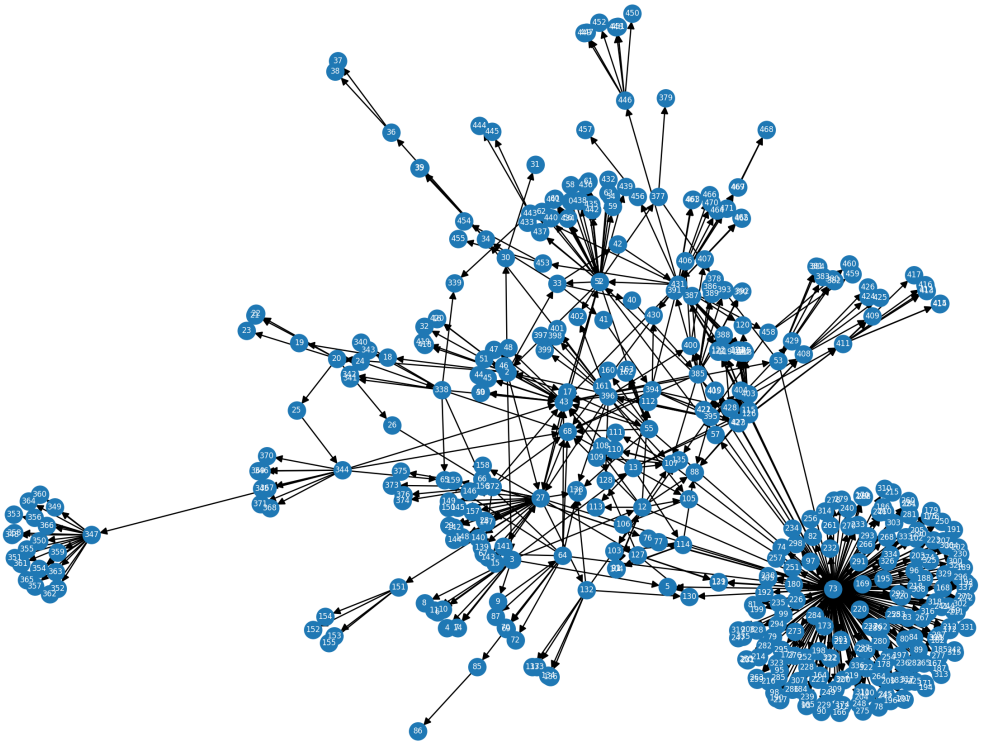

Figure F.4: Final event flow graph of LibreOffice StartCenter

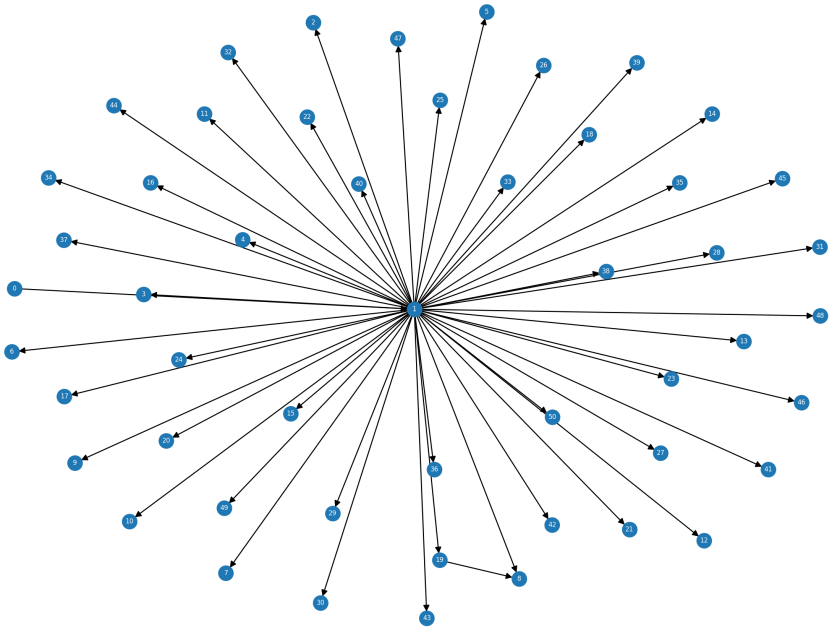

Figure F.5: Initial event flow graph obtained after the start of Evince

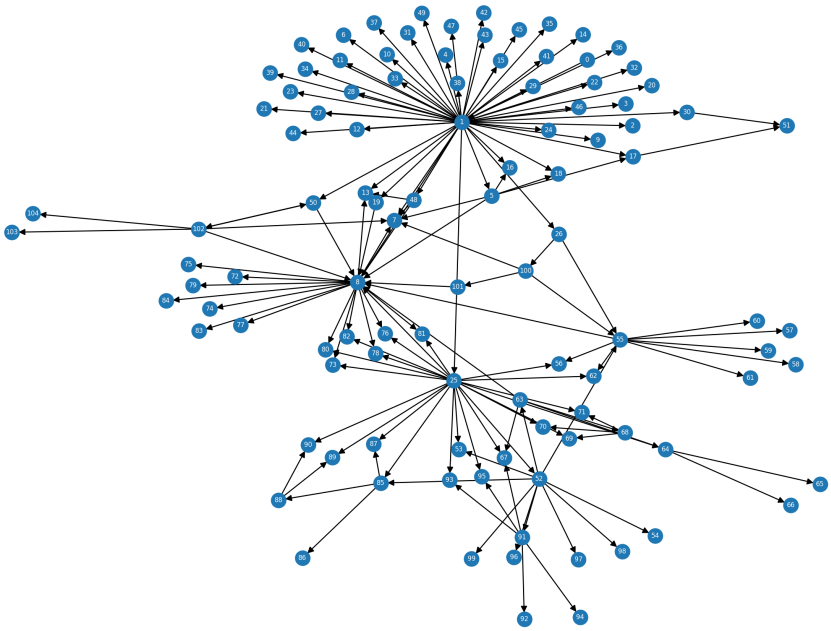

Figure F.6: Final event flow graph of Evince

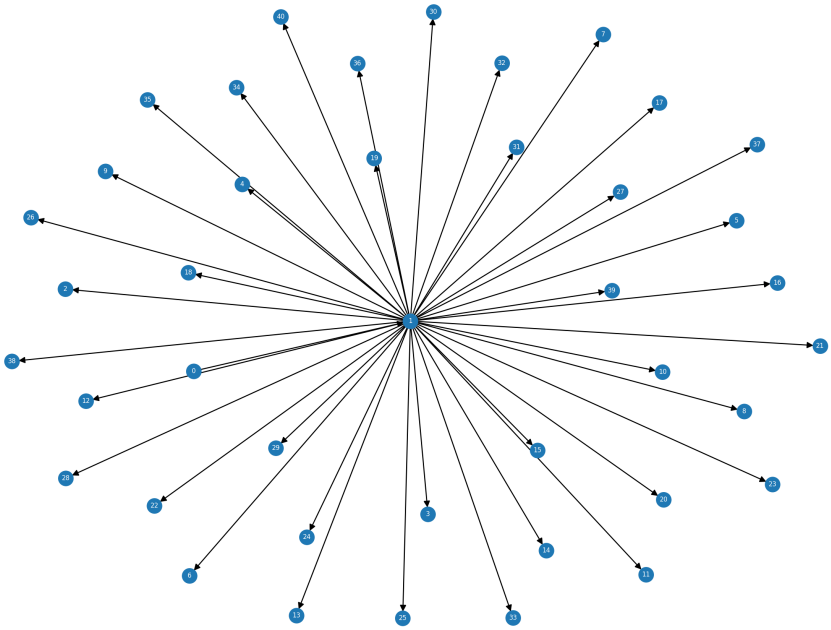

Figure F.7: Initial event flow graph obtained after the start of Gedit

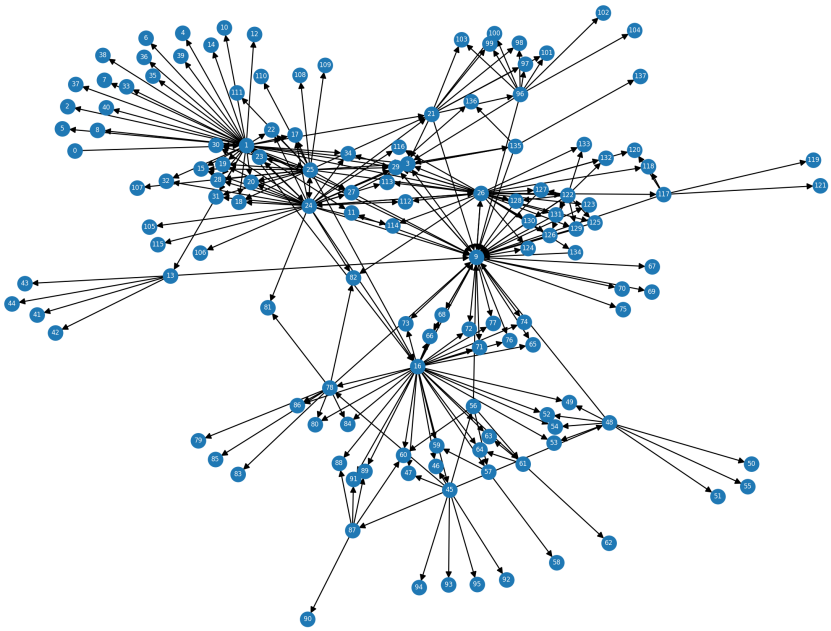

Figure F.8: Final event flow graph of Gedit

# Appendix G

# Contents of the Attached Medium

```
/
├── CI_test_runs - logs from performed CI test runs
├── coverage - the gcov code coverage reports for 3 of test components
├── examples - examples of generated test cases for tested components
├── install - required dependencies
├── testextractor - source files of our test generator
├── text - sources of this thesis
└── xkrajn02.pdf - final version of this thesis
```