Programming Tangible Proxy Interfaces using GUI task demonstrations and follow-up questions

Author: Tom Veuskens

Advisor: Prof. dr. Raf Ramakers

A thesis proposed to achieve the degree of Master in Computer Science

Hasselt University
2018-2019
HASSELT UNIVERSITY

Abstract

Master in Computer Science

Programming Tangible Proxy Interfaces using GUI task demonstrations and follow-up questions
by Tom Veuskens

The work presented in this thesis investigates novel techniques for automating GUI actions for users without any technical background. The thesis presents a toolkit which has been developed to proxy user interface behaviour to and from the physical world. To accomplish this, novel pixel-based reverse engineering approaches are developed which leverage programming-by-demonstration concepts as well as automated follow-up questions to allow easy specification of user interface behaviour. In addition to the toolkit, a novel tangible kit with an accompanying tangible programming paradigm was developed which is specifically designed to allow easy linking of a tangible to the demonstrated user-interface behaviour to provide input from the physical world to the digital environment, or output from the digital environment to the physical world. To overcome the inherent limitation of pixel-based reverse engineering approaches, which require pixels to be visible in order to be analysed, a novel solution is presented which is leveraged to process non-visible pixels in addition to visible pixels.
Acknowledgements

Without the help of several individuals, it would have been impossible to realise the work presented in this thesis. I would like to express my deepest gratitude to everyone who contributed in one way or another in realising this thesis.

First and foremost, I would like to thank my advisor, Prof. dr. Raf Ramakers, for giving me the opportunity to work on this highly interesting topic. The weekly guidance and feedback on my work helped shape the thesis to what it has become today. Your insightful ideas and suggestions laid the foundation of many of the features explored in this thesis. In addition, I would like to thank Prof. dr. Raf Ramakers and Prof. dr. Kris Luyten for inviting me to two summer internships at the Expertise Centre for Digital Media. These internships truly introduced me to the HCI research culture, and helped me further develop myself.

In addition to Prof. dr. Raf Ramakers, I would like to thank everyone who was present at the bi-weekly Master Thesis meetings. In particular, I would like to thank Danny Leen, Jose Maria Tijerina Munoz, and Tom Valkeneers for criticising my work and providing feedback. Some of your comments lead to great improvements of the work presented in this thesis.

On a more personal note, I would like to thank my parents to give me the opportunity to pursue a Master’s degree, and for supporting me in every choice I make. Last, but certainly not least, I would like to express profound gratitude to my girlfriend for bearing with me in one of the most stressful periods of my life. Thank you for your endless support, patience, and love Jana!

Tom Veuskens
# Contents

Abstract i

Acknowledgements ii

List of Figures vi

List of Code Examples ix

1 Introduction 1

1.1 Automating digital user interfaces 2

1.2 Physical interfaces 4

1.3 Our approach 5

1.4 Thesis structure 6

2 Related work 7

2.1 Introduction 7

2.2 Automating the desktop 7

2.2.1 Traditional approaches to automate desktop actions 7

2.2.2 Pixel-based techniques 8

2.2.3 Hybrid approaches 13

2.2.4 Other platforms 13

2.3 Proxy interfaces 15

3 System overview 19

3.1 Introduction 19

3.2 Example cases 19

3.2.1 Introduction 19

3.2.2 Example case 1: Music player 20

3.2.3 Example case 2: Image manipulation 23

3.2.4 Example case 3: Mail 24

3.2.5 Example case 4: Weather 26

3.3 Workflow 26

3.4 Supported User Interfaces 28

3.4.1 Introduction 28

3.5 Supported Tangible Proxy Interfaces 29

3.5.1 Introduction 29

3.5.2 Input Proxy Interfaces 32

3.5.3 Output Proxy Interfaces 33

3.6 Conclusion 34

4 Implementation of the software toolkit 35

4.1 Introduction 35

4.1.1 Programming language and system setup 35

4.1.2 Motivation and chapter overview 36
4.2 Recording the sequence ........................................ 38
  4.2.1 Introduction ............................................ 38
  4.2.2 Managing applications ................................ 39
    4.2.2.1 Introduction .................................... 39
    4.2.2.2 Finding installed applications .................. 39
    4.2.2.3 Starting an application ......................... 41
    4.2.2.4 Moving and focusing an application ............ 41
    4.2.2.5 Closing an application .......................... 42
  4.2.3 Tracking user input .................................. 43
    4.2.3.1 Introduction .................................... 43
    4.2.3.2 Tracking mouse input ............................ 43
    4.2.3.3 Tracking keyboard input .......................... 44
  4.2.4 Identifying clicked icons ............................. 46
    4.2.4.1 Introduction .................................... 46
    4.2.4.2 Taking screenshots ............................... 46
    4.2.4.3 Finding the clicked icon ....................... 48
    4.2.4.4 Ignoring clicks delivered to the interface .... 50
  4.2.5 Adding the action .................................... 51
    4.2.5.1 Introduction .................................... 51
    4.2.5.2 Unique features ................................ 51
    4.2.5.3 Enlarging an icon ............................... 52
    4.2.5.4 Specifying the final widget .................... 53
  4.3 Finding icons ............................................ 54
    4.3.1 Introduction ........................................ 54
    4.3.2 Finding visible icons ............................... 56
      4.3.2.1 Interacting with the web server ............... 56
      4.3.2.2 Exposed functions ............................ 57
    4.3.3 Finding non-visible icons ......................... 59
      4.3.3.1 Introduction ................................ 59
      4.3.3.2 Screen capture of invisible screens .......... 60
      4.3.3.3 Recognize icons on the captured screen ..... 62
  4.4 Checking the sequence .................................. 62
    4.4.1 Introduction ........................................ 62
    4.4.2 Ambiguities ........................................ 63
      4.4.2.1 Introduction ................................ 63
      4.4.2.2 First iteration ................................ 63
      4.4.2.3 Second iteration ................................ 65
    4.4.3 Scrollpanel ......................................... 66
    4.4.4 Text ................................................. 66
  4.5 Executing an action ..................................... 68
    4.5.1 Introduction ........................................ 68
    4.5.2 (Re)starting an application ....................... 69
    4.5.3 Executing the sequence ............................ 69
      4.5.3.1 Introduction ................................ 69
      4.5.3.2 Clicking actions .............................. 69
      4.5.3.3 Typing actions ................................. 71
    4.5.4 Attempt scrolling .................................. 72
    4.5.5 Controlling the final widget ..................... 75
      4.5.5.1 Introduction ................................ 75
      4.5.5.2 Button .......................................... 75
      4.5.5.3 Toggle .......................................... 75
5 Implementation of Tangible Proxy Interfaces

5.1 Introduction ................................................. 79
5.2 Used hardware .............................................. 80
  5.2.1 Introduction ........................................... 80
  5.2.2 Microcontroller Development Board ....................... 80
  5.2.3 Input Proxy Interfaces .................................. 80
  5.2.4 Output Proxy Interfaces ................................ 81
5.3 Programming the hardware .................................. 82
  5.3.1 Introduction ........................................... 82
  5.3.2 WiFi ................................................... 82
  5.3.3 Button and toggle ...................................... 83
  5.3.4 Potentiometer .......................................... 83
  5.3.5 Raspberry Pi LCD ...................................... 84
  5.3.6 Monochrome OLED ...................................... 84
5.4 Coupling a Tangible Proxy Interface ....................... 85
  5.4.1 Introduction ........................................... 85
  5.4.2 Recognising a tangible on a touch screen ............... 86
  5.4.3 Identifying the touched tangible ......................... 88
  5.4.4 Communicating between a tangible and the software toolkit 89
5.5 Interacting with a Tangible Proxy Interface ............... 90
5.6 Conclusion .................................................. 91

6 Discussion and future work .................................. 92

6.1 Introduction ................................................ 92
6.2 Extending the tracking and replay paradigm ................ 93
6.3 Handling changing interface elements ....................... 94
6.4 Extending the set of Tangible Proxy Interfaces ............ 95
6.5 Truly wireless Tangible Proxy Interfaces ................... 96
6.6 Polishing the Tangible Proxy Interfaces .................... 97
6.7 Conclusion .................................................. 98

7 Conclusion .................................................... 99

7.1 Introduction ................................................ 99
7.2 Research challenge 1 ....................................... 99
7.3 Research challenge 2 ....................................... 100
7.4 Research challenge 3 ....................................... 100
7.5 Research challenge 4 ....................................... 101

A Dutch Summary ................................................. 102

A.1 Inleiding .................................................... 102
A.2 Programmeren door te demonstreren ...................... 104
A.3 Robuuste pixel-gebaseerde aanpak ......................... 104
A.4 Fysieke objecten koppelen ................................ 105
A.5 Pixel-gebaseerde methodes in de achtergrond ........... 106
A.6 Conclusie .................................................. 106

Bibliography ..................................................... 108
List of Figures

1.1 A manually adapted remote. ............................................. 2
1.2 RetroFab: a retrofitted desk lamp [71]. .......................... 3

2.1 An example Sikuli script to move all office documents to the recycle bin [90]. ......................................................... 9
2.2 Sikuli Slides example. ...................................................... 10
2.3 Prefab prototype for Windows Vista button [22]. ............. 10
2.4 Spotify interface. .......................................................... 11
2.5 Help, it looks confusing: supporters. .............................. 12
2.6 ActionShot’s history view [46]. ........................................ 14
2.7 User Interface Façade: original application on the left, façade dialog in the middle, and resulting façade on the right [78]. ...... 16
2.8 AMI: provided tangible controls [74]. ............................. 16
2.9 VoodooSketch: combining physical controls with sketched controls [13]. ........................................... 17

3.1 The interface to used to demonstrate an action to the system. (a) A button to add a new external application to the recording. (b) A button to indicate the final widget or icon has been reached. (c) A button to close the recording application. (d) The list of recorded actions, which is currently empty. (e) The list of installed applications in which the user can record an action. ....................................................... 20
3.2 The recording interface, with Spotify is opened in the free area. . . . 21
3.3 (a) Add a button action interface. (b) Add a action for the play button belonging to the song "It’s A Risk". ................................. 22
3.4 An enlarged play button. .................................................. 22
3.5 Follow-up question for text: the icon "Favorite songs" has been clicked, and the system identifies text in the clicked icon. .............. 23
3.6 (a) Add a toggle action interface. (b) Add a toggle for the "play" and the "pause" button in the Spotify interface. ..................... 23
3.7 (a) Add a slider action interface. (b) Add a slider for the volume slider in the Spotify interface. ........................................ 24
3.8 (a) Add a monitor action interface. (b) Add a monitor for the number of unread emails in Gmail. .................................. 25
3.9 A demo of the text monitoring feature to the physical output proxy. (a) The interface in Gmail [31] showing the label (Inbox) and the number of unread emails (14). (b) Showing the values extracted from (a) on a physical proxy interface. ........................................... 25
3.10 A demo of the area monitoring feature to the physical output proxy. (a) The area of the interface which is being monitored, in this case a part from the weather widget from www.weather.com. (b) Showing the monitored area on the physical output proxy with the LCD screen. ... 26
3.11 The pipeline a user and system will follow when specifying an action in the toolkit. The user symbol indicates a step that the user executes. The computer symbol indicates a step executed by the system. (a) Indicates the steps in the pipeline which are handled or executed in the digital environment, and are thus implemented in the toolkit (brown colored steps) (b) Indicates the steps performed in the physical world (green colored steps). (c) Indicates steps which are only performed once, during specification. (d) Indicates steps which can repeat indefinitely, during everyday usage of the tangibles.

3.12 Example buttons from a variety of real user interfaces. (a) The forward and previous button in the Spotify [1] interface. (b) Three buttons in the Windows Steel Layout style, which was used in Windows Vista. (c) The compose and refresh button in the Gmail [31] interface.

3.13 Example toggles from a variety of real user interfaces. (a) Toggles found in the Spotify [1] interface. Left: a toggle between the play and pause state. Right: a toggle in the settings menu (b) Toggles found in the MacOS Mojave Settings. Top: a toggle in the form of a checkbox. Bottom: a toggle in the form of radio buttons. (c) Toggles found in the Gmail [31] interface. Top: the toggle to star or unstar a mail. Bottom: a toggle for a setting in Gmail.

3.14 Example sliders from a variety of real user interfaces. (a) The volume slider found in the Spotify [1] interface. Top: the volume slider without hovering it. Bottom: the volume slider while hovering it. (b) Two sliders found in the MacOS Mojave Settings to change mouse or trackpad settings. (c) A vertical slider to set the volume on www.facebook.com.

3.15 Example monitoring controls from a two real user interfaces. (a) The number of unread emails in Gmail [31]. (b) The weather widget from www.weather.com.

3.16 The set of developed physical input proxy interfaces. (a) A physical button proxy interface. (b) A physical toggle proxy interface. (c) A physical slider proxy interface.

3.17 The set of developed physical output proxy interfaces. (a) A physical color screen. (b) A physical monochrome screen.

4.1 The high-level flowchart followed to execute an action.

4.2 The flowchart followed to execute a sequence when executing an action.

4.3 (a) The original screenshot. (b) The screenshot converted to gray-scale. (c) A sobel filter applied to the gray-scale image in (b) in the Y-direction. (d) The result of a Sobel filter applied to the gray-scale image in (b) in the X-direction added to the same operation in the Y-direction as shown in (c). (e) the AbsDiff function applied to (d). (f) A binary filter applied to (e) to remove noise. (g) The bounding boxes extracted from (f), shown on top of the original screenshot shown in (a).

4.4 The bounding rectangles extracted for the play button, the forward button, and the previous button in the Spotify interface.

4.5 The list of favorite songs in the Spotify interface.

4.6 A traditional interface in Windows Vista with clear borders around each widget and panel.

4.7 The architecture of the application. The C# application communicates with Java by using web requests. The Java web server localises images through an adapted Sikuli JAR.
4.8 The spatial functions of Sikuli [18]. ........................................ 59
4.9 The screen to start a recording of an application with the Screen Capture API. ..................................................... 61
4.10 Windows 10: multiple desktops. ............................................. 61
4.11 The first approach taken to resolve ambiguities. (a) The ambiguities highlighted, so the user can select the right one. (b) The selected icon highlighted in green, and the non-selected icon highlighted in red. (c) An unique feature selected. .................................................. 64
4.12 Resizing panels in the Spotify interface. (a) The original interface, where every label is visible and occupies one line. (b) The resized interface, where some labels get partially hidden and one label occupies two lines. ................................................................. 67
4.13 An interface where the current implementation will fail on. (a) The visible area of the interface, where two matched ambiguities are highlighted in red. (b) The part of the interface which is hidden behind a scroll panel, where the actual target is highlighted in green. ................. 74

5.1 The used microcontroller development board in every Tangible Proxy Interface: a NodeMCU ESP32 development board [42] by Joy-it [41]. 81
5.2 The settings used to upload code to the NodeMCU ESP32 from the Arduino IDE. (a) Adding the ESP32 boards to the boards manager. (b) Select the right board, the right upload speed, frequency, etc. .......... 82
5.3 The bottom side of a Tangible Proxy Interface, containing four 1x1 centimetre conductive foam [44] pads in the corner, which are interconnected with copper tape to form a conductive whole. ................. 87
5.4 The system recognising one of the Tangible Proxy Interfaces being placed on a touchscreen. The drawn rectangle and text is only used for debugging and illustrating purposes. ................................. 88
5.5 A capacitive touch sensor inside a Tangible Proxy Interface, which allows the tangible to be aware of touches. ....................... 89

6.1 Individually changing icons: icons which change their appearance based on a variable they represent in real interfaces. (a) Icons in the MacOS dock, which can either show no notification, or show a badge containing the number of notifications. (b) The battery percentage indicator on MacOS, which changes appearance based on the percentage of remaining battery and based on whether the laptop is charging or not. (c) A conversation in iMessage, where the layout of icon changes based on the length of the message and the appearance changes based on the message. .......................................................... 95
6.2 (1) The Powerboost 1000C in action [2] to power a Raspberry Pi with a screen attached. (a) An external power supply providing power to the Powerboost, charging the battery shown in (b) and providing power to the output in (c) while connected. (2) An example of a magnetic micro-USB charger [85] with in (d) the tip which goes into the device and in (e) the charging cable which snaps to the tip in (d). ................. 97
List of Code Examples

4.1 PowerShell: Get installed applications with install locations. 40
4.2 Start a process from C#. 41
4.3 Move and focus a process from C#. 42
4.4 SetWindowsHookEx function in C#. 43
4.5 Converting key events to a typed action. 45
4.6 Capture a window by its handle from C#. 47
4.7 Sikuli: configure OCR with Tesseract settings. 58
4.8 Algorithm to select the best match. 66
4.9 Algorithm to perform a click in the sequence. 69
4.10 The signature of the SetCursosPos [56] function in C#, together with the convience function to call it. 70
4.11 The signature of the mouse_event [54] function in C#, together with the convience function to call it to perform a click (left button down and left button up events chained). 71
4.12 The convenience function used to control the scroll wheel. 73
5.1 WiFi communication functions used on the NodeMCU. 83
5.2 Draw an image on the RPi LCD with a NodeMCU by using the TFT_eSPI library. 84
5.3 Draw text on the SSD1305 OLED panel by using the Adafruit_SSD1306 library on the NodeMCU. 85
5.4 An example JSON string used to communicate between a Tangible Proxy Interface and the toolkit. 90
Chapter 1

Introduction

Over the years, digital user interfaces have become increasingly extensive with an expanding number of widgets and variations on these widgets being introduced by both the industry as well as research projects. In contrast, the possibilities to control these widgets are very limited, as they only can be controlled with general input devices most of the time, such as a mouse, a keyboard, or a joystick. Instead, the work presented in this thesis take a different approach to controlling these widgets. We will focus on providing a physical counterpart to control a widget or an action on a digital user interface. We will refer to these physical counterparts as physical proxy interfaces.

Physical proxy interfaces have several advantages over traditional general input devices. First of all, individuals, such as disabled people, might not be able to control general input devices. This might be because they lack the fine-grained control needed to operate e.g. a mouse. In addition, these people might not need the flexibility offered by a modern desktop computer as they might only want to perform a very specific task on a computer, such as controlling their music player. Oftentimes we see these disabled people using manually adapted physical interfaces to perform their tasks, as shown in figure 1.1. While this approach might help the user, it is not foolproof as wrong actions can still be performed which can not be undone, for example by sitting on the remote. A better solution would be to adapt the physical interface to better suit the needs of the user, as previously done in RetroFab [71]. While RetroFab retrofits existing physical interfaces with new physical interfaces, as shown in 1.2, the work presented in this thesis focuses on providing physical interfaces to control arbitrary digital interfaces on traditional desktop computers.

We argue a wide variety of users can benefit from controlling a digital interface with physical controls. As previously noted, disabled individuals could benefit from physical controls to digital interface elements as it can be configured to only use the controls they are familiar with and to limit the system to only perform the actions they need, similar to the approach proposed by Seymour et al. [74]. However, also more experienced computer users could benefit from the same system. For example, graphical designers, creative professionals, or artists could use physical controls to access digital interface elements to adjust the brush size in their favorite design tool while simultaneously drawing the line with their other hand, similar to the functionality offered by commercial tools, such as Palette Gear [68] and Novation Dicers [73]. Having a tangible control mapped to a specific function on a digital interface stimulates two-handed interactions, as pointed out by Fitzmaurice et al. [29].
Chapter 1. Introduction

2

Figure 1.1: A manually adapted remote.

Commercial products, such as Palette Gear, already offer tangible controls to control several functions on a digital user interface. However, these tools tend to focus on specific user groups such as creative professionals or artists. As a result, this product only works with specific sets of software interfaces. As an example, Palette Gear focuses mainly on providing physical controls which automate functionality in Adobe [5] programs. Controlling the digital user interface in this case is accomplished by executing keyboard shortcuts while operating the physical control.

While this approach works well for specific use cases supported by Palette Gear, it is clear that this approach has limitations when one wants to control other applications, not directly supported by Palette Gear. For example, it is not possible to control an action on a digital interface for which no keyboard shortcut is available. This is a pretty severe limitation, as explicit developer support is needed to provide keyboard shortcuts to every action in every application. In addition, it can be a tedious process to setup and fine-tune these controls to the work flow of a specific user.

1.1 Automating digital user interfaces

Current desktop automation tools, such as Jitbit [75] and QuickKeys [79], enable people with limited computer knowledge to automate the execution of actions on a computer screen. While these types of software tools are ideal for strictly repetitive tasks, automating tasks in more dynamic user interfaces is not supported by these existing systems. For example, applications such as Spotify, in which the interface can vary between sessions due to, for example, personalized recommendations, cannot easily be automated.

Previous research enabled people to replace these desktop automation tools with
FIGURE 1.2: RetroFab: a retrofitted desk lamp [71].
more advanced tools, such as Sikuli [90], RecurBot [38] and CoScripter [45]. While these research tools try to accomplish goals such as providing the user with greater control on how to specify a desktop automation task, or to increase performance by automating repetitive actions, they all focus on more experienced computer users, such as programmers, preventing computer novices from using similar tools. To accomplish their goals, they allow variations between different executions of the same automation to be generalized or parameterized. In Sikuli [90], this is accomplished by programming a script based on screenshot parameters. On the other hand, CoScripter [45] takes an approach where parameters differ between users. In this case, every user has a personal database containing personal information such as their email address and name, which get inserted into the script during execution. With the additional flexibility offered by these tools however comes an increased complexity. End-users need to be able to specify scripts, which can contain basic programming constructs (such as for- and while-loops) or to be able to create a personal database, which can be cumbersome and exclude computer novices from using such tools.

Because previous research doesn’t provide end-users with tools to automate desktop actions in an user-friendly way, we identify a significant research challenge:

1. Research challenge 1: provide end-users with a tool to automate desktop actions without requiring knowledge of basic programming constructs or any other technical knowledge.

1.2 Physical interfaces

As previously noted, we define the concept of controlling a digital user interface with a physical control as a physical proxy interface. While previous work, such as AMI [74] and VoodooIO [82, 81], provide cheap modular components to quickly prototype physical interfaces, none of the approaches focus on coupling these physical controls to an arbitrary digital user interface. Instead, they are used to reconfigure interfaces to a specific user’s needs [74] or provide an API to be implemented by an application developer [82, 81]. In addition, the end goal of a novice end-user is not to prototype physical interfaces, but instead to link physical controls to existing digital controls in a user interface.

While these approaches are useful for specific use-cases, it is troublesome for novice end-users to couple the prototyped physical controls to an arbitrary action on a digital user interface. However, these approaches provide useful insights in which physical controls should be provided for successfully implementing proxy interfaces, such as buttons, toggles, and sliders.

In addition, many existing approaches, such as VoodooIO [82, 81] don’t focus on end-user specification of physical interfaces. As a result, knowledge of electrical components is required to use such systems, which in turn excludes novice computer users or disabled people from using similar systems.


1.3 Our approach

In the aforementioned sections, several limitations of the most relevant related research projects and commercial products are pointed out. As these are severe limitations to accomplish the goal of this thesis we have identified several research challenges to address the most important limitations of related work towards our goal:

1. Research challenge 1: Provide end-users with a tool to automate desktop actions without requiring knowledge of basic programming constructs or any other technical knowledge.

2. Research challenge 2: A robust pixel-based reverse engineering approach for automating dynamic user interfaces.

3. Research challenge 3: A tangible programming paradigm for non-programmers to link physical controls to automated user interface behavior as defined in (2).

4. Research challenge 4: Overcome the inherent limitation of pixel-based approaches which requires pixels to be visible on the screen in order to be recognised.

To address these research questions, we have implemented a low ceiling, high threshold [62] prototype, which focuses on providing both novice end-users as well as experienced computer users with a tool to automate arbitrary digital user interfaces and to couple a physical proxy interface to this automated action.

The goal of automating an arbitrary digital user interface is accomplished by allowing the user to demonstrate the action they want to automate as they normally would and by recording every step the user performs. After recording the sequence of actions, the system will replay the sequence to check for ambiguities and resolve these by asking follow-up questions to the user, similar to the approach taken by Intharah et al. [39].

To work towards the third research challenge, handling the linking of physical proxy controls to an action on a digital user interface, the work presented in this thesis provides the user with pre-built physical controls containing only low-cost, off-the-shelf hardware. These pre-build physical controls come in different shapes and functionalities, such as buttons, toggles, and sliders. This choice is made to reach the most target users as possible. For example, physically disabled persons might not be able to control a physical slider, and as such would benefit from three buttons to control the same digital slider, linking one button to 0% on the digital slider, one to 50%, and one to 100%. In addition, by providing different types of physical controls, more experienced computer users can adapt their work flow to be as productive as possible.

Since the goal of our approach is to be accessible as possible, we again provide a low threshold [62] to the user for pairing a physical proxy interface to a digital user interface automation. Pairing can be accomplished by putting the physical control on an enlarged view of the widget the automation wants to control. By several heuristics, the system decides how the user wants to couple the physical controls to the digital
widget, for example by placing three physical buttons on a digital slider, the system will suggest the aforementioned distribution.

1.4 Thesis structure

In the remainder of this thesis we present our work in the following structure:

1. Chapter 2: this chapter provides an overview of the current state-of-the-art research in both automating digital user interfaces as well as physical proxy interfaces.

2. Chapter 3: this chapter gives an overview of our approach, stating all of the aspects so subsequent chapters are easier to follow.

3. Chapter 4: this chapter provides implementation details of the software aspect of our approach.

4. Chapter 5: this chapter provides implementation details of the hardware aspects of our approach.

5. Chapter 6: this chapter provides a discussion and talks about the possible directions of future work.

6. Chapter 7: this chapter provides a conclusion of the thesis.

7. Appendix A: this appendix provides a dutch summary of the thesis.
Chapter 2

Related work

2.1 Introduction

The goal of this thesis is to provide the user with an easy-to-use toolkit to couple physical proxy interfaces to arbitrary actions on a digital user interface. As such, this thesis focuses on two distinct parts. Firstly, related work about automating the desktop will be discussed in section 2.2. This will be followed by an overview of past and current research about proxy interfaces and tangible controls in section 2.3.

2.2 Automating the desktop

2.2.1 Traditional approaches to automate desktop actions

In the past, multiple approaches to automating actions on a digital user interface have been proposed. Automating actions on a digital user interface is an interesting research challenge as it requires a constant trade-off between the ceiling and threshold [62] of a system. This means that current systems that are very flexible in the actions they can automate on a digital user interface (and thus have a high ceiling) are typically hard to use as they require specific knowledge. On the other hand, systems that are very easy to use (and thus have a low threshold) are typically very limited in what they can and cannot automate, resulting in a low ceiling. From this, it is clear to see that the challenge is to build a system that is easy to use yet flexible, allowing automation of arbitrary digital user interfaces.

Throughout the past century, several different approaches to building systems for desktop automation can be noted. Traditional approaches require support from the application developer, for example, by them implementing accessibility APIs [78]. However, Hurst et al. found 26% of elements completely missing in Accessibility APIs [36] due to application developers failing to implement the API. As a result, Hurst et al. [36] proposed a solution to enrich the accessibility APIs in an attempt to automatically identify targets users interact with during real-world tasks. While the accessibility API correctly identifies 74% of targets, their proposed approach is able to find the size and location of targets with 89% accuracy. This goal is accomplished by enriching the information provided by the accessibility API. The accessibility API, however, is still used to retrieve the location and size of the element that is below
the cursor during recognition. This information is enriched by taking the boolean difference between screenshots taken right before a button press event and a button release event. The bounding box around the differing pixels might be a potential target. Furthermore, the data is enriched with computer vision techniques, such as color matching and pattern matching.

Other traditional approaches, such as macro recorders, achieve automation without explicit developer support by recording and replaying low-level mouse and keyboard events. These approaches are, however, very sensitive to noise as the slightest change in visual appearance might render them useless.

Other attempts tried injecting code to record actions into an interface toolkit [26, 25, 66]. While these approaches might work for applications implemented in the injected toolkits, the usefulness for automating arbitrary applications is very limited as people typically use a wide range of applications which are implemented with several different toolkits. WADE, presented by Meng et al. [49], takes a similar approach. WADE injects a dynamically linked library (DLL) into a host program to retrieve the GUI hierarchy of that host program. While their approach looks very promising to retrieve user interface information without explicit developer support, injection of DLLs is limited to applications developed with the .NET framework. Because of this, a similar approach would be unable to reach our goal of automating arbitrary applications in a desktop environment.

While APIs and toolkit injection are useful for specific use cases in desktop applications when properly implemented, these approaches are not useful in automating web interfaces. However, for automating web interfaces, solutions, such as Chick-enfoot [16], have been proposed to expose a higher-level API for accessing and manipulating common web page elements, such as buttons. While limited to the Firefox web browser, several interesting systems have been built on top of Chick-enfoot [16], such as Koala [48] and later CoScripter [45], which focus on increasing productivity by enabling users to capture, share, automate, and personalize business processes on the web. While these systems are useful for the context they are proposed in, they only provide limited generalization options of the generated script while still requiring some configuration. As a result, similar systems might be unsuited for novice computer users. The generalization of scripts these systems offer are limited to injecting user-specific parameters, such as names and addresses, into predefined positions in the script while executing, to enable sharing of the script between users. This however comes at a cost, as users need to be able to specify a database containing these parameters, which might require some technical knowledge.

### 2.2.2 Pixel-based techniques

As discussed in section 2.2.1, most approaches suffer from severe limitations, as they require either explicit developer support or are targeted towards very specific types of interfaces or systems. As a result, pixel-based methods have been proposed to overcome the aforementioned issues. Pixel-based methods use computer vision approaches to inspect user interface information on visible areas of the desktop screen. While several pixel-based approaches have been proposed, the most noteworthy approaches are Sikuli [90] and Prefab [22]. Pixel-based methods, however, have their own inherent limitations. Firstly, they are sensitive to visual updates, requiring
Chapter 2. Related work

FIGURE 2.1: An example Sikuli script to move all office documents to the recycle bin [90].

re-specification after an application has changed its layout. They also can’t control or access information that is not visible on screen, such as items contained in a closed drop-down menu or hidden behind a scroll bar.

Sikuli [90] provides a scripting-like language in which users specify an automation script by using commands like `find`, `click`, and `dragDrop`. The parameters for these commands are specified with screenshots, as shown in figure 2.1. Figure 2.1 displays a script to move all visible office documents to the recycle bin. When executing a script, Sikuli uses pixel-based methods to match the specified screenshots with the currently visible desktop area. Matches are made with varying sizes and varying color schemes to enable the sharing of scripts between different screen resolutions and themes. After finding the target, which Yeh et al. reported to take less than 200msec on a 1600x1200 screen for a 100x100 target [90], the specified action is executed and the next step in the script is started. However, as shown in figure 2.1, specifying a Sikuli script requires knowledge of programming constructs, such as `for-loops`, `while-loops` and `if-tests`. Because of this, novices can’t use Sikuli without first learning basic programming constructs, which might be a cumbersome process. An attempt to overcome this issue and to make visual automation more accessible has been proposed by Alharbi and Yeh in Sikuli Slides [6], which uses PowerPoint slides to specify a sequence of screen actions, as shown in figure 2.2. Obviously, by removing programming constructs, not only the threshold is lowered, but also the ceiling.

Prefab [22], on the other hand, uses pixel-based methods to enhance user interfaces by adapting the behaviour of arbitrary widgets. While this might not seem relevant to the work presented in this thesis, the shared goal of both approaches is identifying targets and widgets on the screen. Prefab uses raw pixels to reverse engineer the interface structure. It accomplishes this by building a prototype for a specific widget, based on provided positive and negative examples. An example prototype for a Windows Vista button is shown in figure 2.3. After the prototype has been built, it can look for occurrences of said prototype on the desktop screen. Because Prefab’s goal is to real-time enhance widgets during use, and thus all occurrences of multiple widgets need to be identified, Dixon and Forgaty reported Sikuli’s 200msec is to slow for their goal [22]. To overcome this, Prefab first conducts a single pass over an image to identify all occurrences of features from the prototype library. Based on hypotheses, potential occurrences are identified. Afterwards, these occurrences are filtered and checked for validity to identify real occurrences. Prefab was used to re-implement Baudisch et al.’s Phosphor [10] based entirely on the pixels of an interface, showing the approach is fast enough for near instantaneous recognition of widgets.
Chapter 2. Related work

Figure 2.2: Sikuli Slides example.

Figure 2.3:Prefab prototype for Windows Vista button [22].
Dixon et al. proposed several improvements to pixel-based methods, building upon Prefab. One of these improvements is extracting content and hierarchy in pixel-based methods for reverse engineering interface structure [23]. In this work, Prefab is extended with hierarchical models for complex widgets, methods for extraction and interpretation of widget content, and methods for recovering content and hierarchy of an entire interface. In 2014, Dixon et al. further improved pixel-based methods with Prefab Layers and Prefab Annotations [24]. Prefab Layers helps developers write interpretation logic that can be composed, reused, and shared to develop pixel-based interpretation applications. Prefab Annotations enables annotating interface elements with metadata needed to enable runtime enhancements. While Prefab’s techniques work well for widgets and windows with clear borders, such as the Windows Vista button shown in figure 2.3, our tests with Prefab’s open source code [21] have shown that Prefab’s approach struggles with modern interfaces, such as the Spotify [1] interface shown in figure 2.4. The reason Prefab fails on modern interfaces is that these interfaces don’t have clear borders around each widgets or panel in an interface, which causes difficulties for Prefab to build a prototype. In addition, widgets may not be visible until a certain action is executed, such as hovering over a specific position or element. As a result, Prefab is unable to recognise widgets that are not visible when inspecting the interface.

While Sikuli [90] tends to be more robust on modern interfaces, Prefab [22, 23, 24] takes an advantage in recognition speed. However, both solutions have one significant weakness in common. Both Sikuli as well as Prefab only focus on programmers, providing a high ceiling in combination with a high threshold. Because of this, novices can find it difficult to specify their own pixel-based automation script. The approach presented in this thesis differentiates itself in this aspect, as our solution specifically focuses on novice computer users, abstracting the required knowledge to use both systems away from the end-user.

While Sikuli and Prefab are the most significant contributions to pixel-based methods, other research is also noteworthy. Help, It Looks Confusing (HILC) by Intharah et al. [39] focuses on providing non-programmers with a pixel-based tool. They try
to accomplish this with a learn-by-demonstration prototype, in which the user can specify a task script by demonstrating the task. After demonstration, the user needs to answer follow-up questions to resolve ambiguities. Intarah et al. [39] showed that their approach allowed non-programmers to specify a script in less time when compared to Sikuli slides. When resolving ambiguities, HILC uses what they call supporters. An example of supporters is shown in figure 2.5. In figure 2.5, blue boxes are positive examples provided by the user, red are negative examples, yellow are potential targets, and a green box indicates a user-provided supporter. While the user-provided supporter, shown in figure 2.5b, resolves ambiguities by specifying positive examples in the left column, the concept of supporters might not be clear to novice users. First of all, users have to learn and remember the meaning of every color code. In addition, it might not be clear to the user where to place the supporter. In contrast, our solution uses intuitive features to resolve inconclusiveness of the system when checking an automation, such as relative positions (above, below, left of, right of) or the position in a list (first item, last item). In addition, our system shows where an ambiguity occurs and tries to resolve this by asking human-readable questions, such as "select the icon that uniquely identifies the correct target". After specification, our system determines the relation between the target and the uniquely identifying feature by using several heuristics.

In follow-up work, Intarah et al. presented RecurBot [38]. RecurBot is a system to automatically detect recurring actions in a user demonstration and to later automate these actions. RecurBot addressed several challenges related to end-user specification of automation scripts. First of all, user-demonstrated loops are non-identical. Iterations might differ due to noise introduced by the machine or by the user. An example of noise introduced by the machine might be a message box appearing that is not related to the sequence, which requires an unrelated button to be clicked. Examples of user-introduced noise are delays between clicks, omitting steps, and accidentally clicking. Another interesting problem addressed by RecurBot is the identification of the iterator by only a few examples of a lengthy task. While our system does not focus on recurring tasks with slight differences, such as one or more parameters differing between iterations, the work presented in RecurBot is still helpful as they propose a human-in-the-loop system, which allows users to approve or, if necessary, modify actions.
2.2.3 Hybrid approaches

As the approaches presented in section 2.2.1 and section 2.2.2 each have their own advantages and disadvantages, Chang et al. proposed a hybrid framework, PAX [17], which associates the visual representation of user interfaces (i.e. the pixels) and their internal hierarchical metadata (i.e. the content, role, and value). They identified issues with both accessibility APIs as well as with using computer vision algorithms to analyse a user interface, and as such combined both approaches to make them more robust. The most important issues with accessibility APIs they identified are:

1. Indifference to visibility: the API doesn’t know whether an element is visible or not.
2. Incomplete support: similar to Hurst et al [36], Chang et al. found implementation of the APIs to be incomplete.
3. Inconsistent text: APIs might return internal representations of data instead of content shown on screen (e.g. different time formats).

Issues with pixel-based methods Chang et al. [17] identified are:

1. Visibility constraints: invisible information cannot be detected, something our approach focuses on.
2. Visual variations: dramatic interface variations can cause issues.
3. Exhaustive screen search: pixel analysis is an expensive operation, especially on high-resolution monitors.
4. Low-resolution text.

By combining both approaches, Chang et al. [17] developed a more robust recognition system that automatically gives the most accurate results from available resources.

2.2.4 Other platforms

Other notable research focused on automating and/or recognising actions in other environments as opposed to the general desktop environment, such as the browser, the smartphone and even in prerecorded videos. This section provides a brief overview of notable systems.

Firstly, an example of a browser-based recording-by-example system is ActionShot by Li et al. [46]. ActionShot is an extension to the Firefox web browser built on top of CoScripter [45] to record web browsing history on the level of interactions, as entering a value into a form field, turning on a checkbox, or clicking a button. While ActionShot is only able to record these actions in the Firefox web browser, our systems aims to record these interactions in arbitrary applications. ActionShot provides the user with useful insights into the recorded actions by providing a history view, as shown in figure 2.6.
Chapter 2. Related work

a) List Mode

<table>
<thead>
<tr>
<th>Command</th>
<th>Time</th>
<th>Page Title</th>
<th>Page URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>go to &quot;liuconf.org&quot;</td>
<td></td>
<td></td>
<td>liuconf.org</td>
</tr>
<tr>
<td>click the &quot;Conference Overview&quot; link</td>
<td>03:23</td>
<td>2009 International Conference</td>
<td><a href="http://liuconf.org/">http://liuconf.org/</a></td>
</tr>
<tr>
<td>enter &quot;The 2009 International Conference&quot;</td>
<td>03:24</td>
<td>SIGCMS</td>
<td><a href="http://campus.acm.org/">http://campus.acm.org/</a></td>
</tr>
<tr>
<td>enter &quot;Researchers and practitioners interested in the conference&quot;</td>
<td>03:24</td>
<td>SIGCMS</td>
<td><a href="http://campus.acm.org/">http://campus.acm.org/</a></td>
</tr>
<tr>
<td>click the &quot;Call for Papers&quot; link</td>
<td>03:24</td>
<td>2009 International Conference</td>
<td><a href="http://liuconf.org/every">http://liuconf.org/every</a></td>
</tr>
<tr>
<td>enter &quot;We distribute a Call For Papers &quot; link</td>
<td>03:24</td>
<td>SIGCMS</td>
<td><a href="http://campus.acm.org/">http://campus.acm.org/</a></td>
</tr>
<tr>
<td>click the &quot;back to previous screen&quot; link</td>
<td>03:24</td>
<td>SIGCMS</td>
<td><a href="http://campus.acm.org/">http://campus.acm.org/</a></td>
</tr>
<tr>
<td>click the &quot;<a href="http://www.acm.org/sigs/vol">http://www.acm.org/sigs/vol</a>...&quot;</td>
<td>03:25</td>
<td>SIGCMS</td>
<td><a href="http://campus.acm.org/">http://campus.acm.org/</a></td>
</tr>
</tbody>
</table>

b) Timeline Mode

Figure 2.6: ActionShot’s history view [46].

Ringer [9] is another web automation by demonstration system, focusing on non-programming users. Ringer creates a script from a user demonstration that interacts with a web page the same way a user would. In contrast to CoScripter [45], Ringer focuses on automating the user interface of a web page instead of the underlying implementation of that web page, because while underlying web page implementations change frequently, user-facing interfaces remain relatively stable. This is a benefit all pixel-based methods, including our approach, also have because they are all based on the user interface instead of underlying implementations. However, Ringer’s approach is based on DOM events (such as mouseup, andmousedown) being executed on DOM nodes, an approach we can’t take to achieve automation in arbitrary applications.

While aforementioned work focuses on automating web platforms, SUGILITE by Li et al. [47] focuses on automating smartphone interactions by demonstration. They accomplish this by using Android’s Accessibility API [30] to support automating arbitrary tasks in any Android application. By leveraging the demonstrated procedures and the user interface hierarchy, SUGILITE automatically generalizes the script from the recorded actions. SUGILITE learns how to perform tasks with different parameters and variations from a single demonstration. A lab study performed by Li et al. [47] suggests that users with little to no programming knowledge can successfully automate tasks on a smartphone using SUGILITE, suggesting that same user group can successfully use our system to automate desktop tasks. While SUGILITE’s approach seems promising, we can’t use the same approach as we cannot rely on an operating system wide accessibility API. In addition, our work differs from
SUGILITE’s approach as they use voice control to replay an action as opposed to our proxy interfaces.

In contrast to most of previously mentioned approaches, Waken by Banovic et al. [8] takes an approach where no information is available to extract usage information and interface structure from interface videos. This is a challenging task, as no information from cursor movements, mouse clicks, and accessibility APIs is available to aid recognition. They implemented computer vision techniques for cursor tracking and for identifying icons, tooltips, and menus. Interestingly, their approach does not require any application specific rules or templates, similar to our approach. In contrast, we do have access to cursor movements and mouse clicks, so we don’t have to apply the algorithms presented in this work.

2.3 Proxy interfaces

The concept of proxy interfaces maps controls of one interface to controls on another interface. While the work presented in this thesis maps digital user interfaces to physical proxy interfaces, related work focused on mapping physical interfaces to other physical interfaces or digital interfaces to other digital interfaces. While this work is only loosely related to our physical proxy interfaces for digital user interfaces, the work presented in this section provides useful insights. In addition, this section briefly mentions recent research on tangible controls, from which we gathered ideas to build our physical proxy interfaces.

An example of a recent research project which maps digital user interfaces to other digital user interfaces are User Interface Façades [78]. User Interface Façades provides a solution which allows end-users to adapt the interface of any application without coding. By doing this, they allow users to optimize a digital user interface to their liking, effectively providing digital proxy interfaces for digital user interfaces. They accomplish this by gathering info and replacing widgets through Accessibility APIs. An example of a façade is shown in figure 2.7. As shown in figure 2.7, the resulting façade allows the end-user to be more productive by having immediate access to frequently used options, and only hiding the infrequent options behind a dialog. While taking an entirely different approach, both User Interface Façades and our approach can be used to enable increased productivity.

Another research project providing proxy interfaces is RetroFab by Ramakers et al. [71]. RetroFab provides an end-to-end toolkit which allows end-users to retrofit an existing physical interface with another physical interface, allowing end-users to adapt the behaviour and layout of physical interfaces, effectively providing physical proxy interfaces for other physical interfaces. An example of a retrofitted interface by RetroFab is shown in figure 1.2. Ramakers et al. identify several interesting use-cases for this retrofitting physical interfaces. For example, RetroFab can be used to adapt to the evolving or custom needs of a specific, potentially impaired, user. A similar use case can be identified for physical proxy interfaces for digital user interfaces, as impaired users or their caregivers can configure the physical proxy interfaces exactly to their needs.

Other work focusing on adapting physical interfaces to the needs of an impaired
user is AMI [74]. AMI focuses on providing an adaptable physical music interface to support the varying needs of people with dementia. AMI provides several tangible controls to reconfigure a music player, so people with dementia can use the controls they remember and are best suited for them. AMI has three different categories of tangible controls, which are power components, tuning components, and volume components, as shown in figure 2.8. While we don’t need as fine grain control as AMI as our focus isn’t specific on people with dementia, the controls provided by AMI provide useful insights in which types of controls we need to provide.

VoodooSketch [13] is another research project providing what we identify as proxy interfaces. VoodooSketch lets users dynamically deploy controls on interactive surfaces. These controls can act as shortcuts to applications the user is interacting with. VoodooSketch lets the user build a palette of controls, combining both physical components, such as buttons and sliders, with sketched components, drawn by hand, as shown in figure 2.9. While this work looks similar to the work proposed in this thesis, several key differences can be noted. Firstly, while VoodooSketch focuses on interactive collaboration on large displays, our work focuses on desktop scale computers. Secondly, as noted by Block et al. [13], VoodooSketch requires explicit
Figure 2.9: VoodooSketch: combining physical controls with sketched controls [13].

developer support for implementation and configuration of the proxy.

Phidgets by Greenberg and Fitchett [32] also influenced the hardware aspect of our physical proxy interfaces. Phidgets are to physical user interfaces what widgets are to digital user interfaces, meaning they abstract and package input and output devices. While the goal of Phidgets is to package physical devices and their software so developers can focus on overall use, modification, and recombination of devices into a physical user interface, keeping in mind they are easy enough so an average programmer can program and extend them, our focus isn’t on system developers but instead on end-users. In contrast, our system uses the concept of bundling every control into a stand-alone box to abstract input and output devices into a single unit, resulting in a pre-built physical control requiring no hardware knowledge of the user.

To couple proxy interfaces to an action, we built upon PUCs presented by Voelker et al. [84]. PUCs are Passive Untouched Capacitive Widgets, a technique implemented to recognize passive tangibles on a capacitive touchscreen. Being passive means they require no power, and they can be detected reliably even when no user is touching them. In addition, they do not require any modification of the touch display, meaning they work out-of-the-box with the touch screens of end-users. Voelker et al. improved upon PUCs by proposing PERCs [83]. PERCs are persistent capacitive tangibles that ‘know’ whether they are currently on a capacitive touch surface or not. Since we don’t need persistent tracking, the work described in PUCs [84] is of higher relevance to us. As Voelker et al. [84] pointed out, there is only a limited number of widgets that can be encoded by using PUCs. Because of this, we incorporated capacitive sensing inside our physical proxy interfaces to sense which physical control is being touched when placed on a touch screen, to enable an arbitrary number of proxy interfaces to
be recognized on screen.
Chapter 3

System overview

3.1 Introduction

To address the research challenges presented in Chapter 1, a system consisting of both hardware components and software components was designed and implemented. The system was designed and implemented building on top of previous state-of-the-art research, which is presented in Chapter 2. This chapter presents a general overview of the system, without going in-depth on the design or implementation of the system. The designed system lets the user automate an action in the digital environment. After specifying an action, the user can couple one (or more) of the developed physical proxy interfaces to the action. After a physical proxy interface has been coupled, the user can interact between the digital environment and the physical world. This interaction can be either input or output, providing physical input to the digital environment, or providing output from the digital environment to the physical world.

Section 3.2 provides four example cases illustrating how the system can be used. Section 3.3 describes the workflow a user has to follow to use the system. Section 3.4 describes which types of user interfaces and which user interface elements are supported by the system. Section 3.5 provides an overview of the developed physical proxy interfaces. This chapter is concluded in section 3.6.

3.2 Example cases

3.2.1 Introduction

To illustrate the wide range of possibilities the toolkit presented in this thesis has to offer, this section provides four example cases. Section 3.2.2 describes a scenario for a user using the toolkit to control their favorite music player. Section 3.2.3 describes a graphical designer using the toolkit to enrich the experience in their favorite image manipulation program. Section 3.2.4 describes a person who wants to monitor their favorite mail client using the toolkit. Finally, section 3.2.5 describes a user monitoring a digital weather widget in the physical world.
3.2.2 Example case 1: Music player

A person wishes to use physical proxy interfaces to control their favourite music player. Instead of having to physically move to the computer to control the music player, physical proxy interfaces would allow them to control the music player from wherever they want in their house. The person wishes to automate three distinct functions in the music player. Firstly, they want to start their favorite song. In addition, they want the ability to pause and start the currently playing song. Finally, they want to be able to change the volume of the music player. To specify these actions, the person opens up the toolkit.

After opening the toolkit, the user starts a new recording. The user is greeted with a list of applications installed on their system, from which they can chose one or more applications, as shown in Figure 3.1. To specify how the system should start their favorite song, the user first tells the toolkit to open their favorite music player. After selecting an application, the application the user has chosen is opened by the toolkit. The user then demonstrates once which icons and buttons need to be clicked to start playing their favorite song. After demonstrating all the steps required to reach the favorite song, the user clicks on the button "End state reached", which is shown in Figure 3.1b.

As shown in Figure 3.1, the recording interface has five main components. Figure 3.1a shows a button which can be interacted with to start a new external application in which the user wishes to record an action. Clicking this button will reveal the panel shown in Figure 3.1e. This panel contains the applications installed on the operating system. When clicking an application, the panel shown in Figure 3.1e will
be replaced by the application that has been clicked, as shown in Figure 3.2, where the panel has been replaced with the Spotify application. Figure 3.1d shows a list of recorded actions, which is empty in the figure. During recording, the list contains the actions demonstrated by the user, which can consist of both typing and clicking actions. Figure 3.1b shows a button to indicate that the final widget in the automating sequence has been reached. Interacting with this button will open a menu to add the final widget, together with the actions performed to reach this action, to the toolkit. Finally, Figure 3.1c shows a button to close the application. As the recording interface is implemented as a full-screen application, the Windows taskbar is covered and the normal window controls, such as the minimise and the close button, are hidden. As a result, a close button is provided to allow the user to return to the operating system. Covering up the Windows taskbar, by making the application full-screen, makes sure the user can only start applications through the toolkit, and prevents direct interaction with the operating system. As the toolkit is full-screen, a spot has been reserved to show the external applications, as shown in Figure 3.2, where the user is recording an action in Spotify, which caused Spotify to open on top of the toolkit.

After the user pressed the "End state reached" button, shown in Figure 3.1b, the user can automate the final widget. To tell the system how the final widget should be controlled, the user chooses the type of the widget, and specifies how the widget is identified, by selecting the play button as the widget and the name of the song as the identifier, as shown in Figure 3.3. To specify which icons will be selected, the user can zoom an icon, which will shown an interactive enlarged version of the icon, as shown in Figure 3.4. If the enlarged widget proposed by the system does not cover the entire widget, or covers too much of the interface, the user is given the ability to resize the suggestion by dragging the corners of the magnifying window.

After the user has demonstrated how to start the favourite song, the systems checks the demonstrated steps. Possibly, the system asks one or more follow-up questions to
Figure 3.3: (a) Add a button action interface. (b) Add a action for the play button belonging to the song "It's A Risk".

Figure 3.4: An enlarged play button.
help resolving potential ambiguities. An example follow-up question presented to the user in this scenario is shown in Figure 3.5. This question is asked so the system can guarantee a robust playback of the sequence, as explained in Chapter 4. Finally, the user couples the physical button proxy control (Figure 3.16a) to the action by placing it on the enlarged play button on their touch screen.

After specifying how to start the favorite song, the user tells the toolkit they want to automate another action to play and pause a song. Again, they tell the system they want to record an action in their favorite music player. As the play and pause button are immediately visible upon opening the music player, they don’t need to demonstrate anything, and click immediately on the "End state reached" button, shown in Figure 3.1b. To tell the system the play and the pause button can switch between each other, they specify the action as a toggle widget, as shown in Figure 3.6. The user couples the physical toggle control (Figure 3.16b) to this button.

Finally, to control the volume of the music player, the user starts a new recording once more, again opening their favourite music player. Similar to the play and pause button, the volume slider is permanently visible in the music player, requiring no demonstration. After clicking the "End state reached" button, the user selects the slider widget in the interface, and shows the system how the empty and the full slider look, as shown in Figure 3.7. The user enlarges the slider and places the rotary potentiometer (Figure 3.16c) on the magnified view to couple it. After specifying the three actions, the user can interact with the physical controls to control their music player.

### 3.2.3 Example case 2: Image manipulation

Imagine a graphical designer who wishes to use physical proxy interfaces to enrich their interaction experience within their favorite image manipulation program. Using physical proxy interfaces in this case would stimulate two-handed use [29] and increase productivity, as direct control to some functions can be provided. The graphical designer wishes to automate two functions in the image manipulation program. The first function is to change the color of their brush between two colors.
The second function is to change the brush thickness. According to the definitions in section 3.4, the user can use a toggle widget to change between the two discrete color values. To control the brush thickness, the slider widget is the most appropriate widget, as the thickness can assume a continuous range of values.

Similarly to the scenario in section 3.2.2, the graphical designer demonstrates how to reach the settings for the brush thickness and brush color. Once done, the graphical designer couples the physical toggle (Figure 3.16b) to the color setting, and the physical rotary potentiometer (Figure 3.16c) to the thickness setting.

After coupling the physical proxies, the user can interact with them to adjust the brush settings while designing. As the toolkit can click very fast, simultaneous input from a physical proxy and the mouse is allowed. The system can perform its clicking action and return the mouse to where the user was working in such a short time, that the user doesn’t notice the control of the mouse was taken away from them.

### 3.2.4 Example case 3: Mail

This example case describes a user who wishes to monitor the number of unread mails in their favorite mail client. As the mail client from the user is browser-based, they tell the toolkit to open the browser, and they surf to the URL of the mail client. They then specify which widgets in the interface they want to monitor by enlarging them in the toolkit and specifying them in the "Add Monitor Action" window, shown in Figure 3.8. As the number of unread emails is a textual variable, the user grabs the physical output interface containing the monochrome OLED screen (Figure 3.17b) and places it on the enlarged label.

After specification of the widget, the system places the browser window containing the mail client in the background, and starts monitoring the widgets of interest. Placing the browser window in the background is done so the monitoring does not interfere with the user using their computer. As the user coupled the monochrome OLED screen, the system knows it needs to convert the widgets to text, for which it uses its OCR feature. The system sends an update to the OLED screen every time the monitored value changes, after which the OLED screen starts continuously scrolling this value together with its label, as shown in Figure 3.9.
Figure 3.8: (a) Add a monitor action interface. (b) Add a monitor for the number of unread emails in Gmail.

Figure 3.9: A demo of the text monitoring feature to the physical output proxy. (a) The interface in Gmail [31] showing the label (Inbox) and the number of unread emails (14). (b) Showing the values extracted from (a) on a physical proxy interface.
Figure 3.10: A demo of the area monitoring feature to the physical output proxy. (a) The area of the interface which is being monitored, in this case a part from the weather widget from www.weather.com. (b) Showing the monitored area on the physical output proxy with the LCD screen.

3.2.5 Example case 4: Weather

Similarly to the user in section 3.2.4, the user in this example wants to monitor a part of an interface to the physical world. In this case, the user wishes to monitor a part of his weather application to the physical world. The user starts the process by choosing the application they wants to monitor in the toolkit, so the toolkit can open this application. After reaching the part of the screen the user wants to monitor, they click on the icon to enlarge it. By default, the toolkit enlarges the clicked icon. However, in this case, the user wants to monitor a part of the interface instead of a single widget. As a result, the user overrules the suggestion made by the system, and resizes it to contain the part of the interface they which to monitor.

The user couples the output proxy with the LCD screen (Figure 3.17a) on the enlarged area. As a result, the system knows it doesn’t need to extract the text from the image, but can just send the image to the proxy. Figure 3.10 shows an example of this scenario, where the www.weather.com weather widget, shown in Figure 3.10a, is sent to the output proxy with the LCD screen, as shown in Figure 3.10b.

3.3 Workflow

To specify and execute an action within the system, the user needs to follow a predefined sequence. When the user is executing the sequence, the system will act upon the actions taken by the user. The pipeline both the user and the system follow is illustrated in Figure 3.11. In Figure 3.11, every step is marked with either a user icon or a computer icon, which indicates whether the step is executed by the user or by the system, respectively. Figure 3.11a shows the steps the user and the system take in software. Chapter 4 provides implementation details about every step
Figure 3.11: The pipeline a user and system will follow when specifying an action in the toolkit. The user symbol indicates a step that the user executes. The computer symbol indicates a step executed by the system. (a) Indicates the steps in the pipeline which are handled or executed in the digital environment, and are thus implemented in the toolkit (brown colored steps) (b) Indicates the steps performed in the physical world (green colored steps). (c) Indicates steps which are only performed once, during specification. (d) Indicates steps which can repeat indefinitely, during everyday usage of the system.

illustrated in Figure 3.11a. Figure 3.11b shows the steps the user takes in the physical world, which deal with coupling and interacting with physical proxy interfaces. Implementation details about the two steps shown in Figure 3.11b are handled in Chapter 5. Figure 3.11 is split up into two parts, Figure 3.11c and Figure 3.11d. The steps illustrated in Figure 3.11c only have to be performed once for every automated action, during specification. The two steps illustrated in Figure 3.11d happen more than once, during daily interaction with the system.

The first step shown in Figure 3.11 indicates that the user needs to demonstrate the sequence. This means that the user has to show the system which steps they want to automate. The user can demonstrate these steps using a Programming-By-Demonstration paradigm. The actions demonstrated by the user include opening one or more applications, typing, clicking, and interacting with the final widget the user wishes to automate and offer a physical shortcut for.

After the user demonstrates a sequence, the system checks this sequence, which is the second step in the pipeline illustrated in Figure 3.11. This step is necessary because the system is implemented using a pixel-based approach for playing back an action. In a pixel-based approach, the pixels of the screen are matched with the pixels of the interface element which the user interacted with during demonstration to find the interface element on the screen. As a result, checking the sequence is necessary to resolve potential ambiguities, to guarantee a robust playback of the action. When the
system is unable to resolve the ambiguity algorithmically, a follow-up question is asked to the user, which is designed to aid in resolving the ambiguity. So the system can resolve potential ambiguities, which in turn guarantees a robust playback of the action.

After the user has demonstrated a sequence, and the system has verified the specified sequence, the system offers the possibility to couple one or more of the supported physical proxy interfaces to the action. This is the third step shown in Figure 3.11. To couple a physical proxy interface, an easy-to-use coupling paradigm is implemented. The user places the tangible control on a touchscreen, positioning it over the final interface element they want to automate, while holding on to the tangible. By placing the tangible onto the interface element, the system will use the positional information to allow richer interactions, so not only a physical slider can be coupled to a digital slider, but, for example, multiple physical buttons can get coupled to specific positions of a digital slider.

After the user has coupled a physical proxy interface to an action, step four and step five in Figure 3.11 are executed. Depending on the type of the tangible, these steps will be interchanged, which is the reason for the double arrow between step four and step five in Figure 3.11. When the tangible is an input proxy interface, the user initiates an action with the tangible proxy. Upon interacting with the tangible, the system will execute the recorded sequence in the digital environment. When the tangible is an output proxy interface, the system monitors for specific changes in the user interface, and transfers this output to the physical proxy components. These steps can repeat indefinitely once an action has been specified and a physical proxy interface has been coupled to it, as illustrated in Figure 3.11d.

3.4 Supported User Interfaces

3.4.1 Introduction

To remain application independent, the system is designed to support as many user interfaces as possible. This is one of the main reasons a pixel-based approach is preferred over operating system dependent or application dependent APIs, such as APIs for specific toolkits or the Accessibility APIs offered by operating systems [30, 53]. As the system is designed to support all different user interfaces, the user is given a list of all installed applications on their own operating system to choose from when specifying an action.

Once one of the installed actions is chosen from the list, the application is opened and the actions taken by the user are recorded to support the Programming-By-Demonstration paradigm. To support every interface, an image processing algorithm is developed to extract the interface elements the user interacts with. In addition, operating system wide keystrokes are recorded to track every keystroke. A combination of these two methods help in supporting a wide variety of user interfaces.

Once the user has reached the final interface element they want to automate, a differentiation between supported interface widgets is made to facilitate controlling
the widget during playback. The system differentiates between four different widgets, which are either input or output widgets:

1. Button: the most simple control of all supported widgets is the button control. A button widget is a widget which will perform an action on the screen by clicking on it. Figure 3.12 shows example buttons from a variety of real user interfaces, which are all supported by the toolkit proposed in this thesis. A button widget at the end of a sequence can be differentiated from all other items clicked during demonstration, by applying the definition that a button widget executes an action, while the other recorded clicks during demonstration are strictly for navigational purposes.

2. Toggle: a special case of the button widget. The difference between a button and a toggle is that while a button is stateless, a toggle is stateful. This means that interacting with a button will only result in an action being applied to the interface, while interacting with a toggle will not only apply an action, but also change the state of the toggle. In addition, an action applied by changing the state of a toggle can be easily reverted by toggling the widget to its previous state. However, reverting an action executed by interacting with a button widget might be more cumbersome, if not impossible. Figure 3.13 shows a variety of toggle widgets which occur in real-world interfaces.

3. Slider: a stateful input widget that has a continue state, instead of the discrete on and off values that a toggle can assume. Interacting with a slider will change the value of the state, which will typically trigger a setting to be changed. Reverting the slider back to its original value is possible by setting it back to the previous state. Figure 3.14 shows a variety of sliders taken from real-world interfaces.

4. Monitor: not a real widget. Instead, monitoring widgets are parts of an interface which have a special meaning to the user, and which they want to keep track of. In addition, while the button, the toggle, and the slider are input widgets, a monitoring control is an output widget. The difference between input and output widgets is that interacting with an input widget will provide input to the application where the widget resides. The application will handle upon the received input. However, an output widget is a widget residing in the application which will change according to the state of the application. For example, when an event happens within the application, the widget will change its appearance. To make this concept more concrete, Figure 3.15 shows two monitoring widgets from real-world user interfaces. Monitoring widgets can change both textually as well as graphically.

### 3.5 Supported Tangible Proxy Interfaces

#### 3.5.1 Introduction

To control buttons, toggles, sliders, and monitoring widgets in the physical world, a variety of physical proxy interfaces are developed. For every supported digital widget, at least one physical counterpart was designed. However, the user can be
Chapter 3. System overview

FIGURE 3.12: Example buttons from a variety of real user interfaces. (a) The forward and previous button in the Spotify [1] interface. (b) Three buttons in the Windows Steel Layout style, which was used in Windows Vista. (c) The compose and refresh button in the Gmail [31] interface.

FIGURE 3.13: Example toggles from a variety of real user interfaces. (a) Toggles found in the Spotify [1] interface. Left: a toggle between the play and pause state. Right: a toggle in the settings menu (b) Toggles found in the MacOS Mojave Settings. Top: a toggle in the form of a checkbox. Bottom: a toggle in the form of radio buttons. (c) Toggles found in the Gmail [31] interface. Top: the toggle to star or unstar a mail. Bottom: a toggle for a setting in Gmail.
Chapter 3. System overview

Figure 3.14: Example sliders from a variety of real user interfaces. (a) The volume slider found in the Spotify [1] interface. Top: the volume slider without hovering it. Bottom: the volume slider while hovering it. (b) Two sliders found in the MacOS Mojave Settings to change mouse or trackpad settings. (c) A vertical slider to set the volume on www.facebook.com.

Figure 3.15: Example monitoring controls from two real user interfaces. (a) The number of unread emails in Gmail [31]. (b) The weather widget from www.weather.com.
creative with the physical proxy interfaces, and combine multiple proxy interfaces to control a digital widget, or map a different type of tangible to a user interface widget. For example, consider the case where the user wants to control a digital slider in the physical world. This can either be done by coupling the physical slider to the digital slider, or by coupling a set of buttons to the slider. For example, one button can be mapped to 0%, one to 50%, and one to 100%. This allows the user to specify richer interactions between digital widgets and physical proxy interfaces. An example of a one-to-one mapping with where the physical and digital widgets don’t match is when a physical slider is coupled to a digital toggle component. In this case, controlling the physical slider to 0% turns the toggle off, while 100% will turn the toggle to its on state.

Section 3.5.2 discusses the set of input proxy interfaces. These input proxy interfaces are designed to control digital buttons, toggles, and sliders. Section 3.5.3 describes the set of output proxy interfaces, which are designed to interact with digital monitoring widgets in the physical world.

### 3.5.2 Input Proxy Interfaces

To control every supported digital input widget from the physical world, a possible physical proxy counterpart for each widget was designed. Figure 3.16 shows the set of the developed physical input proxy interfaces.

Figure 3.16 shows three developed input proxy interfaces, which are directly mapable to the supported digital widgets discussed in section 3.4. Figure 3.16a shows a physical proxy for the button widget, which is a physical push button. Figure 3.16b shows a physical proxy for the toggle widget, which is a physical toggle. Finally,
Figure 3.16c shows a physical proxy for the slider widget, which is a physical rotary potentiometer.

While the set of physical proxy interfaces, shown in Figure 3.16, shows one possibility to map to every supported digital widget, the set is by no means complete. It is easy to design additional physical components mapping to the same functions. For example, one could control a digital toggle with a physical switch, or use a linear potentiometer to control the slider widget.

### 3.5.3 Output Proxy Interfaces

To proxy digital monitoring widgets to the physical world, two examples of physical output proxy interfaces were created. Figure 3.17 shows the set of fabricated output proxy interfaces, which is, as was the case with the input proxy interfaces, shown in Figure 3.16, by no means a complete set of possible physical output proxy interfaces.

Figure 3.17 shows two examples of output proxy interfaces. Figure 3.17a shows an output proxy containing a 3.5" LCD screen with a resolution of 320x480 pixels [87]. This proxy interface can, for example, be used to monitor rich screen information, such as the weather widget, which is shown in Figure 3.15b. The second output interface, which is illustrated in Figure 3.17b, contains a Monochrome 1.3" OLED screen with a resolution of 128x64 pixels [4]. This proxy interface can, for example, be used to monitor textual screen information, such as the Gmail example shown in Figure 3.15a.

As previously stated, the set of output proxy interfaces, shown in Figure 3.17, can be easily extended. For example, one could want a output proxy interface containing one
or more LEDs which can illuminate based on the monitoring widget. For example, when a new mail arrives, the proxy interface flashes its LED light(s).

3.6 Conclusion

This chapter provided an overview of the implemented system. It started with four real-world scenarios in which the proposed system can be used. Afterwards, a description of the workflow a user follows when interacting with the designed toolkit was provided. In addition, the supported digital user interfaces were described. This description showed the system is designed to support a wide variety of real world user interfaces. Next, the created physical proxy interfaces were shown. These proxy interfaces provide a starter set which allow every widget to be manipulated, but they form by no means a complete set. In the description of the created physical interfaces, suggestions for expanding the set were provided.
Chapter 4

Implementation of the software toolkit

4.1 Introduction

4.1.1 Programming language and system setup

The software toolkit is the primary piece of software the user interacts with to record actions and to couple tangible controls to these actions. This chapter will go into detail about the implementation of the toolkit.

As the toolkit requires native support from the operating system to make certain features possible, such as tracking global mouse and keyboard input, the choices for the programming language were limited. To be as accessible as possible to the target users, an implementation for the Windows operating system was preferred, as Windows has a nearly 80% market share, while OS X combined with Linux distributions account for nearly 16% of the market share in March 2019 [20]. In addition, Windows was preferred due to the available APIs and features, which facilitate the development of the software toolkit.

As a result of the choice for Windows, C# and the .NET framework were preferred for implementation, due to prior knowledge and experience. C# is designed to be a simple, modern, type-safe, and object-oriented programming language [50]. In C#, it is possible to develop desktop applications, web applications, and mobile applications. C# offers several alternatives to develop desktop applications. Windows Forms [60] is the traditional approach for creating desktop GUI applications. A successor to Windows Forms is Windows Presentation Foundation (WPF) [51], which allows for more freedom in the designing of GUI applications, such as transparent backgrounds. In addition, WPF applications are designed to have a clear separation between GUI logic and application logic. The most recent GUI technology for Windows desktop applications are Universal Windows Platform (UWP) applications [59]. Both WPF applications as well as UWP applications use Microsoft’s Extensible Application Markup Language (XAML) [61] to specify GUI components, as opposed to Windows Forms, which uses a What-You-See-Is-What-You-Get (WYSIWYG) GUI paradigm. The difference between WPF and UWP, however, is essential. UWP applications are sandboxed applications, designed with security in mind. As a result, UWP applications require the user to give special permissions to access specific features,
such as writing to the file system. To overcome fiddling with user permissions, a WPF application was preferred for implementing this toolkit where possible. The only functionality implemented in an UWP application is explained in section 4.3.3.2, which uses an UWP specific API.

The toolkit has been developed and tested on a Windows 10 Virtual Machine using VMWare Fusion 10 on a macOS Mojave host. The virtual machine was allocated 2 cores of the Intel 7700HQ processor and 8GB of RAM. At least windows build 1803 is required for the functionality described in section 4.3.3.2. The toolkit has been developed and tested on a Windows 10 Education license on build 1809.

4.1.2 Motivation and chapter overview

To bring structure in this otherwise overwhelming chapter, this chapter will be structured according to the pipeline a user will follow in the digital world when specifying an action in the toolkit. This pipeline is illustrated in Figure 3.11a. As a pixel-based reverse engineering approach is used for a lot of features discussed in this chapter, this section describes a brief overview of the advantages and disadvantages of these approaches, which serve as motivation for most of the features described in this chapter.

One of the current state-of-the-art pixel-based reverse engineering approaches is Sikuli [90]. Sikuli provides, amongst other things, a visual scripting API with which users can specify an automation script using visual screenshots and programming structures, such as loops and conditional statements. However, to write a script that not only works at specification time, but keeps working in the future after a user interface has dynamically adapted, special attention by the user is required. The user is required to anticipate to all possible changes to make sure the script can be played back both now as well as in the future. However, it is very hard to anticipate all possible dynamic changes of an interface. In addition, it might be cumbersome, or even impossible, to visually specify certain interface behaviour, as some widgets might not appear until interacted with. To illustrate the problems with pixel-based reverse engineering approaches, we consider the following example. Imagine a user who wishes to automate an action to start a song from their list of “Favorite songs” in their favorite music player. During specification, the list only contains a few songs, causing all songs to fit into the available screen area. The user builds a script to open the list of favorite songs and to click the play button corresponding to the song they wish to start playing. After testing the script, the user determines it is working. However, in the future, the user adds a song to the list of favorite songs with the same album name as the name of the song they want to play. When replaying the script, the script finds two matches for the song name, both having a corresponding play button. In this case, the script is not able to determine which play button should be chosen, and chooses the first one it finds while searching the pixels of the screen, which might or might not be the right one, and which might differ every execution. After the user has finally determined the problem, they fix the script to add additional specifications to resolve the ambiguity. In the future, as more songs get added to the list of favorite songs, the automated song gets removed from the viewport, and hidden behind a scrollbar. As the user did not anticipate this, the script is broken once again, as scrolling is required to find the right song. However, the scrollbar only becomes visible upon scrolling the area, as is the case in many modern interfaces, further
complicating the specification of the scrolling behaviour in the script. However, pixel-based reverse engineering approaches also have some significant advantages. The main advantage of pixel-based reverse engineering approaches is that, with the needed care, they can automate any interface, statically or dynamically, without requiring any knowledge about the internals of the application, which is why the work presented in this thesis builds upon pixel-based approaches.

To overcome the issues presented in the example, the pipeline shown in Figure 3.11 has been proposed in this thesis. The first step in the interaction is the demonstration of the sequence to specify an action. The details for this functionality are described in section 4.2. After the user has demonstrated the actions, the system will check this sequence and ask follow-up questions to resolve ambiguities, which is discussed in section 4.4. In this section, the issues illustrated in the "Favorite songs" example are dealt with, either algorithmically or by suggesting improvements to the user. Finally, upon interaction with an input tangible, an action needs to be performed on the desktop, which is discussed in section 4.5. To robustly execute a recorded sequence in dynamic user interfaces, which can change appearance over time, the system follows a predefined flowchart. This flowchart is illustrated in Figure 4.1.

As shown in Figure 4.1, the system controls the final widget without any additional steps when it is immediately visible. This is the ideal scenario, as it does not require any additional steps, so the execution is as fast as possible in this case. However, when the final widget is not immediately visible, the system falls back to try to recover to a known state. If the system needs to recover to a save state, the demonstrated actions of the user get replayed, as shown in the flowchart in Figure 4.2, which shows
how every step in the recorded sequence gets replayed.

As both executing and checking a sequence requires the system to find an icon on the screen, this functionality will be described in section 4.3. As this functionality is not a step that fits in the pipeline, but more a dependency for other tasks, Figure 3.11 utilises a different layout for this functionality. Within each section, a chronological structure will be applied as well. Finally, section 4.6 will conclude this chapter.

4.2 Recording the sequence

4.2.1 Introduction

To automate an action in the toolkit, a Programming-By-Demonstration approach is leveraged. This section discusses how a user specifies an action they want to automate in the toolkit. The user interface which the user interacts with to specify an automation is shown in Figure 3.1. To allow the user to automate arbitrary actions installed on their computer, all applications need to be managed by the toolkit proposed in this thesis. Section 4.2.2 discusses how this is accomplished. To get a sense of which actions the user is demonstrating, system-wide user input needs to be tracked. Section 4.2.3 discusses how user input is tracked to record which actions
are being performed. Section 4.2.4 handles how the clicks, performed by the user, are used to identify which icons the user has clicked. As discussed in section 4.2.4, clicks performed to the buttons shown in Figure 3.1 are not added to the sequence, to produce correct results. Finally, after the user is done demonstrating, they can add the action to the toolkit, as discussed in section 4.2.5.

4.2.2 Managing applications

4.2.2.1 Introduction

The first step in automating a sequence of actions is to specify in which application a demonstration should be recorded. To make this as accessible and as flexible as possible, a predefined list of supported applications can not be used, as this would result in limited support. Another approach is to either let the user interact with the operating system directly to open applications, but this complicates tracking the user even further, as this allows to much freedom on behalf of the user. For example, the user could use shortcuts, the start menu, the task bar, desktop icons, and possibly other ways to open an application. In addition, the system is unable to identify which clicks are used to open an application, and which clicks are used for other purposes. As a result, the system would have no notice of which applications need to be opened during playback, while this is required. This is required because, when the system is unable to execute an action immediately, the system will restart the application to reset it to a known safe state. As a result, the toolkit needs to manage the applications which the user wishes to interact with. Managing these applications is possible by, for example, providing a text box where the user can fill in the name of the application they wish to automate. However, this would cause issues when, for example, a novice computer user doesn’t know the exact name of an application. As a result, the toolkit provides an overview of installed applications on the system, which will dynamically adjust when ran on another system. This allows every installed application on every system to be automated.

Section 4.2.2.2 discusses how the installed applications can be collected from within a C# application. Section 4.2.2.3 introduces how an application can be started from C# code. Section 4.2.2.4 discusses how an application window which has been started can be moved and focused. Finally, section 4.2.2.5 handles how externally opened applications can be closed.

4.2.2.2 Finding installed applications

Within Windows, a list of installed applications is collected with the PowerShell command `Get-StartApps`. This command returns the full name and the application ID of every application installed on the operating system. This command returns the application ID in two different formats. The first format, for traditional applications, is a string specifying the ID. An example is `Microsoft.InternetExplorer.Default` for Internet Explorer. The other format, used for modern Windows UWP applications, such as Microsoft Edge, looks like `Microsoft.MicrosoftEdge_8wekyb3d8bbwe`
Listing 4.1: PowerShell: Get installed applications with install locations.

```powershell
c$startapps = Get-StartApps
ForEach ($app in $startapps) {
    $app | Add-Member InstallLocation Done
    $app.InstallLocation = ''
    if ($app.AppID -like '*!*') {
        $package = $app.AppID.Split("!",[StringSplitOptions]'
            'RemoveEmptyEntries')[0]
        $temp = (Get-AppxPackage
            | Where { $_.PackageFamilyName -eq $package }
            | Select-Object InstallLocation
        )
        $app.InstallLocation = $temp.InstallLocation
    }
}
return $startapps
```

!MicrosoftEdge. Note the exclamation mark in the latter example, which differentiates both formats from each other. Both types of application IDs will be treated slightly different throughout this section.

To make the UI for starting an application more user-friendly, both the application name and the application icon are shown, when available. To get access to the application icon, the executable for each of the applications needs to be located. For all applications, the executable is stored in the install location of the respective application. Querying the install location for modern Windows UWP applications is straightforward. However, querying the install location of traditional applications is slow, as it takes up to 1 second to receive an answer, per application. As a result, only UWP applications get their associated icon shown, while traditional applications get a default icon in the toolkit.

To get access to the install location of UWP applications, the Get-AppxPackage PowerShell command is used. To eliminate overhead for communicating between the WPF application and PowerShell, a script has been built. This script combines the Get-StartApps command and the Get-AppxPackage command in one single call to PowerShell for all installed applications. The script is shown in Listing 4.1.

In Listing 4.1, the Get-AppxPackage command is called for each application which contains an exclamation mark in its application ID, and from the Get-AppxPackage command, the InstallLocation is returned. This location is then appended to the application object. At the end of the script, all applications are returned with their name, their application ID, and, if found, their install location.

To run PowerShell commands from within C#, the NuGet package Microsoft.PowerShell.4.ReferenceAssemblies [65] is added to the toolkit. NuGet is the package manager for .NET application, and allows for adding and upgrading packages to a .NET application through their repository [64]. This package allows for executing PowerShell commands by adding command to a pipeline, and then invoking the pipeline. Invoking a successful PowerShell command will result in a list of
Chapter 4. Implementation of the software toolkit

4.2 Starting a process from C#

```csharp
Process process = new Process();
process.StartInfo.FileName = "chrome.exe";
process.StartInfo.Arguments = "-n";
process.Start();
```

Listing 4.2: Start a process from C#.

PSObjects. Each PSObject contains the application name, application ID, and install location for one installed application in its members.

After acquiring the install location, the folder is scanned in C# to find the executable file within the folder. This is straightforward, as the `Directory.GetFiles()` function in C# returns every file in a directory. After this, the location of the executable is known, which can be passed to the `Icon.ExtractAssociatedIcon()` function, which extracts an `Icon` object from an executable file. Finally, this `Icon` is converted to an `ImageSource` object, and shown in the user interface.

4.2.2.3 Starting an application

C# has a built-in way for starting and managing processes. For example, Listing 4.2 shows a built-in way to start the Google Chrome web browser by specifying the executable.

However, the approach shown in Listing 4.2 only works for specific executables, which are located in the search directories of the `Process` class in C#. As a result, a more flexible way to start processes is needed. However, access to the process object of each started object is still needed to manage the application after it has been started.

The implemented solution differentiates between the two types of applications, as discussed in section 4.2.2.2. With the PowerShell command `Start-Process`, an application can be started. For traditional applications, the name of the executable is used as a parameter for this command. However, for modern UWP applications, the command for starting an application from PowerShell will look like `Start-Process shell:AppsFolder\ plus the application ID`. When specifying the `-PassThru` parameter to the `Start-Process` command, the newly started process(es) are returned, and casted to a `Process` object in C#. However, this `-PassThru` parameter only works for traditional applications. For modern UWP applications, the install location of an application is used. This location can be acquired as described in section 4.2.2.2. The install location can be combined with the `Get-Process PowerShell command to obtain the processes corresponding to the application.

4.2.2.4 Moving and focusing an application

As pointed out in Chapter 3, the toolkit opens external applications on top of itself. To do this, a transparent area in the toolkit is used, where the applications get placed. As shown in Figure 3.2, the toolkit has a sidebar with a width of 500 pixels and a header with a height of 100 pixels. To remain independent of the resolution, and thus
Chapter 4. Implementation of the software toolkit

Moving windows of other applications in C# requires a call to the unmanaged Win32 Dynamic-link library (DLL). The Win32 DLL, and other unmanaged DLLs, provide access to low-level operating system functions, such as managing process windows, tracking the keyboard, and others. Calling a function in an unmanaged DLL from managed C# code, requires the correct signature. Most signatures are properly documented on the community wiki pInvoke.net [70].

Moving the application to the transparent panel in the application requires several unmanaged calls. First of all, the handle of the window needs to be found, as the process ID is not sufficient. Firstly, the handle to the desktop window needs to be found, with the unmanaged function GetDesktopWindow(). Next, all the child windows from the desktop window need to be acquired, which is done with the EnumChildWindows() function. Finally, the process ID of every child window is compared with the process ID for which the window is needed, which is accomplished with the GetWindowThreadProcessId() function.

After the handle for the correct window is known, it is placed in the appropriate place and focused. Moving the window to the right place is accomplished with the unmanaged function SetWindowPos(). For focussing the process, the SetForegroundWindow() function is used. Listing 4.3 shows the signature of these functions.

As shown in Listing 4.3, the SetWindowPos() function takes a parameter to specify at which z-index the window should be placed (hwndInsertAfter). As documented in [57], specifying -1 will place the window above all windows, and thus above the recording toolkit, resulting in the layout shown in Figure 3.2.

4.2.2.5 Closing an application

The final step in application management is closing the application. Closing an application happens at several points in the process. First of all, beginning the recording starts with killing all open instances of the application a user wishes to record in. This happens because every recording needs to start from a known begin state, which cannot be guaranteed when the application isn’t closed and then reopened at the beginning of a recording. Secondly, closing an application happens at the end of a recording, to reset the application back to the known safe state. This
Chapter 4. Implementation of the software toolkit

happens so that the system, as described in section 4.4, can check the sequence beginning from the same state as where the user started recording. Finally, closing the application happens during the executing of a sequence, as described in section 4.5, to reset the application to the safe state and replay the sequence when the interface element is not found immediately.

Killing an application from C# is a straightforward procedure, as access to the Process objects corresponding to an application is already handled in section 4.2.2.3. For each Process object, the Kill() function can be called to kill the corresponding process.

4.2.3 Tracking user input

4.2.3.1 Introduction

In order to record the actions a user is performing within another application, mouse and keyboard events need to be tracked. However, as the user is not uniquely interacting with the toolkit, but also with third-party applications in which they want to record a sequence, tracking input within the toolkit is not sufficient. Instead, global mouse and keyboard events need to be handled.

Getting access to global mouse and keyboard events requires functions from the Win32 unmanaged DLL. Section 4.2.3.2 will explain how global mouse events are tracked. Section 4.2.3.3 will handle global keyboard events.

4.2.3.2 Tracking mouse input

For tracking global mouse events, a MouseHook class is implemented, which uses native functions from the Win32 APIs to get access to global mouse events. With the unmanaged function SetWindowsHookEx(), a hook is defined. The signature for this function is shown in Listing 4.4.

As documented in [58], specifying the value 14 for the parameter idHook will install a hook procedure that monitors low-level mouse input events. The second parameter, lpfn, will be the C# callback function, which is called upon a low-level mouse event.

This callback function provides a parameter containing the mouse state. This parameter specifies the type of mouse event, and can contain the following values:

- Left mouse button down;
- Left mouse button up;
• Mouse move;
• Mouse wheel;
• Right button down;
• Right button up.

However, the callback function does not provide access to the mouse position by default. As a result, the unmanaged function GetCursorPos() is used within the callback function to get access to this position.

After gathering the necessary parameters within the callback function, a C# event is invoked containing the acquired parameters. The tracking classes within the toolkit can subscribe to this event, and handle correctly upon receiving an event.

Finally, within the callback function, the CallNextHookEx() function needs to be called. This function will pass the intercepted hook to other listening hooks. When not doing this, the mouse event will get absorbed in the C# application, and not reach the operating system. This will cause the mouse event to not happen within the application which is being recorded.

4.2.3.3 Tracking keyboard input

To track which actions the user performs on a keyboard, the keyboard is tracked in addition to the mouse. Tracking both keyboard input as well as mouse input allows for richer demonstration, such as opening a browser and surfing to a URL. Depending on the tracked keystrokes, even richer interactions can be tracked, such as shortcuts to menu actions, or copy-paste actions shortcuts.

Tracking keyboard input takes an approach very similar to tracking mouse input, as described in section 4.2.3.2. A keyboard hook can be started by passing the value 13 to the idHook parameter of the SetWindowsHookEx() function, which is shown in Listing 4.4.

The callback function will have access to the key being modified, and whether it is a key-up or key-down event. However, the callback function does not have access to whether any modifier keys are pressed. As a result, the GetKeyState() function is used to add this information to the event. With this function, it is for example possible to check whether or not the Shift or CapsLock buttons are pressed.

Similar to section 4.2.3.2, key-up and key-down events are defined and invoked within the callback function, to which the tracking classes are subscribed to. Also, the CallNextHook() function is called, to pass the keyboard event to the next hook in the chain.

In the tracking class, the key-up event is subscribed to. Subscribing only to one event of the two is sufficient to track keyboard input of the user, as both events receive the modified keys. Subscribing to the key-up event guarantees that the key is released, and thus that it has effectively been typed. Subscribing to the key-down event might yield small errors, in the case the user presses a key and doesn’t release
Chapter 4. Implementation of the software toolkit

LISTING 4.5: Converting key events to a typed action.

```csharp
private void OnKeyUp(KeyEventArgs e)
{
    string typed;
    if (e.KeyCode == Keys.Space) typed = " ";
    else typed = e.KeyCode.ToString();

    if (typed == "OEMPERIOD") typed=". ";
    if (typed == "RETURN") typed="[enter] ";

    if (e.Shift) typed = typed.ToUpper();
    else c = typed.ToLower();

    if (lastAction.GetType() == typeof(TypingAction))
        add typed string to last action;
    else
        create new action with typed string;
}
```

until after recording. In the key-up event, the logic to extract the typed characters is implemented. The logic implemented to keep track of all typing actions performed by the user is more complicated than necessary. However, this is done to facilitate playing back the typing action as described in section 4.5.3.3.

The algorithm to track the typing actions of the user first checks whether the previous action was also a typing action or whether it was a clicking action. If the previous action was a clicking action, a new typing action is created. If the previous action, however, was a typing action, the newly typed key is added to the previous typing action. This results in one typing action containing all the text typed at a specific text area before clicking again.

To facilitate the playback of the typing action, as described in section 4.5.3.3, the keys received in the key-up event are converted to a string which described what the user typed. To handle the possible return values of the hooked key, some logic is implemented to build the string. If the received KeyCode object is a Space, a space character (" ") is added to the string. In all other cases the string representation of the KeyCode object, which is extracted with the ToString() method, is processed. If the user typed a period ("."), the string representation returns the value OEMPERIOD. When this string is found, the string representation is replaced with a "." character. When the user pressed on the enter key, the string representation is RETURN. In this case, the string representation is replaced with the string "[enter] ", to facilitate playback. In all other cases, the string representation will contain the pressed key (e.g. pressing the key "e" will return the value "e"). After the string to process has been extracted, the KeyEventArgs passed to the key-up event are examined to determine whether the user was pressing the shift key or not. If the shift key was pressed, the string representation is converted to upper-case, else it is converted to lower-case. Finally, the string representation of the typed key is added to the last typing action, or a new typing action is created containing the string representation. The pseudo-code for this algorithm is listed in Listing 4.5.

When a user surfs to the URL `www.google.com` in a browser and then presses enter, without performing any clicks in between, the code in Listing 4.5 will add the string
"www.google.com[enter]" to a single typing action. This string can then be parsed to playback the action, as described in section 4.5.3.3.

### 4.2.4 Identifying clicked icons

#### 4.2.4.1 Introduction

An essential part in any Programming-By-Demonstration system is identifying the actions performed by the user. Section 4.2.3 described how to get access to low-level mouse events. However, it is necessary to get information about the widgets the user is clicking in. As discussed in chapter 2, several approaches for acquiring this information have been proposed. Possibilities are, for example, injecting code into the target application or using accessibility APIs.

However, to be application independent, another approach has been taken. As previously mentioned, the work proposed in this thesis builds upon pixel-based reverse engineering methods [90, 22] to obtain an application independent implementation. Pixel-based approaches rely solely on the visible appearance of an application, and not on application specific toolkits or APIs. As a result, no support from an external application developer is required to add supported applications to the toolkit presented in this thesis. To support pixel-based reverse engineering within the toolkit, a screenshot is taken after every mouse click. This screenshot is then processed to find the widget the user interacted with when a mouse event occurs. Section 4.2.4.2 discusses how screenshots are taken from within C#. The steps taken to extract an icon from a screenshot are described in section 4.2.4.3. As clicks executed in the recording interface, shown in Figure 3.1, are not part of the sequence to be recorded, these clicks need to be ignored. Ignoring clicks delivered to the application is discussed in section 4.2.4.4.

#### 4.2.4.2 Taking screenshots

In order to take a screenshot from C#, a `CaptureWindow()` function is created. This function takes a handle to a window as input, and returns a C# `Image` object containing the current representation of the window. By passing the handle of the desktop window, as returned by the unmanaged function `GetDesktopWindow()`, a full-screen screenshot is obtained.

Within the `CaptureWindow()` function, functions from the Graphics Device Interface (GDI) API are used to copy the screen pixels to a bitmap. These functions reside inside `gdi32.dll`, and can be called similar to functions in `user32.dll`. The `CaptureWindow()` function is shown in Listing 4.6.

The `Image` object, returned by the `CaptureWindow()` function, as shown in Listing 4.6, is then casted to a `Bitmap` object, so it can be processed to find the clicked icon, as described in section 4.2.4.3.
```csharp
public Image CaptureWindow(IntPtr handle)
{
    // get the hDC of the target window
    IntPtr hdcSrc = User32.GetWindowDC(handle);
    // get the size
    User32.RECT windowRect = new User32.RECT();
    User32.GetWindowRect(handle, ref windowRect);
    int width = windowRect.right - windowRect.left;
    int height = windowRect.bottom - windowRect.top;
    // create a device context we can copy to
    IntPtr hdcDest = GDI32.CreateCompatibleDC(hdcSrc);
    // create a bitmap we can copy it to,
    // using GetDeviceCaps to get the width/height
    IntPtr hBitmap = GDI32.CreateCompatibleBitmap(hdcSrc, width, height);
    // select the bitmap object
    IntPtr hOld = GDI32.SelectObject(hdcDest, hBitmap);
    // bitblt over
    GDI32.BitBlt(hdcDest, 0, 0, width, height, hdcSrc, 0, 0, GDI32.SRCCOPY);
    // restore selection
    GDI32.SelectObject(hdcDest, hOld);
    // clean up
    GDI32.DeleteDC(hdcDest);
    User32.ReleaseDC(handle, hdcSrc);
    // get a .NET image object for it
    Image img = Image.FromHbitmap(hBitmap);
    // free up the Bitmap object
    GDI32.DeleteObject(hBitmap);
    return img;
}
```

LISTING 4.6: Capture a window by its handle from C#.
4.2.4.3 Finding the clicked icon

After capturing the screenshot, as described in section 4.2.4.2, it is processed to determine which interface element has been clicked. For image processing, Emgu CV [27] is used. Emgu CV is a cross platform .NET wrapper to the OpenCV [67] image processing library. Using Emgu CV allows OpenCV functions to be called from .NET languages such as C#. The wrapper was added to the toolkit by including the NuGet package `EMGU.CV` [19].

The first step to processing an image with Emgu CV is to convert it to an appropriate object. To accomplish this, the `Bitmap` containing the screenshot from section 4.2.4.2 is converted to an `Image<Argb, byte>` object. On this object, the image processing functions from Emgu CV are called.

The first step of the image processing is converting the image to a gray-scale image. This will make icon recognition more robust, as color information is not needed to extract the widget in a specified position. Figure 4.3b shows the result of converting the screenshot, which is shown in Figure 4.3a, to gray-scale.

The resulting gray-scale image, as shown in Figure 4.3b, will be used for further processing. On the gray-scale image, edge detection is applied to extract potential interface elements. A Sobel filter is used to extract edges from the gray-scale image. With the Sobel filter, a derivative of the image is computed in both directions. The result of the Sobel filter in the Y-direction on the gray-scale image shown in Figure 4.3b, is shown in Figure 4.3c.

After applying the Sobel filter on the gray-scale image in the Y-direction, resulting in the image shown in Figure 4.3c, a Sobel filter in the X-direction also needs to be applied to find all edges. Note that the Sobel filter needs to be applied on the original gray-scale image, and not on the resulting image. Figure 4.3d shows the result of adding the result of the Sobel filter in the X-direction to the Sobel filter in the Y-direction, shown in Figure 4.3c.

After acquiring all the edges by applying a Sobel filter in both directions, resulting in the image shown in Figure 4.3d, all the detected edges are highlighted to enable further processing. Highlighting the edges is accomplished by applying the `AbsDiff()` function to the resulting image. This function, when called with another color as parameter, computes the absolute difference between the image and the specified color. When applying the `AbsDiff()` function to Figure 4.3d with a black color, the result shown in Figure 4.3e is obtained.

Figure 4.3e shows an enhanced intensity for the edges when compared to Figure 4.3d. In this intensified image, the difference between lines is more obvious. Some edges are more white than others, which is the result of the Sobel filter certainty. Everything that is not completely white can be filtered as this is noise to the icon identification process. This filtering is accomplished by applying a binary threshold filter to the image shown in Figure 4.3e. Figure 4.3f shows the result of this operation, where only true white colours in the source image remain white, and the rest is converted to black.

Figure 4.3f shows all the user interface elements with the noise removed. To identify which icon or widget has been clicked, the bounding rectangles of the image shown
Chapter 4. Implementation of the software toolkit

Figure 4.3: (a) The original screenshot. (b) The screenshot converted to gray-scale. (c) A sobel filter applied to the gray-scale image in (b) in the Y-direction. (d) The result of a Sobel filter applied to the gray-scale image in (b) in the X-direction added to the same operation in the Y-direction as shown in (c). (e) the AbsDiff function applied to (d). (f) A binary filter applied to (e) to remove noise. (g) The bounding boxes extracted from (f), shown on top of the original screenshot shown in (a).
in Figure 4.3f are used. As a widget has a minimum size, bounding rectangles should have a height and a width of at least 5 pixels. This heuristic further reduces noise. Figure 4.3g shows the bounding rectangles for Figure 4.3f which are at least 5x5 pixels.

Finally, the bounding boxes are checked with several heuristics to determine which one is the most probable target. Out of all the rectangles which contain the clicked position, a combination of size and click centrality is used. Click centrality is used because the largest bounding rectangle would always be a panel or the application window containing the widget, and this is not the desired widget. However, rectangle size is also needed to extract the correct widget with enough context. For example, when zooming in on the play button in Figure 4.3g, multiple rectangles are placed inside each other. This phenomenon is illustrated in Figure 4.4.

When the largest rectangle around the play button, shown in Figure 4.4, is not chosen, only the triangle would be chosen for the play button. However, this would cause additional ambiguities when playing back or checking the sequence, as for example the forward and previous buttons also contain very similar triangles.

Lastly, the correct part of the screenshot (i.e. the correct bounding rectangle), is saved to the hard drive. Saving the icon is necessary because the image is needed for checking and playing back the sequence, as explained in section 4.4 and section 4.5.

4.2.4.4 Ignoring clicks delivered to the interface

The method discussed in section 4.2.4.3, identifies all clicked icons given a position of a mouse click. As the global mouse hook, which is explained in section 4.2.3.2, raises an event for every mouse click executed in the operating system, the toolkit would also add clicks which are delivered to the recording interface. This causes issues, for example, when the user clicks the New application or the End state reached button, shown in Figure 3.1a and Figure 3.1b respectively, during recording. These clicks need to be ignored as they are clicks required to start, stop, and further configure the recording sequence. To distinguish clicks delivered to the external application from clicks delivered to the toolkit, the VisualTreeHelper.HitTest() C# function is used. This function returns a result when a given click would hit a given
interface element. By hit-checking a click produced by the global mouse hook to the panels of the interface of the toolkit, a click can be properly distinguished and accordingly be processed or ignored.

4.2.5 Adding the action

4.2.5.1 Introduction

When the user reaches the final widget of the action they want to automate, they end the recording. After ending the recording, the user chooses between the four types of supported widgets, which are buttons, toggles, sliders, and monitoring widgets. The user has to specify the type of the final widget as every type requires a slightly different specification, and the system controls every widget in a different way. In future work, algorithms could be implemented which use, for example, machine learning, to automatically find out the type of the final widget. To specify the final widgets in the toolkit, some notion of unique features is required. These unique features are introduced in section 4.2.5.2. Section 4.2.5.3 handles the enlarging of an icon, which places the widget in a magnifier view, to allow the user to precisely specify the final widget or the corresponding unique feature. Specifying the final widget to the toolkit is discussed in section 4.2.5.4.

4.2.5.2 Unique features

Unique features are used to specify context to the final widget. Context is useful or necessary in different scenarios. As pixel-based recognition of interface elements oftentimes requires considering and recognizing the context around a widget in the user interface. The exact, or a very similar, user interface element could, for example, be present at multiple locations in a single interface, causing ambiguities. Oftentimes, a label or interface element around the element is then a unique identifier for that widget. Take, for example, the Spotify Interface, illustrated in Figure 4.5. To differentiate between the play buttons in the Spotify, a label or another widget is needed, which uniquely identifies the play button. A unique feature can, for example, be the name of the song corresponding to the play button.

In addition to uniquely identifying the play button, the unique feature also serves another key feature. As shown in Figure 4.5, the play button corresponding to a song is only visible when hovering over the row of the song. Having access to name of the song, it becomes straightforward to make the right play button visible, as the mouse can first be moved to the name of the song before going to the play button. This is not only needed in the specific case of the Spotify interface, but happens in a variety of modern interfaces.

When the icon and its corresponding unique feature are known, the toolkit calculates the relative position of both widgets. This relative position is used during playback to identify the correct widget. The relative position is calculated by comparing the positions of the bounding boxes of every widget. Relative positions are specified by orientations, which can be left, right, bottom, and top in the current implementation. These positions are sufficient in all tested scenarios, as most user interfaces are clearly
and strictly aligned. However, the paradigm could be easily extended to include positions such as top-left and bottom-right when deemed necessary.

While the relative orientation between two icons are currently the only supported unique features, which make it possible to specify a lot of existing actions, one could easily imagine other possibilities for unique features. For example, a user could wish to specify an action where the first song of the “Top 50” playlist will start playing. In this case, a position in a list would be the unique feature. This would allow a different song to play each time the action is being executed, as the first song in the list can change over time.

4.2.5.3 Enlarging an icon

When the user has reached the final icon they want to automate, they can enlarge it to further fine-tune the selection before adding it to the action. Right-clicking or pinching-to-zoom on an icon will show the enlarged version of the icon, as shown in Figure 3.4, where the play button shown in Figure 4.5 is enlarged.

The enlarging window is implemented with the Magnification API [52]. The open-source GitHub library karna-magnification [69] provides a C# wrapper for this API, which was adapted for the purposes of the toolkit presented in this thesis.

When right-clicking or zooming on an icon, the click or zoom position is used to identify which icon has been clicked. To allow identification of the icon, the method described in section 4.2.4 is reused. The rectangle identified by this method is then passed on to the adapted karna-magnification library, which shows the pop-up shown in Figure 3.4. This library accomplishes the enlargement of the icon by calling
functions from the DLLs `Magnification.dll` and `user32.dll`. They first create a window with the `CreateWindow` function, and then use the handle to this window to pass to the `MagSetWindowTransform` function, together with the magnification factor. Every time the user wants to place another icon in the magnified rectangle, a call to `MagSetWindowSource` is made to specify the new rectangle area to magnify.

The `karna-magnification` library has been adapted so it can also be used in scenarios where the system wrongly identifies the clicked icon. This can, for example, happen when one of the incorrect bounding boxes is chosen, as shown in Figure 4.3g. Another case where this can happen is when only part of the icon is selected, for example when an icon consists of multiple parts separated by empty space. To overcome these issues, the library is adapted so the magnification window can be resized and moved. In addition, the user can interact with the magnified icon the same way as with the normal icon. For example, when the user is clicking in the magnifier, the click will be delivered to the real icon, providing a virtual proxy interface for the icon.

Dragging the content in the magnifier will place other content in the magnified area. This is implemented by catching an event on mouse movement in the magnifier. When the mouse is pressed down, the magnified content is moved. To realize this, the magnified rectangle is continuously updated, by calling the `MagSetWindowSource` function.

To allow resizing of the magnified view, to select a larger or smaller part of the icon or the user interface, eight resize thumbs were added to the magnified window. One thumb is placed on each corner, and one thumb is placed in the middle of every side. Dragging these thumbs calls the `SizeChanged` event of the window. Within this event, a new rectangle is calculated and passed to the `MagSetWindowSource` function.

Finally, to deliver the clicks executed in the magnified view to the real interface, `MouseDown` and `MouseUp` events are subscribed to. Upon receiving an event, the corresponding position in the interface is calculated, and a click is performed at this location. Performing clicks is discussed in section 4.5.3.2.

### 4.2.5.4 Specifying the final widget

With the definitions of uniquely identifying features and the magnifying window introduced, the final widget can be specified in the toolkit. Selecting a widget for one of the fields in the user interface is done by enlarging to the magnifying view, if required adjusting the widget selected by the toolkit, and then clicking the button corresponding to the icon which the user is specifying. Clicking one of the buttons saves the screenshot currently magnified. This section discusses which fields can be filled in for every supported type of widget. Note that for all input widgets (i.e. the button, the toggle, and the slider), it is possible to specify a keyboard shortcut. Keyboard shortcuts will trigger the action in the same way as interacting with a tangible would. Specifying the supported widgets can be done as follows:

- **Button**: a button widget requires at the very least the icon representing the button to be specified. In addition, a unique feature can be provided, to resolve ambiguities.
• Toggle: as toggles change state when interacted with, and changing state oftentimes changes the appearance of the toggle widget, the appearance of both states need to be specified by the user. Knowing both the appearances will allow the system to interact with the toggle in both states.

• Slider: as a slider widget can assume a continue range of values, it can change appearance in an undefined number of ways. As it is not feasible to specify every possible appearance of the slider, the user is asked to specify only an icon representing a full slider, and an icon representing an empty slider. By setting the recognition certainty lower for slider, and using both the empty and the full representation of the slider, these two icons have proven to be sufficient for recognising the right slider during playback.

• Monitor: to specify a monitoring action, both the text to monitor as well as the unique feature identifying the text need to be specified. While for buttons, toggles, and sliders both images get saved to the disk for future reference, only the identifying icon gets saved in this case. The reason for this is that the text changes while monitoring, so searching this icon is pointless. Instead, the relative position between the identifying icon and the text icon is calculated, and is used to monitor the right region during playback.

4.3 Finding icons

4.3.1 Introduction

While finding widgets is not a step in the pipeline, shown in Figure 3.11, it is an important dependency for the checking and replaying steps in the pipeline. As the toolkit takes a pixel-based reverse engineering approach, the widgets specified in the actions need to be re-found on the screen to allow checking and playing back of the sequence. Finding the widgets is necessary because their position needs to be known to accurately play back a sequence.

To find a widget on the screen, several approaches have been taken. Initially experiments were performed with the Prefab [22, 23, 24] open source implementation [21]. Prefab was initially preferred over Sikuli [90] as Prefab has support for hierarchical user interfaces. This would ease development of several features of the toolkit, as knowing the hierarchy of a user interface would inherently resolve some of the ambiguities the toolkit now deals with. While Prefab’s implementation looks very promising, initial tests were not able to use its full potential in modern user interfaces. The reason for this is that Prefab builds prototypes to recognize widgets and build the hierarchy of a user interface. An example of a Prefab prototype was already shown in Figure 2.3. While this approach works for traditional user interfaces, where every widget and every panel has a clear border, as shown in Figure 4.6, it fails to reliably build an hierarchy in modern interfaces, such as the Spotify interface, which is shown in Figure 4.5.

In addition, as Prefab requires multiple examples to build a prototype, it works best for building up hierarchies for user interfaces with similar layouts, such as interfaces implemented in the Windows Steel Layout style, which is illustrated in Figure 4.6. As
Figure 4.6: A traditional interface in Windows Vista with clear borders around each widget and panel.

As a result, after training Prefab to recognise widgets and hierarchies which are implemented in a specific interface style, it will correctly recognise widgets and hierarchies in other user interfaces which are implemented with the same toolkits. However, the toolkit proposed in this thesis is designed to support arbitrary applications. As most modern user interfaces and websites are often implemented in different toolkits or different styles, Prefab would require training on every type of interface.

However, an experiment indicated training Prefab can not be automated. In this experiment, all returned potential icons in a screen, extracted with the method described in section 4.2.4, were provided as examples to Prefab. When subsequently rendering the hierarchy of the interface in Prefab, it would fail. Note that this test is run on the Spotify interface, with unclear separation between widgets. Running a similar test on the interface illustrated in Figure 4.6, might produce different results.

As Prefab didn’t fulfill the requirements for the toolkit, and adapting Prefab’s source code would be too complex, Sikuli [90] was preferred. Sikuli helps the toolkit to localise visible icons on the screen. This is accomplished by matching an image file with the visible pixels on the screen. When a match is found, Sikuli returns, amongst other
FIGURE 4.7: The architecture of the application. The C# application communicates with Java by using web requests. The Java web server localises images through an adapted Sikuli JAR.

data, the location of the match on the screen.

However, Sikuli is implemented in Java, while the toolkit presented in this thesis is implemented in C#. As a result, a bridge between both applications was built. A web server was implemented in Java. This web server is run locally on the same computer as the toolkit. The toolkit sends HTTP requests to the web server, containing, for example, a path to an image, and the server answers with a JSON object containing the information the toolkit needs. The server gathers this information by interacting with an adapted Sikuli JAR file. This architecture is illustrated in Figure 4.7.

Section 4.3.2 discusses how the architecture shown in Figure 4.7 is used to find icons visible on the screen. As shown in Figure 4.7, the Sikuli JAR which the server interacts with is not the standard JAR as found on the Sikuli website [34]. Instead, it is an adaptation of the open source Sikuli code [35]. Section 4.3.3 explains why an adaptation to the Sikuli implementation was necessary and how this is implemented.

4.3.2 Finding visible icons

4.3.2.1 Interacting with the web server

Sikuli [90] is designed to find the contents of a specified image in the visible area of a screen. However, as Sikuli is implemented in Java, some creativity is required to add its capabilities to the toolkit. As shown in Figure 4.7, a web server is implemented in Java with the Spark [76] library. This web server exposes some of the functions from Sikuli which are needed in the toolkit. The toolkit presented in this thesis communicates with the exposed methods of the web server through JSON, which is implemented in C# by using the Newtonsoft.JSON [63] NuGet package.
4.3.2.2 Exposed functions

To support the different use cases and scenarios of the toolkit, several GET methods have been written in Java. While some functions may seem similar, they differ in their return types. For example, one function returns the best match, while another function returns all matches. While it would be possible to do this filtering in C#, this would introduce overhead in multiple stages. Firstly, Sikuli would need more processing time to find all matches. In addition, there would be an increased communication overhead between the toolkit and the web server, as more data needs to be transferred. Finally, C# would require extra processing time to compare the scores of each match to find which one is the best. The exposed functions for finding visible icons are:

- **Find all the matching icon**: finding all icons matching to a specified icon is a function used mainly for resolving ambiguities, as discussed in section 4.4.2. To ask the user which icon they meant in the case of an ambiguity, all matching icons are required. Finding all matches in Sikuli is straightforward. This function takes the file path to the icon to search all the matches for. In Sikuli, all operations start with a `Screen` object. A `Screen` object has access to the method `findAll()`. When passing an image path to the `findAll()` function, a Java `Iterator` object of `Match` objects is returned. While iterating through the returned `Iterator`, the JSON object to be returned is built. For every `Match` returned, the X and Y coordinates are returned, together with the width and the height of the matching rectangle. In addition, the score of each match is returned.

- **Find the best matching icon**: finding the best matching icon is a function used to control the final widget. This function takes as parameter the file path of the image to search the best match for. In Sikuli, calling the `find()` function on a `Screen` object returns a `Match` object containing the best match. From this best match, the X and Y coordinates, together with the width, the height, and the score are returned in JSON format.

- **Find icon next to other icon with orientation**: finding an icon next to another icon is a function used to find icons in proximity to a distinguishing icon. The first icon, which helps to uniquely identify the second icon, is searched for with the Sikuli `Find()` function. The found region is then grown according to the provided orientation. Growing a region is accomplished by calling the `right()`, `left()`, `above()`, `below()`, and `nearby()` functions. These functions extend the found region to the respective direction, as shown in Figure 4.8. As the orientation between both icons is known, and passed to the exposed function, it is straightforward to know which function to use. In the extended region, resulting from one of the function calls, the second icon is searched with the `find()` function, resulting in a `Match` object. From this `Match` object, the X and Y coordinates, together with the width, the height, and the score are returned in JSON format.

- **Find text in icon**: a function for finding text in an icon has been exposed. This function checks if an icon contains text, and returns the found text. This function is used to overcome resizing issues, as explained in section 4.4.4. It uses Sikuli’s support for Optical Character Recognition (OCR), which is built on top of Tesseract [80]. To use the OCR feature in Sikuli, some configuration
for Tesseract is required. Listing 4.7 shows how to enable OCR in Sikuli with the parameters used in the toolkit. In Listing 4.7, a TextRecognizer object is created. On this object, the language is set to english (eng). This tells Tesseract which training data to use to perform the character recognition. The OEM and PSM settings define how Tesseract should find the text. OEM stands for OCR Engine Mode and can take four values: original Tesseract, Neural Networks, Tesseract combined with Neural Networks, and the default mode which selects the best based on what’s available. In Listing 4.7, the combination of Tesseract with Neural Networks is selected. PSM means Page Segmentation Mode, and specifies how the text will be structured in the image. In Listing 4.7, sparse text is selected, which has proven to work best in conducted tests. Finally, the character whitelist defines which characters should be recognised by Tesseract, which are all upper- and lowercase letters and numbers in Listing 4.7.

- Find text near icon: finding text near an icon is used when monitoring an interface element. For example, when the user wants to monitor incoming messages or the number of unread mail, this functionality is used. This function takes an icon and an orientation as input, and returns the text which is close to this icon in the specified orientation. To implement this function, the spatial functions as shown in Figure 4.8 are used in combination with the OCR functionality described in finding text in an icon. A real-world example how this function can be used is shown in Figure 3.9a, where specifying an icon containing the text "Inbox" and RIGHT as orientation will return the number of mails shown to the right of the inbox label.

- Find text on screen: the function to find text on the screen is used in combination with the function described in finding text in an icon to overcome resizing issues, as described in section 4.4.4. To function takes as input a string to search for and returns the position where this text is found on the screen if it is found. To obtain this position, the Sikuli Finder class is used. This class is used to find text on the screen with the findText() function, which returns an iterator of Match objects containing the positions where the text is found.
Chapter 4. Implementation of the software toolkit

4.3.3 Finding non-visible icons

4.3.3.1 Introduction

The functions described in section 4.3.2 are all implemented with the standard Sikuli API. As a result, they only work on visible pixels, which implies that everything has to be visible on the screen in order to be recognised. While this is an inherent limitation of traditional pixel-based methods, this thesis proposes a solution to overcome this problem, which is described in this section.

This functionality is useful in the toolkit to control or monitor things which are not visible on the screen, which allows things to happen completely in the background. This allows, for example, multiple monitoring controls to run at the same time, or to start a song in the Spotify interface without anything visibly changing on the screen. As this is not a trivial feature to implement, several dependencies are needed, which are discussed in the following sections. Section 4.3.3.2 discusses how the applications get hidden and how their interfaces can still be captured. Section 4.3.3.3 handles how pixel-based reverse engineering approaches can be used on the captured interfaces instead of on the visible area of a screen.
4.3.3.2 Screen capture of invisible screens

Capturing a screen in the background is implemented with the Screen Capture API [55] by Microsoft. This is a new API which was recently added to Windows and requires at least Windows 10 version 1803 to run. As it is a new API, which might impose security risks, it is only implemented for Universal Windows Platform (UWP) applications. As UWP applications are sandboxed applications, the security risks are mitigated. To build an application using the Screen Capture API, the application should have the Graphics Capture capability enabled in the Package.appxmanifest file.

As the Screen Capture API is only available for UWP applications, and the toolkit is written as a WPF application, the API cannot directly be used. To overcome this, a second application, in UWP, is written. This application is opened and interacted with from the WPF application code by using the PowerShell commands described in section 4.2.2. Within an UWP page, the OnNavigatedTo is called when the application is started from the command line. In this function, the command line arguments can be obtained, which is used to pass the filename for the screen captures from the WPF application to the UWP application.

After the application has been opened, the user is presented with a list of open applications, from which they needs to pick one, as shown in Figure 4.9. Clicking an icon gives the UWP application access to the frames rendered by the respective application. While it has been attempted to automatically click on an application, for example by searching for the name of the application in the text of a screenshot of the picker, this has turned out to be impossible. The reason for this is that when taking a screenshot of the application, from within the application, the picker is automatically removed from the screenshot, due to security reasons. As a result, these pixels cannot be accessed, and thus a manual step is required for the user.

After an application has been chosen in the interface shown in Figure 4.9, the Direct3D frames of the selected application are obtained in the UWP application. A code example to implement this functionality is provided by Microsoft [55]. This example has been adapted to save the captured frame, instead of showing it in the UWP application. To save images to the disk, an UWP application needs access to the File system, which needs to be set once in the setting of the application. After this permission has been granted to the application, the obtained Direct3D frames are converted to a CanvasBitmap object with the CreateFromDirect3D11Surface() function. This CanvasBitmap is then saved to the disk with the file I/O functions provided by UWP.

The Screen Capture API, however, can only access frames of applications which are not minimised. This would cause issues, as requiring the applications to not be minimised would not make them invisible to the user. To overcome this, the multiple desktop feature of Windows is used. For each application to be automated in the background, a separate desktop is created in which only the respective application resides. An example of this is shown in Figure 4.10, where Spotify resides in one desktop, and Gmail in the second.

Placing every application in its own desktop area, as shown in Figure 4.10, is an automated feature. To implement this, the Open Source library VirtualDesktop [72]
Chapter 4. Implementation of the software toolkit

**Figure 4.9:** The screen to start a recording of an application with the Screen Capture API.

**Figure 4.10:** Windows 10: multiple desktops.
is used, which provides a C# wrapper for the functions which manage desktops. This library is used to create desktops and to move applications between desktops. Moving an application is only performed once, after the specification of a sequence. Afterwards, the desktop is designated for the respective application, and restarting the application, finding and monitoring icons, etc. will take place in this separate desktop.

4.3.3.3 Recognize icons on the captured screen

As Sikuli is originally intended to automate visible widgets, it is not straightforward to pass the screen images, gathered in section 4.3.3.2, to Sikuli. The default Sikuli API only has basic support to find things in an image, through the Finder class. However, as this is not the core concept of Sikuli, the Finder class lacks a lot of features which are present on the Screen class, or it uses old implementations of certain features.

To overcome these problems, and to expose all the functions described in section 4.3.2.2 to visible icons as well as invisible icons, the Sikuli open-source code [35] was adapted. Instead of the default Screen class, a MockScreen class was implemented. This MockScreen class implements the same interface as the default Screen class. However, instead of processing visible pixels, the MockScreen class processes the pixels from a BufferedImage Java object, which is constructed from the images saved by the method discussed in section 4.3.3.2. By doing this, all functions that are available on the default Screen class become available for images from a file, and thus all the functions described in section 4.3.2.2 are also usable for invisible screens.

4.4 Checking the sequence

4.4.1 Introduction

After demonstration and specification of an action, all steps taken to reach the final widget are known. As discussed in section 4.2, these actions include opening one or more applications, and typing and clicking within these applications. During the Programming-By-Demonstration by the user, no checks are performed on the sequence. This allows the user to interact with the program as they normally would, without interruption of the toolkit. However, while specifying the steps to the actions, several issues might occur. These issues can include ambiguities in the interface, and elements being, partially or completely, hidden due to scrolling or resizing of the application.

To resolve these issues after specification of an action, the toolkit checks the actions executed. To perform this check, it first closes the application, to reset it back to the known safe state in which the user started the recording. Next, all the executed actions are performed step-by-step, and checked by the system. When the system can’t decide, a follow-up question is asked to the user to resolve potential issues. The possible follow-up questions for resolving ambiguities are discussed in section 4.4.2. How the system should react when an icon is completely hidden due to scrolling
is discussed in section 4.4.3. Section 4.4.4 discusses how the system should handle elements becoming partially invisible due to resizing issues.

### 4.4.2 Ambiguities

#### 4.4.2.1 Introduction

The first thing that is checked for every clicked icon are ambiguities. There may be unexpected ambiguities on the screen, which lead to incorrect results during playback of a section. For example, the interface shown in Figure 2.4 would contain an ambiguity when the user has clicked on the "songs" button in the left side bar. However, as modern interfaces are highly dynamic, the position of the clicked icon can not be used to identify the right icon. For example, additional playlists can be added, and playlists can be interchanged. This would cause positions to change, which is one of the reasons why the toolkit is implemented with a pixel-based reverse engineering approach.

In order to resolve the ambiguities, some additional information is required. Section 4.4.2.2 discussed the first approach taken to resolve ambiguities, which required user intervention. However, to reduce the load on the user even further, an automated approach to resolve the ambiguities is developed, which is discussed in section 4.4.2.3.

#### 4.4.2.2 First iteration

The first approach highlighted the ambiguities after the specification. Out of all the highlighted ambiguities, the user had to select the one they pressed during specification. As the checking didn’t occur during specification, but afterwards, this step had to be taken, as the interface could have changed after closing and reopening the application. As a result, the user had to specify which one they clicked during demonstration, which was a cumbersome step. The follow-up question presented to the user, in combination with the possible ambiguities, is shown in Figure 4.11.

As shown in Figure 4.11, there exists an ambiguity for the icon "Favorite Songs" in the Spotify interface. In Figure 4.11a, the ambiguities are presented to the user, and the user is asked to select the right one. In Figure 4.11b, the user selected the one they meant. The selected icon is highlighted in green, while all other ambiguities are highlighted in red. In Figure 4.11b, the user is asked to select which icon uniquely identifies the selected "Favorite Songs" from the other matches. This is called an unique feature, which the user can select by clicking on it, as shown in Figure 4.11c. In Figure 4.11c, the user selects the icon "Library" as unique feature.

An unique feature is an icon which should uniquely identify the icon which the user means. As shown in Figure 4.11c, the system identifies the spatial relation between the icon the user means and the selected unique feature. This spatial relation can be either below, above, to the right, and to the left. The unique feature, together with the spatial relation, are used during playback of the action. During playback of the action, the unique feature is first searched for on the screen. When it is found, the matching region is extended in the direction of the spatial relation, and the first match
Figure 4.11: The first approach taken to resolve ambiguities. (a) The ambiguities highlighted, so the user can select the right one. (b) The selected icon highlighted in green, and the non-selected icon highlighted in red. (c) An unique feature selected.
Chapter 4. Implementation of the software toolkit

for the chosen icon in this direction is returned if found, as described in section 4.3.2.2. This means that in the example in Figure 4.11, both icons for "Favorite songs" will be found as they are both below the "Library" icon. However, the first one, and thus the right one, will be selected, as it is the first one below the unique feature.

However, the method presented in Figure 4.11 is not sufficient in all cases. For example, if there are two playlists with the same name, and the icon "Playlists" is selected as the unique feature, the method might possibly fail. The reason the method might fail in this case is that the playlists might be interchanged at a later stage, possibly changing the order of which icon is the first icon below the unique feature. To overcome this issue, a second iteration of the algorithm is implemented, which is described in section 4.4.2.3.

4.4.2.3 Second iteration

The algorithm described in section 4.4.2.2 only used one unique feature to differentiate between ambiguities on the screen. However, in multiple cases, using only one unique feature is not sufficient. In addition, in the method described in section 4.4.2.2, the user had to be aware what an unique feature is, and had to manually choose and select an unique feature. Finally, the method described in section 4.4.2.2 fails when no ambiguity exists during specification, but appears in the future. The reason this fails is that the system only asks the user to specify an unique feature when an ambiguity is present. To overcome these issues, a second algorithm is developed to resolve ambiguities.

The second algorithm automatically saves the context around each clicked icon. It does this during specification, so the system knows which icon is clicked in the case of ambiguities, so the user doesn't have to manually differentiate between the different matches. As the system saves the context for every clicked icon, it doesn’t matter whether the ambiguity is already present during specification or not.

The system saves the context around the clicked icon by performing the same algorithm as described in section 4.2.4. By using this algorithm, the N closest neighbours of the clicked icon are identified. For each neighbour, the distance to the clicked icon during specification and the spatial relation between the neighbour and the clicked icon during specification are saved. During playback of a sequence, all matches for a specific icon are found, and each match is compared to the saved information of the neighbours. After locating all the matches, all the neighbours get located. The new position of each neighbour is then compared to each match, to build a score. If the neighbour is further away from the match as during specification, a penalty is added. Similar, when the relative spatial orientation between the match and the neighbour has changed from specification to playback, an additional penalty is added. In the end, the match with the lowest penalty gets chosen, and should be the correct one if the interface has not dramatically changed. The pseudo-code for this algorithm is shown in Listing 4.8.
Chapter 4. Implementation of the software toolkit

Listing 4.8: Algorithm to select the best match.

```
foreach (var match in matches)
{
    foreach (var neighbour in match.neighbours)
    {
        if distance(match, neighbour) > originalDistance
            add penalty;
        if orientation(match, neighbour) != originalOrientation
            add penalty;
    }
}
return match with lowest penalty;
```

4.4.3 Scrollpanel

Originally, the user was presented with the question whether or not an icon can be hidden behind a scroll panel. This is necessary information as any scrolling that the user performs does not get tracked. In addition, an item might be visible during specification, but might be hidden behind a scroll panel later on. For example, the most favorite songs or the most favorite playlists of a user can be visible during specification, but can disappear from the main view as songs or new playlists get added over time. To overcome this, the system can automatically start scrolling when it knows a widget is hidden in a scroll panel when the icon is not immediately found.

However, as scrolling doesn’t imply a penalty on performance, as it is not executed when it is not needed, the follow-up question has been removed. Instead, this setting is automatically turned on for every icon, so the system will attempt to scroll when an icon cannot immediately be found. During the checking of the sequence, the relative position of the clicked icon within the application is saved. For example, when the icon is exactly in the middle of the application, it will be saved that it is at 50% from the left edge and 50% from the top edge. Percentages are used instead of absolute positions to remain invariant to changes in the application. The saved location is used during playback to specify on which position the toolkit should attempt scrolling, which will be dependant on the application state at that time based on the saved percentages.

4.4.4 Text

To overcome resizing issues, which may cause some interface elements to become partially invisible or to change layouts, the system identifies whether or not an icon contains text. Two examples where this is useful are shown in Figure 4.12, where the labels in the sidebar become partially invisible and change layouts due to resizing.

Figure 4.12 contains two interesting examples where searching based on text is beneficial over searching based on pixels. Firstly, as shown in Figure 4.12a, the "your library" icon only occupies one line before resizing. However, after resizing the interface, the same icon occupies two lines, as shown in Figure 4.12b. In this example, an image containing the icon "your library" taken in the situation illustrated in Figure 4.12a will not be found in Figure 4.12b. The second example where selecting
based on text is beneficial are the icons for "Recently Played" and "Favorite Songs" shown in Figure 4.12. In Figure 4.12a, both icons contain the full text, while after resizing, the text gets partially replaced by the dots, as shown in Figure 4.12b. In this case, a 70% match based on text would yield better results as a 70% match based on pixels. This is the case because the image would significantly change, while on text it would still be able to match seven out of ten letters.

To overcome issues arising with text-based icons due to resizing issues, the "Text in icon" and "Text on screen" functions, described in section 4.3.2.2, are developed. The "Text in icon" function checks whether a given icon contains any text. If this function returns any text, the user is presented with a follow-up question, as shown in Figure 3.5.

In Figure 3.5, the icon "Favorite Songs" was clicked. With the "Text in icon" function, the text is recognized and extracted from this icon. The extracted text is shown in the follow-up question, so the user can determine whether or not the text is recognized correctly. As shown in Figure 3.5, the user is given the option whether they want to select based on text or not. If the user chooses to select based on text, the "Text on screen" function is used during playback of the action to search the extracted text on the screen. The reason the user is given the option to select based on text is because searching the screen for text is slower when compared to searching the screen for an image. If searching based on text would not have this disadvantage, the system could automatically decide to select based on text. However, at this moment, the user is given the option between a more robust but slower solution, or the faster solution which might possibly fail.
4.5 Executing an action

4.5.1 Introduction

When a user interacts with a tangible, they expect the coupled action to be executed on their computer. When interacting with a tangible, the computer of the user is in an unknown state. As a result, the toolkit has to be flexible to deal with all possible states. To be as flexible as possible, the system acts different based on the current state of the system. The high-level logic taken by the system is illustrated in Figure 4.1, which shows the system tries to recover to a known state if the widget it is trying to control is not immediately visible.

Recovering to a known state is accomplished by first checking if the first icon in the recorded sequence is visible on the screen. If this is the case, the recorded sequence is executed, starting from the first clicked icon. However, when the first icon of the sequence is not visible, the application corresponding to the application is (re)started. Restarting the application brings it to a known save state, which contains the first recorded icon of the sequence. This can be assumed as the application is also (re)started during recording, so the application resets to the same state when restarted during playback. After (re)starting the application, the sequence is thus executed from the beginning, the same way as when the first icon was already visible. Checking if the first step in the sequence is visible on the screen, instead of immediately restarting the application to execute the sequence, is done to speedup the process. Oftentimes, the first step of the sequence is an action performed in a sidebar or a header, which is permanently visible in the application. In this situation, the first icon of the sequence is always visible, while the final icon not necessarily is visible. In these cases, the time required to restart an application can be eliminated, by immediately executing the recorded sequence.

After the recorded sequence has been executed, the application has reached the end-state of the recording, and thus the final widget should be visible. However, as explained in section 4.4.3, the final widget can be hidden behind a scroll panel. In this case, the final widget will not be visible and cannot be controlled. When this situation occurs, the system automatically starts scrolling to find the final widget. Scrolling happens until the final widget has been found. The reason the system will not start scrolling in the very first stage of the flowchart is that there is too much uncertainty about the state of the computer. As explained in section 4.4.3, the toolkit will start scrolling at a relative position of the application. However, in the beginning of the flowchart, it is not guaranteed that that the application is open, nor that the interface is in the right state to start scrolling (e.g. when the application is on the wrong screen).

This section discusses all actions taken in the flowchart illustrated in Figure 4.1. Section 4.5.2 describes how an application is restarted. Section 4.5.3 discusses how the recorded sequence is executed. Section 4.5.4 describes how scrolling is within the application is performed. Finally, section 4.5.5 discusses how the final widgets are controlled, where every type of supported widget is described. To answer the questions in the flowchart in Figure 4.1, whether the final widget or the first icon in the sequence are visible, the "Find all matches" function described in section 4.3.2.2 is used. When this function returns no matches, it is safe to assume that the icon or widget is not visible on screen, and thus that alternative paths in the flowcharts have
4.5.2 (Re)starting an application

(Re)starting an application is done to reset an application to a known save state. When an application is (re)started, the state of the application is known, as the same application is also (re)started during the recording of a sequence. This results in both recording and playback occurring on the exact same state of an application. The implementation to (re)start an application is accomplished by first closing an application if it’s open, as described in section 4.2.2.5. After closing the application, it is started again, as described in section 4.2.2.3.

4.5.3 Executing the sequence

4.5.3.1 Introduction

If the final widget is not immediately visible, the recorded action has to be replayed to reach the state in the application where the final widget should be visible. To reach this final state, the system replays the user-demonstrated actions step-by-step, as illustrated in the flowchart in Figure 4.2.

As illustrated in the flowchart in Figure 4.2, the system differentiates between typing actions and clicking actions, as these actions need different handling. Section 4.5.3.2 discusses how clicking actions get executed. Section 4.5.3.3 describes the procedure taken for typing actions.

4.5.3.2 Clicking actions

For every recorded action in the sequence which is a click, the same procedure is executed, as shown in Figure 4.2. For each action in the sequence, the widget corresponding to the action is looked for on the screen. When the widget is not found, scrolling is performed in the area where the the icon should be residing, until it is found. After the icon is found, a click is performed at the center of the matching rectangle of the widget. If, after scrolling, more than one matches are found for the icon, the position of the best match is used to execute the click. The pseudo-code for this procedure is shown in Listing 4.9.

In Listing 4.9, determining whether an icon is visible or not is implemented with the "Find all matches" function described in section 4.3.2.2. When this function returns

```
while (!icon.isVisible()) {
    performScroll();
}
icon.bestMatch.clickInCenter();
```

LISTING 4.9: Algorithm to perform a click in the sequence.
Chapter 4. Implementation of the software toolkit

```csharp
public static void MoveCursor(int x, int y)
{
    SetCursorPos(x, y);
}

[DllImport("User32.dll")]
private static extern bool SetCursorPos(int X, int Y);

Listing 4.10: The signature of the SetCursorPos [56] function in C#, together with the convenience function to call it.
```

more than one match, the system assumes the icon is visible on the screen. When the "Find all matches" function returns zero matches, a scroll is performed. After each executed scroll, the "Find all matches" function is executed, to check whether scrolling the interface revealed the icon on screen or not. More details about how the scrolling functionality is implemented are described in section 4.5.4.

After the "Find all matches" function described in section 4.3.2.2 returned more than one match, the system stops scrolling. If more than one match are returned from Sikuli’s Find function, the heuristics described in section 4.4.2.3 are used to select the best match of all returned matches. Next, the bounding rectangle of the best matching icon is used to calculate the position where the system should click. To extract this position, the center of the matching rectangle is calculated, which is then used for clicking.

To perform a click from C# code, two native methods are used. Firstly, the SetCursorPos function is used to bring the cursor to the right position on the screen. The SetCursorPos function takes two parameters, the X and Y coordinates to set the cursor position at. As a result, a call to this function is made with as parameters the X and Y coordinates of the center of the best matching rectangle. The signature for the SetCursorPos function in C# is shown in Listing 4.10, together with the convenience function which is used to call the SetCursorPos function.

After setting the cursor position, the mouse_event function is used to perform an action with the mouse. The mouse_event function takes as first parameter the flags which specify which aspects of the mouse are to be controlled. These flags can specify both mouse motion and button clicking. For example, to perform a left mouse click with the mouse_event, the left mouse button first needs to be toggled to the down state, and then toggling it to the up state. This sequence is performed by chaining the flag parameters passed to the function with a pipe (|) symbol. Listing 4.11 contains the signature of the mouse_event function in C#, together with the convenience function to perform a left mouse click, where the left mouse down and left mouse up events are chained in a single call.

The functions shown in Listing 4.10 and Listing 4.11 are combined to perform a mouse click on a specific location. By modifying the flags passed to the mouse_event function, the paradigm can be easily extended to support right mouse clicks, or only do mouse down or mouse up events.
4.5.3.3 Typing actions

While clicking actions require the target to click to be explicitly made visible, this is not the case for typing actions. This is a result of how the keystrokes are recorded. Typing actions don’t have an explicitly defined target. Instead, the target where typing should occur is automatically inferred by analysing the clicking actions performed by a user. For example, when a user directly starts typing after an application was opened, this means the text area where the user is typing in automatically has focus upon starting the application. However, when a user first clicks in a text area before typing, focusing the text area occurs upon clicking the target. As clicking a target is recorded as a clicking action in the sequence, playing back the clicking action brings the focus to the text area. As a result, playing back typing actions is reduced to controlling the keyboard, and does not have to deal with finding icons or focusing text areas.

As described in section 4.2.3.3, a typing action contains a string in a predefined format, which is a combination of typed characters and none, one, or more occurrences of the string "[enter]". This format drastically reduces the complexity for playing back a typing action. To play back a typing action, the string of the typing action is split with as separator the string "[enter]". When using the Split() function in C# with StringSplitOptions.None, empty entries are not removed. This results in an empty entry in the resulting array when the string "[enter]" is at the end of the typed action. Three different scenarios can be identified, which are handled. In addition, combinations of these scenarios are possible, for example when the string "[enter]" is both in the middle as well as at the end of the string to playback. The three scenarios are defined as follows:

1. **The string "[enter]" at the end of the string to playback.** For example the string "www.google.com[enter]", which happens when a user surfs to a website in the sequence. Calling the Split() method on this example with StringSplitOptions.None and "[enter]" as separator returns two strings. The first string will contain www.google.com, and the second string will be an empty string.

2. **The string "[enter]" in the middle of the string to playback.** For example the string "one[enter]two", which happens when pressing the enter key replaces the focus between input areas. In this case, the user probably typed the text one
in the first text area, pressed enter to replace the focus and then typed the text two in the second text area. Calling the Split() method on this example with the same parameters as before returns an array containing two strings. The first string contains the text one, and the second second string contains the text two.

3. The string "$[enter]\" does not occur in the string to playback. For example the string "hello world", which happens when the user does not need to press the enter key, but instead clicks after typing to start the next action in the sequence. Calling the Split() method on this example with the same parameters as before will return an array containing one string. This string has the value "hello world".

After splitting the string, playback is performed by using the SendKeys.SendWait() function in C#. This function takes as parameter a string, and sends this string to the active application, which means the focused application. As a result, the SendKeys.SendWait() function is called for every resulting string of the split operation. When the string to send is not the last string in the array, it is followed by sending an enter. An enter is also sent with the SendKeys.SendWait() function, by passing the string "ENTER" as a parameter. By following this pattern, all three cases described in the enumeration above, and combinations of these cases, are covered.

4.5.4 Attempt scrolling

As shown in the flowcharts in Figure 4.1 and Figure 4.2, the system attempts to scroll at two different locations in the process. Firstly, the system attempts to scroll when a widget is not immediately found during playback of the recorded sequence. Secondly, the system attempts to scroll when the final widget is not immediately visible when all the recorded actions are played back.

Scrolling at these two stages in the process is necessary because of two reasons. Firstly, scrolls performed by the user during recording are not tracked. As a result, the user can scroll before clicking on a target, while the system is unaware of this. The reason user-performed scrolls during demonstration do not get tracked is because these actions cannot be simply played back. They cannot be simply played back because of the second reason scrolling is necessary during playback. As explained in section 4.4.3, in modern dynamic user interfaces, items get added to lists over time. When this happens, items become hidden behind a scroll panel, because only the most recent items are shown. As this can happen in any interface, it is easy to produce wrong results when simply playing back all user-performed scrolling actions, as this results in the wrong screen coordinates when the user interface gets adapted over time.

To overcome this, the system automatically starts scrolling to find the clicked icons. The system accomplishes this by taking the relative position of the clicked icon during demonstration, and translating this relative position to the relative position based on the current size of the application. This is done to scroll in the right part of the application, even after the application has been resized. After calculating this position, the system moves the cursor to this position. Moving the cursor is done with the native function SetCursorPos(), which is shown in Listing 4.10.
After the cursor is moved, the system first scrolls entirely to the top, and then scrolls gradually down, while searching the screen for the icon after each scroll. The reason the system first scrolls entirely to the top is because some applications keep the current scroll position in some panels, even after restarting the application. Scrolling is performed by using the native function `mouse_event()`, for which the C# signature is shown in Listing 4.11. However, to use this function for scrolling, the flags used in Listing 4.11 to control the left mouse button are replaced with the flag to control the scroll wheel of the mouse. Listing 4.12 shows the convenience function used to control the scroll wheel, with the appropriate flag.

As shown in Listing 4.12, the `Scroll` function takes a parameter `scrollValue`. This parameter specifies how much scrolling is performed, and in which direction. Passing a positive value to this function will scroll upwards, while a negative value will scroll downwards. The value determines which distance should be scrolled in either direction. As the time needed to perform the scroll does not depend on the value passed, a high positive value (such as `Int32.MaxValue`) is passed to the `Scroll` function to scroll entirely to the top. After scrolling entirely to the top, the system checks whether or not the screen contains the icon or widget it is looking for. If this is not the case, the system scrolls one wheel click downwards, by calling the `Scroll` function shown in Listing 4.12 with the value -120 [54]. After each scroll, the system checks the screen to determine whether scrolling revealed the icon or widget. This procedure is repeated until the icon or widget is found or scrolling does not change the user interface, which happens in two scenarios. The first scenario where this might occur is when the extracted position is not a scroll panel, which will cause the scrolling events to be ignored. The second scenario where this might occur is when the bottom is reached, and the icon is not found. In either case, the system will not be able to continue the sequence, as the current step cannot be executed.

As reported by Yeh et al. [90], a call to the `Find` function of Sikuli takes less than 200 milliseconds for a 100x100 pixel target on a 1600x1200 screen on a 3.2 GHz Windows PC. Similarly, tests conducted on the virtual machine used to implement the toolkit, as described in section 4.1.1, revealed a call to `Find` function of Sikuli takes less than 200 milliseconds on higher-resolution screens, which are more often found in practice nowadays [86]. Tests were performed on different resolutions, including 1920x1080, 2560x1440, and 1920x1200 pixels. All tests returned correct results from the `Find` function within acceptable times. As a result, the system is able to determine whether or not an icon is visible on the screen in less than 200 milliseconds. This delay is not noticeable while scrolling, so it looks like the system is continuously scrolling to find the icon or the widget, instead of an effect where the system first scrolls, then pauses to check before continuing to scroll.

However, when the system has to deal with scrolling, there exists some cases where it can yield wrong results in the case of ambiguities. Consider an interface with a list
where multiple matches occur, but the incorrect match is positioned higher in the list as the correct match, and they are not close enough to each other to be visible at the same time. An example of a similar interface is shown in Figure 4.13.

In the scenario illustrated Figure 4.13, the part shown in Figure 4.13a is currently visible, while the part of the interface shown in Figure 4.13b is currently hidden behind a scroll panel. The current implementation fails to start the right song highlighted in green in Figure 4.13b in this case. The reason for this is that the scrolling algorithm previously described already finds the ambiguous matches highlighted in red in Figure 4.13a before the actual target enters the view. In this case, the scoring algorithm described in section 4.4.2.3 gives a score to both red matches shown in Figure 4.13a. While both matches most likely have a bad score, the best of the two is still identified as the correct icon, which causes the system to click on it and go the next step in the sequence. This will prevent the system from scrolling further and finding the right icon.

There are two possible solutions to this issue. The first possible solution is to set a minimum threshold for the scoring algorithm described in section 4.4.2.3. However, this still yields the same problem when, for example, the actual target is the first item hidden behind the scroll panel. In this case, some of the neighbours are found, as they are visible, causing the scoring algorithm to give a positive score. The second solution to this problem is more robust, but slower. The second solution is to start at the top of the scroll panel, and to scroll all the way to the bottom. While scrolling, the system should keep track of all the matches and rate them with the algorithm described in section 4.4.2.3. Only after reaching the bottom, the system should identify the best match, scroll back to the position where this match was found, and then click on the match. While this is the most robust solution, this would cause a lot of overhead, especially in large scroll lists. This overhead will cause a significant delay to the playback of a sequence.
4.5.5 Controlling the final widget

4.5.5.1 Introduction

Once the final widget in a sequence is visible, it needs to be controlled. As shown in Figure 4.1, the system can take several procedures to make the final widget visible. The most optimal case is when the final widget is visible upon starting an action. In this case, the system skips most of the steps shown in Figure 4.1, and go straight to controlling the final widget. This is the optimal case, as the user experiences the least delay in this case. However, when the final widget is not immediately visible, the system takes the necessary steps to make it visible, as shown in Figure 4.1. These steps include restarting the application (section 4.5.2), replaying the user-demonstrated sequence (section 4.5.3), and scrolling in the interface (section 4.5.4).

Once the final widget is found on the screen, it needs to be controlled. However, each type of widget has its own logic to be controlled. As discussed in section 4.2.5, the system differentiates between button widgets, toggle widgets, slider widgets, and monitoring widgets. In this section, the same structure will be followed. Section 4.5.5.2 will describe how a button widget is controlled. Section 4.5.5.3 discusses how the system controls a toggle widget. The implementation for a slider widget is discussed in section 4.5.5.4. Finally, section 4.5.5.5 discusses how a monitoring widget is controlled.

4.5.5.2 Button

As a button widget is the most simple widget of all supported widgets, the implementation to control a button widget is straightforward. All the necessary software components needed to control a button have already been discussed in previous sections. After the icon representing the button is found, either because it is already visible when the application is opened, or by replaying the sequence, the position of the button is known. Once the right position is known, the center point of the match is calculated, and this point is used to control the mouse. Clicking in the center point is performed with the `MoveCursor()` and `DoMouseClick()` functions, which are described in Listing 4.10 and Listing 4.11, respectively.

4.5.5.3 Toggle

The implementation to control a toggle widget contains a bit more complexity as compared to the implementation to control a button, which is described in section 4.5.5.2. Instead of simply clicking on the widget, by using the `MoveCursor()` and `DoMouseClick()` functions, which are described in Listing 4.10 and Listing 4.11, respectively, the system also needs to keep track of the state of the widget. Keeping track of the state of the widget is necessary because it needs to know which state of the widget to look for. In the case where no ambiguities exist, the system can just look for both states and click whichever is visible. In this case, keeping track of the state merely produces a speedup as it reduces the number of screen searches. However, in some cases, keeping track of the state is required to select the right toggle on the
screen. For example, the Gmail [31] star functionality, shown in Figure 3.13c (top) will change the appearance of the star. When the system doesn’t keep track of the state of the star, it will not know if the empty or the filled star needs to be found. As there are multiple stars on the screen (one for each mail), this can lead to mistakes, as a star can be wrongly identified as target. Keeping track of the state, combined with the algorithm described in section 4.4.2.3, will deal with this problem.

4.5.5.4 Slider

Similar to controlling the toggle widget, which is discussed in section 4.5.5.3, the system needs to keep track of the state of slider widgets. Keeping track of the widget in necessary in the case of a slider widget in necessary because not all digital sliders have a knob, or when they do the knob might not be visible until the widget is hovered. As a result, the system might be unable to retrieve the current value of a slider.

To overcome the problem of not knowing the current value of the slider, the toolkit internally keeps track of the current value of the slider. Tracking the value allows the system to know where it should start dragging the widget. In addition, based on the known position, the desired end position of the slider is calculated, based on the values retrieved from the potentiometer in the physical proxy slider. For this idea to work, the slider needs to be at a known position in the beginning, which should be 0%. To make sure the slider is reset to 0%, the system clicks at the very beginning of a slider the first time it controls the widget. Clicking on a certain position in a slider oftentimes sets it to the value at that position. However, there are two scenarios where this solution fails. The first scenario is when the system is unable to automatically reset the slider to 0% at the beginning, and the user failed to do so. The second case where this approach fails is when the slider is controlled externally, in addition to controlling it through the proxy interface. Controlling a slider externally will change the value, without having the toolkit update its internal state of the widget. In the best case, these scenarios will cause a mismatch between the value of the physical slider and the digital slider. However, in the worst case, these scenarios prevent the system from controlling the digital slider.

In the case everything goes how it’s supposed to go, and the system is able to control the digital slider, the system combines several previously discussed techniques to control the slider. Firstly, the system uses Sikuli [90] to find the slider on the screen, with the "Best matching icon" function discussed in section 4.3.2.2. After finding the icon on the screen, the bounding rectangle of the slider is known. Based on the orientation of the rectangle (height > width or width > height), the slider orientation is derived. The slider orientation is used to determine in which orientation the slider should be controlled. For horizontal sliders, the system should do a horizontal drag-and-drop movement, while for vertical sliders the movement should be vertical. Once the position and the orientation of the bounding box are known, the system calculates the current position of the knob, based on the position of the bounding box and the internal state the toolkit has on the slider. In addition, the desired end position is calculated, based on the orientation. After both positions are known, the system controls the mouse with the MoveCursor() and DoMouseClick() functions, which are described in Listing 4.10 and Listing 4.11, respectively. However, as controlling a slider widget requires a drag-and-drop movement, the DoMouseClick() function
is split up in a `DoMouseDown()` and `DoMouseUp()` function. Thus, the cursor is first
moved to the calculated starting position, then a mouse down action is performed
before moving the cursor to the end position. At the end position, the mouse released
to finish the drag-and-drop movement.

To speed up controlling a slider widget, receiving a second value to set the slider
to within a predefined interval to the previous received value, results in the system
reusing the location of the slider widget already known. In the implementation, an
interval of 3 seconds is used. This optimisation will not only reduce the number
of screen searches, but it will also cause a gradually increasing or decreasing slider.
Gradually increasing or decreasing the slider is a desired effect for sliders which
provide immediate feedback, such as volume sliders. In these cases, the user receives
feedback about the value of the digital slider while physically dragging the Tangible
Proxy Interface.

4.5.5.5 Monitor

As opposed to the button, the toggle, and the slider widgets, which are all input
widgets, the monitor widget is an output widget. As a result, controlling monitoring
widgets does not require the system to control the mouse, as is the case for input
widgets. For controlling monitor widgets, two scenarios can be differentiated.

The first scenario is when the user wants to monitor a text-based widget, and thus
is using the OLED physical proxy interface (Figure 3.17b). In this case, the system
retrieves text from the screen, to send to the proxy interface. To extract text from the
screen, the "Find text near icon" function described in section 4.3.2.2 is used, which
uses OCR to retrieve the text near a specified icon. The specified icon provided to
this function can, for example, be the image containing the label which identifies the
text to monitor.

The second scenario is when the user wants to monitor a part of the screen with the
LCD physical proxy interface (Figure 3.17a). In this case, the system merely takes a
screenshot of the part of the application the user selected during specification.
The function to take a screenshot is described in Listing 4.6. Taking only the selected
part of this screen is done by converting it to a Emgu.CV [27] image object, and by
setting a rectangle of interest (ROI) on this object. This screenshot is then sent to the
physical proxy interface for the user to consume.

4.6 Conclusion

This chapter provided implementation details about a toolkit proposed in this thesis,
which is implemented in C#.NET. The toolkit allows a user to automate arbitrary
desktop actions by demonstrating how an action should be executed, which is called
a Programming-By-Demonstration paradigm. To support the Programming-By-
Demonstration paradigm, several features were implemented to automate actions on
a Windows computer, and to track the actions a user is performing on the computer.
These features include opening and closing installed applications, tracking keyboard
and mouse input from the user, and automatically identifying which user interface
elements the user has interacted with. The software components supporting the Programming-By-Demonstration paradigm have been discussed in section 4.2.

The toolkit has been implemented to be application-independent, so it would work on all the applications a user has installed on their system. To be as application-independent as possible, the system uses the pixels from the extracted clicked interface elements to search the same widgets during playback. This approach is called a Pixel-Based approach. The implementation of the Pixel-Based approach has been discussed in section 4.3. To improve the robustness of the pixel-based approach, several measures were taken. These measures include an algorithm to resolve ambiguities, automatically scrolling specific parts of an interface, and using OCR-based recognition instead of Pixel-Based recognition in specific cases. These measures have been described in section 4.4.

Finally, after an action has been specified by the user and checked by the system, the toolkit is able to replay the action. While replaying an action, the system is able to control four different kinds of widgets. The system supports controlling buttons, toggles, sliders, and monitoring widgets, which can be differentiated as input (buttons, toggles, and sliders) and output (monitoring) widgets. To replay an action specified by the user, and to control input widgets, the system needs to take control of the mouse and the keyboard. This functionality was described in section 4.5.
Chapter 5

Implementation of Tangible Proxy Interfaces

5.1 Introduction

To bring digital user interfaces to the physical world, a concept called Tangible Proxy Interfaces was introduced in this thesis. This name contains two remarkable aspects. Firstly, tangible means the interface is perceptible by touch, i.e. it is a physical, graspable item. The second aspect in the name is proxy, which means the interface acts as a replacement or an alternative control for another interface. As such, the name Tangible Proxy Interface indicates a physical, graspable interface which can replace or be used in combination with digital user interfaces. As pointed out in Chapter 3, Tangible Proxy Interfaces can either act as an input proxy or as an output proxy. Input proxies serve as an alternative input to control actions on the computer. On the other hand, output proxies serve as an alternative output from the digital environment to the physical world. Examples of input proxies are shown in Figure 3.16. Output proxies are shown in Figure 3.17.

This chapter discusses the design and the development of the Tangible Proxy Interfaces shown in Figure 3.16 and Figure 3.17. Section 5.2 describes which hardware is used in the Tangible Proxy Interfaces. Section 5.3 discusses how the used hardware can be programmed, to implement the logic for every proxy interface. These introductory sections are followed with a discussion of both steps in Figure 3.11b. As these steps both occur in the physical world instead of in the digital world, these steps were not discussed in Chapter 4. Section 5.4 describes the first step in Figure 3.11b, which deals with coupling the Tangible Proxy Interfaces to a demonstrated action in the system presented in this thesis. The second step in Figure 3.11b, which handles what happens when a user interacts with one of the Tangible Proxy Interfaces, is discussed in Section 5.5. Finally, this chapter is concluded in section 5.6.
5.2 Used hardware

5.2.1 Introduction

The Tangible Proxy Interfaces are designed to be used both as standalone units, as well as in combination with other Tangible Proxy Interfaces to control multiple actions in the digital environment in sequence. To allow every unit to work as a standalone unit, a microcontroller is placed in every single Tangible Proxy Interface. This approach is preferred over a single master module containing a microcontroller and slave modules containing only the input and/or output hardware, because this both causes increased complexity, and it limits the combinations a user can make when using multiple Tangible Proxy Interfaces at once. To facilitate the implementation, the same type of microcontroller is used in every Tangible Proxy Interface. Section 5.2.2 describes which microcontroller is used, and why it is chosen. This section is followed by section 5.2.3 and section 5.2.4, which describe the additional hardware used for input and output proxy interfaces, respectively.

5.2.2 Microcontroller Development Board

To use the same type of board in every Tangible Proxy Interface, a development board supporting various hardware communication protocols is needed. In addition, to limit the size of the tangibles, the development board needs to be as small as possible. In all but one designed tangibles, the development board is the limiting size factor. Only in the output proxy containing the LCD screen, shown in Figure 3.17a, the size is influenced by the screen instead of by the microcontroller. Finally, the board needs to have a wireless way to interact with the toolkit proposed in this thesis. Keeping these factors in mind, a NodeMCU ESP32 development board [42] is chosen. The used board is manufactured by Joy-it [41]. A picture of the used board is shown in Figure 5.1.

The NodeMCU ESP32 is chosen because of its compact size factor of 52x31x12 millimeters [11]. In addition, the development board has plenty of interfacing options, including 2 Digital-To-Analog (DAC) converters, 15 Analog-To-Digital (ADC) converters, 1 SPI interface, 1 I²C interface, and 2 UART interfaces [42]. However, even more important, it has an ESP32 WiFi chip on board, which is one of the main reasons why the NodeMCU is preferred over an Arduino Mini or Arduino Nano. While the Arduino boards have an advantage in size over the NodeMCU, they lose this advantage when adding a WiFi chip to them, which is required for wireless communication between the software toolkit proposed in this thesis and the Tangible Proxy Interfaces.

5.2.3 Input Proxy Interfaces

Every Input Proxy Interface contains some specific hardware, which are the physical components acting as a proxy for the digital actions. The button proxy interface (Figure 3.16a) contains a push button, the toggle proxy interface (Figure 3.16b) contains a physical toggle, and the slider proxy (Figure 3.16c) contains a rotary potentiometer.
Chapter 5. Implementation of Tangible Proxy Interfaces

5.2.4 Output Proxy Interfaces

For the Output Proxy Interfaces, two different screens are used. The large output proxy (Figure 3.17a) contains a 3.5 inch Raspberry Pi color Touch Screen LCD with a resolution of 480x320 pixels [87]. This screen is wired to the NodeMCU ESP32 development board by using the Serial Peripheral Interface (SPI) protocol, which provides an interface bus to communicate between the microcontroller and the screen [33]. The wiring schematics of the screen to the NodeMCU development board are derived by comparing the pinouts of the board [42] to the pinouts of the screen, as shown on the Github page of a software library to control the screen [14] and discussed on a post on the Arduino forum [40].

The small output proxy (Figure 3.17b), contains a 1.3 inch monochrome OLED display with a resolution of 128x64 pixels [4]. This screen supports both the I^2C as well as the SPI protocols for communication with a microcontroller. However, using I^2C requires closing 2 pins by soldering them, which is a permanent action. As a result, SPI is preferred as the communication protocol. Wiring the screen to the SPI pins of an Arduino is discussed in a tutorial on Adafruit [89], which is easy to convert to the right pins for a NodeMCU board.

The physical button and the physical toggle are wired to the digital input pins of the NodeMCU so that the values can be read from within software, as described in section 5.3. The rotary potentiometer is wired to one of the ADCs of the NodeMCU. Wiring schemes to wire the hardware to the NodeMCU can be found online, either by searching for NodeMCU specific examples, or by converting Arduino examples to the NodeMCU.
Chapter 5. Implementation of Tangible Proxy Interfaces

5.3 Programming the hardware

5.3.1 Introduction

To control the used hardware, which has been discussed in section 5.2, the microcontrollers are programmed to contain the right logic. Programming the used microcontroller, which is a NodeMCU ESP32, is similar to programming an Arduino development board. The open-source Arduino IDE, which is used to write code and upload it to an Arduino, is also used to upload code to the NodeMCU board. In fact, code for an Arduino and a NodeMCU is very similar, except for a few missing or different libraries. As such, the Arduino reference can be used as documentation to program the NodeMCU board. Note that interfacing with the NodeMCU board does not work by default in the Arduino IDE. For this to work, the board specifications first need to be added to the IDE, which is done by adding an URL containing the definitions of the board to the "Additional Boards Manager URLs" text area in the preferences of the IDE. The URL used is shown in Figure 5.2a. After adding this URL, the board definitions are added to the boards manager, after which the right board is selected to upload the code to, which is "ESP32 Dev Module" in the case of the used NodeMCU ESP32, as shown in Figure 5.2b, together with the other settings used to upload code to the board.

To control the different aspects of hardware described in section 5.2, different libraries are required. The following sections describe which libraries are used to control every part of the hardware, or how the values from the hardware are obtained in software. Section 5.3.2 discusses how the ESP32 WiFi chip is controlled from within software. Section 5.3.3 describes how the state from the button and toggle components are derived in software. Section 5.3.4 describes how the values returned by a potentiometer are processed in software. Finally, section 5.3.5 and section 5.3.6 describe how the NodeMCU controls the LCD and the OLED screen, respectively.

5.3.2 WiFi

The WiFi protocol is used to communicate between a proxy interface and the toolkit proposed in this thesis. As the used NodeMCU board has an ESP32 WiFi chip
on board, using WiFi to communicate was the obvious choice. The WiFi chip is controlled from Arduino code with the WiFi library [7, 88]. This library allows the microcontroller to connect to a WiFi network. After connecting to a WiFi network, the library allows to open a connection to another host on the network by creating a WiFiClient object, after which the microcontroller can communicate with the remote host. Listing 5.1 contains the sample code to connect to a WiFi network, to open a connection, and to send data to and receive data from the remote host.

By using the functions shown in Listing 5.1, the WiFi is initialised in the setup() function in the Arduino IDE. In the loop() function, the code attempts to reconnect to the remote host if it is not connected, and then sends or reads data depending on the state of the input or output controls. For input proxy interfaces, data is sent from proxy interface to the toolkit, while for output proxy interfaces, data is received on the NodeMCU from the toolkit.

5.3.3 Button and toggle

Reading the current state of a button and a toggle in Arduino code is straightforward, as interacting with them changes the circuit to either give voltage on a digital input pin, or to remove the voltage from the pin. By reading the value of the connected input pin with the digitalWrite() Arduino function, the current value can be derived. If the value is not the same as the value previously read, the button or the toggle has been interacted with, and an event is triggered. This event is, for example, to send a status update to the toolkit so the toolkit can handle appropriately.

5.3.4 Potentiometer

Reading the value from a potentiometer in Arduino code is similar to reading the value of a button or a toggle, but with one important difference. As the potentiometer is wired to one of the onboard ADCs of the NodeMCU, the Arduino function digitalWrite() used for buttons and toggles is replaced with the Arduino function analogRead() to retrieve the value of a ADC pin. The value of the ADC pin will be in the range of [0, 4096], as the ADC input channels have a 12 bit resolution on the NodeMCU ESP32 [28]. These values are then mapped to [0, 100], to retrieve a percentage of the slider. However, for each potentiometer, the values returned at every state of the slider can differ, which is why the mapping is custom for each potentiometer.
#include <TFT_eSPI.h> // adjust settings to specific setup
TFT_eSPI tft = TFT_eSPI();

tft.begin();
tft.setRotation(1); // 1: landscape, 2: portrait
tft.pushImage(x, y, width, height, image);

Listing 5.2: Draw an image on the RPi LCD with a NodeMCU by using the TFT_eSPI library.

5.3.5 Raspberry Pi LCD

While the used LCD screen is designed for use with a Raspberry Pi, it can be used with a NodeMCU if the necessary driver is available. To control the screen from the NodeMCU, the TFT_eSPI library [15] is used, which contains support for various TFT LCD screens and various development boards, including the used Waveshare screen [87] and the NodeMCU ESP32. To use this library with the aforementioned setup, several settings are necessary, which are adjusted in the user_setup.h file in the root directory of the library. The library provides a configuration file for the specific setup used in the proxy interface, which is included by removing the comment tag. The used configuration file for the described setup is Setup11_RPi_touch_ILI9486.h, as the Waveshare screen uses the ILI9486 driver. If custom pins are used to wire up the screen to the NodeMCU, the values in this file need to be changed. After configuring the library to the specific setup used, it can be used to draw images, text, graphics, etc. to the screen. Example code to setup the screen in software and draw an image from memory to the screen is provided in Listing 5.2.

By using the code shown in Listing 5.2, an image is rendered on the screen at a given position with the specified dimensions. As a result, after receiving the bytes of an image over WiFi and placing them in memory, the pushImage() function is used to render the received image.

5.3.6 Monochrome OLED

The OLED screen is more straightforward to use on a NodeMCU than the LCD screen described in section 5.3.5. The reason for this is that the OLED screen is designed for Arduino-like development boards, while the LCD screen is designed for use with a Raspberry Pi computer. As a result, the vendor of the OLED screen, which is Adafruit, provides a library [3] for the screen which is available from within the Arduino IDE. After including the library, it is initialised by specifying the connected pins. Once done, the library is used to draw text, graphics, bitmaps, etc. on the panel. However, as the OLED panel is monochrome, drawing multi-colored bitmaps is not possible. The example code to initialise the screen and draw text on it is shown in Listing 5.3.

With the code shown in Listing 5.3, the text "Hello" is placed at the position (0,0). This sample code is adapted to display text upon receiving input from the toolkit over the WiFi/TCP connection. To place 2 lines on the screen, for example to show both a label and a value, as shown in Figure 3.9, the second line of text is positioned at (20, 0) if a text size of 2 is used.
CHAPTER 5. IMPLEMENTATION OF TANGIBLE PROXY INTERFACES

# Listing 5.3: Draw text on the SSD1305 OLED panel by using the Adafruit_SSD1306 library on the NodeMCU.

```c
#include <Adafruit_SSD1306.h>

// width, height, and 5 pins. Adjust pins to used wiring
Adafruit_SSD1306 display(128, 64, 23, 18, 16, 17, 5);
display.begin(SSD1306_SWITCHCAPVCC);
display.clearDisplay();
display.setTextSize(2);
display.setTextColor(WHITE);
display.setCursor(0, 0);
display.println("Hello");
```

5.4 Coupling a Tangible Proxy Interface

5.4.1 Introduction

After the introductory sections describing the general hardware, this section describes how the hardware is coupled to a demonstrated action in the toolkit proposed in this thesis. This is the third step in the pipeline shown in Figure 3.11, and the first step shown in Figure 3.11b, as this is the first step the user has to execute in the physical world to use the toolkit.

After demonstrating an action in the system, the user is given the opportunity to couple a tangible interface to the action. Note that coupling a physical proxy interface is optional, especially for input actions, as the user is given the possibility to couple a keyboard shortcut to these actions. However, coupling a keyboard shortcut to an action does not rule out the possibility to couple a physical proxy interface, as both can coexist at the same time. However, when the user decides to couple both a physical proxy interface and a keyboard shortcut to the same action, the state of the software widget might not correspond to the state of the physical widget. For example, when a slider is controlled to 100% with a keyboard shortcut, the physical slider does not move, keeping its previous value. Similar situations can occur with the toggle widget. The button widget does not suffer from this issue, as it is stateless.

When a user decides to couple a Tangible Proxy Interface to an action, they place the physical object on an enlarged version of the widget they want to couple the tangible to. Placing the tangible on the screen will connect it to that widget. Placing the physical object on the screen is used to extract the position where it is placed. This allows the system to support more complex scenarios in an intuitive way. One of the possible scenarios which can be supported with this paradigm, which has already been discussed, is placing three buttons on a slider. By extracting the position where the user placed the tangible on the touchscreen, the system derives which buttons map to 0%, 50%, and 100%, respectively. However, current state-of-the-art methods of tracking tangibles on touch screens [83, 84] have a limited number of uniquely identifiable tangibles, which is restricted by the footprint of the tangible and the number of uniquely identifiable touch patterns the footprint can facilitate. As the Tangible Proxy Interfaces are designed to be as small as possible, they suffer from the same limitation, which restricts the number of Tangible Proxy Interfaces a single user can use.
To overcome this problem, and to support an arbitrary number of Tangible Proxy Interfaces in the toolkit, the solution proposed in state-of-the-art work [83, 84] has been augmented with a second paradigm, which allows the toolkit to differentiate between the different proxy interfaces, even if they produce the same touch pattern. This paradigm requires a user to hold the tangible while coupling it, which they naturally do, so this does not create an extra demand on the user. However, this paradigm requires the user only touches the tangible they are currently coupling, and does not touch the other tangibles simultaneously.

This section describes the necessary components to couple a tangible to an action, which include recognising it on a touch screen, recognising which tangible the user is currently touching, and communicating between the tangible and the software toolkit. Section 5.4.2 discusses the implementation of how a tangible is recognised on a touchscreen. Section 5.4.3 describes how the system differentiates between touched and untouched tangibles. Finally, section 5.4.4 describes how a tangible communicates with the software toolkit.

### 5.4.2 Recognising a tangible on a touch screen

The technique used to recognise tangibles on a touch screen is based on the techniques proposed in PERCs [83] and PUCs [84]. These techniques augment tangibles with a conductive material in a specific pattern. By interconnecting all conductive touch points, and grounding the group of touch points, an unmodified touchscreen recognises the pattern as a touch. PERCs and PUCs then recognise the pattern to determine which tangible is placed on the screen.

To build an own implementation of a tangible on a touchscreen, conductive foam [44] and copper tape was used. Conductive foam is used as it is a soft material, which has no risk of damaging the touchscreen when dragging a tangible over it. The foam touch points are interconnected by copper tape to form one touch field. Every Tangible Proxy Interface is assembled with four 1x1 centimetre conductive foam pads in the corners of the bottom plate, with copper tape connecting the four corners, as shown in Figure 5.3.

Placing a tangible with the bottom side, as shown in Figure 5.3, down on a capacitive touchscreen triggers touches, when the user is holding part of the copper to ground the touch. To test this, an Acer T272HUL touch screen is utilised. This is a 27 inch capacitive touch screen monitor with a resolution of 2560x1440 pixels which supports up to ten simultaneous touches [37]. This monitor provides a HDMI port for video input and an USB interface to receive the touch points. In the development setup, which implemented the toolkit proposed in this thesis in a Windows Virtual Machine, it is important to connect the USB interface to the virtual Windows OS, and not to the hosting operating system, to receive and process the touches in the toolkit.

Once the screen is hooked up to the Windows machine hosting the toolkit, touches are captured from within the C# WPF application by opening a transparent full-screen window without a window style (WindowStyle="None" and AllowsTransparency="True"). By making the window fully transparent, touches and clicks are still propagated to underlying windows or interface elements. On the WPF Window, the TouchDown, TouchUp, and TouchMove events are subscribed to to get access to the
The bottom side of a Tangible Proxy Interface, containing four 1x1 centimetre conductive foam pads in the corner, which are interconnected with copper tape to form a conductive whole.

As the window overlays the whole screen, every touch first passes the toolkit before being propagated to the actual target.

Within the TouchDown, TouchUp, and TouchMove events, TouchEventArgs are provided for the triggering touch. From the TouchEventArgs, a TouchDevice object is obtained. While the name suggests otherwise, the TouchDevice objects contain the value for the touch that generated the event. As such, the system keeps track internally of all touches by using this identifier. In the TouchDown event, the touch is added with its current position. In the TouchMove event, the position of the touch corresponding to the identifier is updated. In the TouchDown event, the touch corresponding to the identifier is removed.

After every internal state update, the tracked touched are checked to determine if they form a pattern. To find a pattern, at least four touch points are required. If at least four touch points are currently found, the positions of these points are checked in pairs of four to see if they form a rectangle with corners of 90°, with an error margin of 1°. An error margin of 1° is used to compensate for the sponginess of the used conductive foam, which causes a touch not happening exactly in the middle of the foam. If a matching rectangle is found, the system assumes a proxy interface is placed on the screen, as shown in Figure 5.4.

As shown in Figure 5.4, the proxy interface is recognised when being placed on the screen. In fact, only the corner points are known, and the lines are only drawn between each adjacent corner for illustrating and debugging purposes. After the positions of the corners are known, the center position of the tangible is calculated by taking the average position of the four corners.
While this approach of locating tangibles on touch screens works in most cases, it fails when two touch points align exactly in the same horizontal plane. In this case, the touch screen fails to recognise two different touches, which is a result of the grid alignment of a capacitive touch screen. However, experimenting with multiple touch screens is required to identify if this is only an issue on the used touchscreen or if all touchscreens suffer from this issue. In addition, if a user is able to form a perfect rectangle with his fingers, he is able to trick the system into thinking a tangible is placed on the screen. Both issues, however, are solvable by changing the pattern, at the cost of a more complex pattern recognition algorithm.

5.4.3 Identifying the touched tangible

The implemented method to recognise tangibles on a touchscreen, which is described in section 5.4.2, only identifies whether or not a tangible is placed on the touchscreen, but does not differentiate between different tangibles (i.e. the method does not identify which tangible is placed on the touchscreen). However, the system needs to know which tangible the user is interacting with in order to couple the right tangible to the right action. To overcome this issue, the tangibles have been augmented with a non-intrusive technology, which allows differentiation between them.

To know which tangible the user is placing on a touchscreen, every Tangible Proxy Interface has been equipped with a Do-It-Yourself (DIY) capacitive touch sensor. With a capacitive sensor, it is possible to detect touches or proximity to the sensor. As a result, each NodeMCU is aware whether the tangible it is contained in is being
touched or not. A capacitive sensor is created by connecting two input pins of the NodeMCU with a high-value resistor (100kΩ - 1MΩ), and placing a conductive material, such as copper tape, at one end of the resistor. Touching the conductive material will change the capacitance, allowing the software to detect changes. The higher the value of the resistor, the more sensitive the capacitive sensor is, which allows the sensor to detect proximity instead of touches, if desired.

To place a capacitive sensor in the Tangible Proxy Interfaces, the inner sides of the enclosure have copper tape fixated on them. One side of a 100kΩ resistor has been soldered to the copper tape, with both ends of the resistor connected to two different input pins of the NodeMCU. Figure 5.5 illustrates the inside of one of the Tangible Proxy Interfaces, where the copper and the resistor are visible.

By using a 100kΩ resistor, the NodeMCU is able to detect touches, but not proximity. However, tests have shown that with the current setup, the NodeMCU is able to reliably detect touches, even through the 3 millimetre MDF used to build the box. To detect touches on the NodeMCU, the CapacitiveSensor [77] library is used, which takes as input the two pins the resistor is connected to, and returns the measured capacitance. Afterwards, a threshold is placed on the measured value, to differentiate between touch and non-touch values.

5.4.4 Communicating between a tangible and the software toolkit

After a tangible is aware that is being touched, by checking the capacitance as described in section 5.4.3, the tangible needs to communicate this to the toolkit proposed
Chapter 5. Implementation of Tangible Proxy Interfaces

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;InputType&quot;: &quot;Capacitive&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&quot;Value&quot;: &quot;On&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&quot;Identifier&quot;: &quot;Button 1&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Listing 5.4**: An example JSON string used to communicate between a Tangible Proxy Interface and the toolkit.

In this thesis, so the toolkit is aware which tangible is being touched. As previously discussed, the NodeMCU ESP32 has a WiFi chip on board. Connecting to a WiFi network and opening a TCP socket to a remote host on the connected network is discussed in section 5.3.2.

To receive the data from a tangible in the toolkit, the toolkit starts a socket listening for incoming connections. The tangibles connect to this socket when they have data to exchange. To exchange structured data, messages are structured in JSON semantics. Each tangible sends an unique identifier with every message, to allow the toolkit to link messages to tangibles. In addition, it sends the type of the sensor which has been updated, and the new value of said sensor. For example, the JSON string shown in Listing 5.4 indicates that Button 1 is currently being touched. This string is then send through the socket to the toolkit, upon which the toolkit takes appropriate action.

### 5.5 Interacting with a Tangible Proxy Interface

The last step of the flowchart (Figure 3.11) a user performs using the system is interacting with a Tangible Proxy Interface. Interacting with an input proxy interface triggers an action on the system, while an action occurring on the system can trigger an action to happen on an output proxy interface. As a result, communication for input proxies goes from tangible to the toolkit, as opposed to output proxies, where the tangible receives data from the toolkit.

For all communication, the TCP socket server used to communicate between a tangible and the software toolkit, which is described in section 5.4.4, is used. For input proxies, the tangible sends data to the server in the format shown in Listing 5.4. The JSON string is adapted based on the sensor type and value, and every tangible uses its unique identifier, so the toolkit knows which action to execute upon receiving a value update.

For output proxies, the server sends the data to the tangible. The data is either textual data containing an update for a monitored label or binary data containing an image. In either case, JSON is used to structure the data, and the library ArduinoJson [12] is used to interpret the received data on the NodeMCU. After interpreting the data, the code examples in Listing 5.3 and Listing 5.2 are used to draw text or images on the screens, respectively.
5.6 Conclusion

This chapter discussed the steps in the pipeline (Figure 3.11) the user follows when interacting with the toolkit proposed in this thesis which occur in the physical world. These steps are highlighted in Figure 3.11b, and deal with coupling a Tangible Proxy Interface to an action, and interacting with the Tangible Proxy Interface to trigger an action. This chapter described how a NodeMCU is used as microcontroller in every single Tangible Proxy Interface, and which hardware is used in every tangible. Afterwards, the chapter described how the NodeMCU is programmed with the Arduino IDE to write custom logic for every Tangible Proxy Interface. Afterwards, all the used Arduino libraries are enumerated, with some code examples how the libraries are used. Nextly, this chapter described how a Tangible Proxy Interface is placed on a touchscreen to recognise it, and how a capacitive sensor is used to derive which Tangible Proxy Interface is being interacted with. Finally, the chapter discussed the usage of a TCP socket server to communicate between the tangibles and the software toolkit, to exchange updates from the tangibles to the toolkit and vice versa.
Chapter 6

Discussion and future work

6.1 Introduction

The current implementation of Tangible Proxy Interfaces is designed to automate actions which consist of multiple steps. The typical structure of an automated action consists of one or more steps to reach a final widget, and then to control the final widget. The actions currently supported to reach a final widget are opening applications, typing, scrolling, and clicking. The supported final widgets which the current implementation is able to control can be divided in four categories: buttons, toggles, sliders, and output widgets, which have been referred to as monitors within this thesis. While this paradigm covers many actions a user would want to automate, it does not cover every imaginable action a user can perform on a computer. Extending the current paradigm is extensively discussed in section 6.2. In addition, the current implementation is not designed to work with interface elements which change significantly between different executions of the same action. However, some measures can be implemented in future work to overcome these issues. Directions to handle changing interface elements are provided in section 6.3.

Not only the software paradigm can be extended. A minimal set of Tangible Proxy Interfaces has been developed, containing a physical button, a physical toggle, a physical rotary slider, and two physical screens. These tangibles are chosen as they illustrate the workflow clearly, and to provide at least one physical counterpart to every supported software widget. Section 6.4 provides guidelines to extend the set of supported Tangible Proxy Interfaces to allow richer and more customised interactions. In addition, the current iterations of Tangible Proxy Interfaces require an external power supply, which restricts the movement of the tangibles. Section 6.5 provides future directions to make the Tangible Proxy Interfaces truly wireless. Section 6.6 provides guidelines to polish the Tangible Proxy Interfaces, to make them look more like a finished product instead of a prototype. Finally, this chapter is concluded in section 6.7. Apart from the issues and future work highlighted in this chapter, discussions about the effectiveness of specific techniques, and guidelines to improve the techniques, have been provided when describing the techniques throughout the thesis text.
6.2 Extending the tracking and replay paradigm

As described in Chapter 4, the current version of the toolkit tracks actions taken by the user during recording an action. However, currently, the system only tracks simple user input, such as mouse clicks and basic keystrokes, such as typed characters, the enter key, and the shift key. However, the user can perform more actions than the ones tracked, which the current implementation will fail to record. For example, the user can perform drag-and-drop gestures, which they expect the system to replay. Recognising a drag-and-drop action is straightforward, for example by comparing the position of the *mousedown* event with the position of the *mouseup* event. If these positions are significantly different, the system can assume a drag-and-drop gesture has been performed. However, extracting the drag-and-drop targets is not as straightforward. While extracting the drag-target is similar to extracting a clicked widget, which is discussed in section 4.2.4, extracting the drop-target is more complex. The reason for this is that the appearance of the target often changes when dropping the element on it, while the system needs to know how to target looked before it was dropped on, to search the drop-target during playback. A possible solution to solve this is to take a screenshot at the beginning of a drag-and-drop action, and to use this screenshot together with the position where the gesture ended to identify the target.

In addition to the drag-and-drop issue, other extensions to the tracking and replay paradigm can be made to make it more robust and support every action performed by the user. While the system takes several countermeasures to prevent the user from interacting directly with the operating system during recording, this is not completely impossible. Preventing the user from interacting directly with the operating system is desired to make sure the user opens applications through the toolkit, so these actions get tracked, which allows the system to replay these actions. The toolkit attempts to limit interactions with the operating system by resizing to a full-screen application, covering the desktop and the Windows task bar. In addition, it opens and positions the applications opened through the toolkit so that they also cover the task bar. While this solution limits the ability of the user to click applications in the task bar or on the desktop, the user can still press the Windows key on their keyboard to bring back the task bar. There are two potential solutions to overcome this problem. The first solution is to disable the Windows key. However, this solution requires an edit of the registry, which both needs modification to the system and is a permanent change (i.e. it cannot be turned on while recording and be turned off once done). The second solution consists of adding the Windows key to the tracked keystrokes, and simply letting the user interact with the key, but adding it to the typed keystrokes to replay the keystroke during playback. In theory, this solution would work as the toolkit would simply recognise the clicked icons in the task bar or the Windows menu, and also replay these actions. However, the solution would not allow the system to position the externally opened applications to take up the empty space, as illustrated in Figure 3.2. This could either cause the application to interfere with the interface of the toolkit, if the application is able to take over the topmost position of the toolkit, or to be partially hidden underneath the toolkit. In addition, this results in the system not knowing which applications are opened, which results in the toolkit not being able to reset the applications to a known safe state when it fails to find an icon.

On a similar vein to adding the Windows key to the tracked keystrokes, the user can perform OS-wide or application specific keyboard shortcuts without the toolkit
being aware of them. By adapting the keystroke tracking algorithm to include every possible keystroke a user can perform, these can be tracked while still remaining application-independent, as the toolkit does not need to look for application-specific keystrokes. This would also require adaptation of the logic implemented to replay keystrokes, to support playing back every key.

An additional feature which extends the set of supported actions on the system is supporting specification of positional items in a user-friendly manner. For example, if a user wishes to automate an action to start the first, or the tenth, song of a hit list, this is currently not possible. The reason this is not possible is because the song at the corresponding position will change over time, requiring re-specification currently. To implement a similar feature, the system needs a way to distinguish list items from each other, and to infer a position given the look of one list item. This feature allows actions which execute a changing action over time, as the action executed is dependant on changes in the list.

### 6.3 Handling changing interface elements

The current implementation of the toolkit is designed to work for automating tasks containing a sequence to reach a final widget and to control the final widget, as discussed in section 6.1. While several algorithms are implemented to ensure a robust playback, even in the case of dynamically changing interfaces, which are discussed in Chapter 4, the current implementation is unable to reliably deal with individually changing icons, unless all states of the icon have been specified by the user, for example, as is the case for a toggle. In other words, the system is designed to robustly handle changes at interface level, but not at individual icon level. This is, however, a serious limitation, as it is easy to find plenty of examples of individually changing icons. Examples of individually changing icons are shown in Figure 6.1.

As shown in Figure 6.1, individually changing icons are found in all sorts of interfaces. The examples shown in Figure 6.1 can be divided in two categories. The first category is shown in Figure 6.1a and Figure 6.1b. In these cases, the size of the icon remains the same, but some of the inner parts of the icon change according to the state they represent. For example, the icons in the MacOS dock (Figure 6.1a) add a badge to indicate notifications of the respective application. The battery icon (Figure 6.1b) is filled according to the remaining battery percentage. For these scenarios, a possible solution is to provide the user with a masking tool, to let the user indicate the system which parts of the extracted icon need to be ignored. By making these pixels, for example, transparent, these pixels can be used as wildcards when looking for the icon during playback. Obviously, this approach may suffer from human errors, as the user may fail to identify the icon in the top left of Figure 6.1a as an individually changing icon, as the icon does not yet contain the number of unread notifications during specification.

The scenario illustrated in Figure 6.1c, however, is not as straightforward to solve as the cases illustrated in Figure 6.1a and Figure 6.1b. Imagine a user who wishes to display incoming messages textually on the OLED output proxy interface. To specify a message in the system, the user clicks the message, which correctly identifies the blue speech bubble with the text inside it. However, another incoming message
will not only contain other text, but also have another length, which will change the dimensions of the speech bubble. While differing text can be solved similar to the masking solution proposed for the scenarios in Figure 6.1a and Figure 6.1b, the changing dimensions need another approach to be solved. A possible solution would only take into account the corners of the bubble, and not the length of the edges, to identify a bubble, similar to how Prefab [22, 24] builds a prototype for a widget. However, this specific example would require two such specifications, one for the rounded bubble, and one for the bubble with the little arrow (the bottom bubble in Figure 6.1c). After a similar solution has been implemented in the toolkit, the already implemented algorithm to resolve ambiguities will be able to identify the most recent incoming messages, as these are the closest to the surrounding interface elements, such as the text area to type new messages, as was the case during demonstration, which will result in the smallest penalty score.

6.4 Extending the set of Tangible Proxy Interfaces

In addition to extending the software paradigm, as discussed in section 6.2 and section 6.3, the hardware paradigm can be extended as well. The currently supported set of Tangible Proxy Interfaces can be easily extended with additional or alternative input or output interfaces. An example of an additional proxy interface can be, for example, an output proxy interface containing a LED light which the user can program to light up when a notification arrives. Other additional proxy interfaces can include, for example, accepting touch screen input on the current version of the LCD output proxy. Supporting touch input will convert the output proxy to a Hybrid Tangible Proxy Interface, as it would be able to serve both as a output medium to the physical world, as well as an alternative input to the digital environment.
In addition to providing additional input or output proxy interfaces, alternative tangibles can be provided as well. Alternative tangibles are tangibles which serve the same functions as the currently supported proxy interfaces, but which take a different physical form factor. Supporting more than one physical counterpart for every digital interface element would allow a user to chose the best suited physical control for their needs. This can, for example, be useful for disabled users, who might not be able to control a rotary potentiometer, but have no difficulty controlling a linear potentiometer, or vice versa. This allows custom tailored proxy interfaces to the specific needs of a user, similar to AMI [74], where adaptable tangible music interfaces are provided to support the varying needs of people with dementia.

On a similar vein to extending the set of Tangible Proxy Interfaces by providing alternative inputs, extending the set of Tangible Proxy Interfaces by providing duplicates of the input proxies can also allow for more flexible interfaces which support the varying needs of disabled users. For example, by providing more than one button, the slider example where every button corresponds to a specific location of the slider, can be realised. While all aspects, both in software and hardware, have already been provided to realise similar scenarios (for example by tracking the position of a tangible on a touchscreen), providing duplicates of every tangible effectively makes similar interactions possible.

### 6.5 Truly wireless Tangible Proxy Interfaces

The current generation of Tangible Proxy Interfaces require an external power supply in order to function. An external power supply can be either a USB connection to a computer, a USB charger plugged into a wall outlet, or an external USB power bank. Within every Tangible Proxy Interface, a hole has been provided to allow connection of the NodeMCU to a USB power supply. However, the three supported solutions to provide power all have a significant disadvantage. Providing USB power from a computer or a wall outlet limits the freedom of movement a user can perform with the tangible. Using an external USB power bank, on the other hand, adds significant bulk to the tangibles.

One of the possible solutions to make the Tangible Proxy Interfaces truly wireless is to add a battery to every tangible. A NodeMCU can be powered by an external 3.7V Lithium battery by using a 3.3V Low-dropout regulator (LDO). Powering the NodeMCU on battery power would, however, require some modifications to the software of the NodeMCUs, such as putting them into deep sleep mode [43] when they are not being used to save battery life. In addition, tests need to be performed on the output proxy interfaces with the LCD and the OLED panels to see how much current they are using, to see if powering them with battery power is a valid option.

To make the batteries easily rechargeable, something like the Adafruit Powerboost 1000C [2] can be used. The Powerboost is a small DC/DC boost converter module that can be powered by any 3.7V Lithium battery, and can convert the battery output to 5.2V DC. This output voltage means even the proxy with the LCD screen can be operated on battery power, as this screen requires 5V to be powered. Even more interesting is that the Powerboost module is able to simultaneously provide power to the NodeMCU, as well as to charge the battery, when plugged in to USB power. This
would allow the user to use the tangibles both wireless, as well as while charging, without any interruption. By including a cheap magnetic USB charger (e.g. [85]), plugged in permanently to the charging port of the Powerboost, even a small docking station can be created where the tangibles can snap into to charge themselves, for maximum user convenience. Figure 6.2 illustrates the workings of the Powerboost module, and the magnetic charger. In Figure 6.2a, power is provided from an external power supply to charge the battery in Figure 6.2b and to provide power to the output in Figure 6.2c while connected. If the external power supply in 6.2a gets disconnected, the battery takes over to power the circuit. Figure 6.2d shows an example of a magnetic charging tip, which can be placed in the external power supply port shown in Figure 6.2a to provide a convenience charging option to the user. Once attached, the cable shown in Figure 6.2e is able to snap into the tip to start charging the battery.

6.6 Polishing the Tangible Proxy Interfaces

In contrast to the previous sections, which discussed improvements or additions which can improve the user experience or the extend the actions the user can accomplish with the system, this section describes how the Tangible Proxy Interfaces can be polished so they look more like a finished product instead of a prototype, which is merely a visual improvement. Firstly, the tangibles can be made smaller by removing the header pins which came attached to the NodeMCU and the other components. By removing these, and soldering the cables directly to the holes, the height of the Tangible Proxy Interfaces can be reduced. However, shrinking the width and the depth of the tangibles is not as easy, as these are determined by the largest component in every tangible, which is either the LCD panel or the NodeMCU. Removing the header pins and other unneeded connectors can, however, instead of reducing the
size, make room for the battery hardware, discussed in section 6.5, while maintaining the same footprint. This is not currently done as it are permanent actions, with the risk of damaging the hardware. A second visual improvement which can be made is covering up the copper tape at the bottom, which is visible in Figure 5.3, so that only the conductive foam pads are visible. This can either be done by interconnecting the foam pads on the inside of the tangible, or by applying a second bottom plate with the corners cut out to the bottom of the tangible.

6.7 Conclusion

Within this chapter, the current weaknesses of the system have been discussed and guidelines have been provided to improve the system in future work. Section 6.2 identified issues in the current tracking and replaying paradigm of the software toolkit, and described which aspects should be implemented to be able to better track every action performed by the user and to replay them. Section 6.3 identified a set of interfaces which is not currently supported by the automation toolkit, and proposed different algorithms which can help in adding these interfaces to the set of supported interfaces. Turning the focus to the hardware aspects, section 6.4 proposed future directions to extend the set of supported Tangible Proxy Interfaces. Section 6.5 discussed how Tangible Proxy Interfaces can be designed to be truly wireless, with an on-board battery, and even proposed a docking station to which the tangibles could snap into to charge them while being used. Finally, section 6.6 provided some guidelines to further polish the Tangible Proxy Interfaces, which is required if they were to evolve from prototype to a daily used product.
Chapter 7

Conclusion

7.1 Introduction

Within this thesis, a software toolkit is proposed to automate user interface behaviour in dynamic user interfaces. An accompanying set of tangible controls is proposed with which the automated user interface behaviour is controlled from within the physical world. In this thesis, several state-of-the-art research systems are identified, which work towards similar goals as the toolkit presented in this thesis. However, several important gaps have been identified in the state-of-the-art, which opened up novel opportunities for the work presented this thesis. The identified gaps in the state-of-the-art systems lead to four novel research challenges, which have been addressed in this thesis. This chapter provides a conclusion of how every research challenge is handled by the proposed system. Sections 7.2, 7.3, 7.4, and 7.5 handle the first, the second, the third, and the fourth research challenge, respectively.

7.2 Research challenge 1

Current state-of-the-art systems require either programming knowledge from their users to automate actions, or they are tailored for very specific applications, allowing only restricted automation possibilities. Providing end-users without any technical knowledge with tools to automate arbitrary desktop actions has been under-explored in current research. As a result, a first research challenge for this thesis was identified:

- Research challenge 1: Provide end-users with a tool to automate desktop actions without requiring knowledge of basic programming constructs or any other technical knowledge.

The toolkit proposed in this thesis addresses the first research challenge by letting the user specify an action they want to automate by demonstrating the action to the system, and by asking follow-up questions. This is called a Programming-By-Demonstration paradigm. To specify an action, the user interacts with both the toolkit and the application(s) they want to automate. While interacting with the application(s), the actions performed by the user are recorded, so they can be played back at a later stage. When ambiguities occur in the specification, the system first tries to solve them algorithmically. When this is, however, not possible, a follow-up
question is presented to the user, to resolve the ambiguity. With this paradigm, a user who can execute an action on a computer, can automate that action, without any additional knowledge.

7.3 Research challenge 2

To support automating arbitrary interfaces, recent research has introduced a method referred to as pixel-based reverse engineering methods [90, 22]. Pixel-based reverse engineering methods use the pixels of an application to automate the application, without knowing anything about the internal structure of the application, in fact only automating based on the appearance of an application. While the approach of pixel-based methods works well for automating interfaces which keep their appearance over time, they are unable to handle highly dynamic interfaces which change appearance over time. Previous state-of-the-art systems have solved this by, for example, providing a visual scripting language [90]. However, this requires, at the very least, knowledge of basic programming structures by the end-user. As the work presented in this thesis focuses on end-users without any technical knowledge, a second research challenge has been identified:

- Research challenge 2: A robust pixel-based reverse engineering approach for automating dynamic user interfaces.

The second research challenge is handled in the proposed toolkit by utilising two main approaches. Firstly, the toolkit assumes every user interface is dynamic, and as such can change appearance over time. The system handles this assumption algorithmically, by saving the context around every automated action, to assure robust playback in the case of current or future ambiguities. In addition, the system utilises several heuristics to automatically start scrolling interfaces, when, for example, a user interface element is not immediately visible where it was expected. If the system is unable to deal with an ambiguity algorithmically, a follow-up question is presented to the user as a fall-back method.

7.4 Research challenge 3

Early research [29] already pointed that having multiple physical input methods mapped to multiple digital inputs stimulates two-handed interactions. In addition, commercial systems [68, 73] exists, which provide additional input methods for specific functions, showing the importance of similar products. However, commercial tools only focus on providing input for specific applications, such as photo editing or music editing software. As the toolkit proposed in this thesis is able to automate arbitrary user interfaces, coupling tangibles to these actions allows the user to provide a custom input method to an action of their choice. As this thesis focuses on users without any technical knowledge, it is necessary to have an easy-to-use paradigm to couple physical tangibles to automated digital actions. This is the foundation of the third research challenge identified in this thesis:
• Research challenge 3: A tangible programming paradigm for non-programmers to link physical controls to automated user interface behavior as defined in research challenge 2.

To address the third research challenge, a set of tangible interfaces has been specifically developed for the purposes of this thesis, which this thesis refers to as Tangible Proxy Interfaces. The tangible interfaces are designed to support a non-intrusive coupling paradigm, so they can be coupled to automated interface behaviour by non-expert users. Users can take one of the supported tangible controls, and place them on a touchscreen to couple them to an enlarged version of the widget they want to automate. The hardware is designed in a way so the system can unambiguously differentiate between the different supported tangibles, and as such couple the right tangible to the right action.

7.5 Research challenge 4

To automate arbitrary user interfaces, the work presented in thesis builds upon what the literature refers to as pixel-based reverse engineering methods, as discussed in section 7.3. As pixel-based reverse engineering methods work strictly on the visible pixels of a screen, they require the elements they want to automate to be visible on the screen. However, when a user wishes to automate multiple actions simultaneously, such as monitoring four different variables to an output proxy interface, this causes a lot of screen clutter, if it is even possible to position them all at once on the screen. In addition, traditional pixel-based methods prevent the user from using the computer at the same time, as the automated interfaces need to stay on top. As this is not an interesting scenario for the user, this thesis also investigated how pixel-based methods can be used on non-visible applications, which is the fourth research challenge:

• Research challenge 4: Overcome the inherent limitation of pixel-based approaches which requires pixels to be visible on the screen in order to be recognised.

The fourth research challenge is addressed by using some Windows specific features. By programmatically creating additional virtual desktops, and placing every automated application on its own desktop, there can be an arbitrary amount of automated applications, without any clutter on the screen occurring. In addition, the toolkit proposed in this thesis implemented a way to access the pixels from an application placed on a second virtual desktop, without making that desktop visible. After getting access to the pixels from another desktop, a state-of-the-art pixel-based approach [34] has been adapted to work on the acquired pixels.
Appendix A

Dutch Summary

A.1 Inleiding

Het werk voorgesteld in deze thesis introduceert de term "Tangible Proxy Interfaces". Dit zijn fysieke voorwerpen die enerzijds dienen om invoer te geven van de fysieke wereld naar de digitale wereld, bijvoorbeeld door invoer te voorzien naar een computer. Anderzijds worden deze voorwerpen gebruikt om uitvoer van de digitale wereld om te zetten naar de fysieke wereld. Deze voorwerpen kunnen bijgevolg ofwel als vervanging van traditionele invoer methodes, zoals een toetsenbord en een muis, dienen, ofwel in combinatie met de traditionele invoer methodes gebruikt worden. Het combineren van traditionele invoer methodes samen met tastbare voorwerpen die elk hun eigen functie uitvoeren bevordert interacties met twee handen [29].

In het verleden zijn er reeds commerciële producten verschenen die gelijkaardige functionaliteiten aanbieden, zoals Palette Gear [68] en Novation Dicers [73]. Deze commerciële producten zijn echter ontworpen voor zeer specifieke gebruikersgroepen, zoals designers of artiesten. Hierdoor zijn ze vaak enkel bruikbaar voor zeer specifieke functies, en werken ze enkel met de applicaties waarvoor ze ontworpen zijn, zoals bijvoorbeeld enkel voor Adobe programma’s. Doordat deze producten zeer applicatiespecifiek te werk gaan, kunnen ze gebruik maken van de kennis van specifieke functies van het programma, zoals shortcuts, om hun functies te verrichten. In tegenstelling tot deze commerciële aanbiedingen, ligt de focus van het voorgestelde werk breder. Het doel van “Tangible Proxy Interfaces” is namelijk om een zo groot mogelijke variatie van gebruikers aan te spreken. Dit kan gaan van expert gebruikers, die de fysieke voorwerpen gebruiken om hun productiviteit te verhogen, maar ook gebruikers met een beperking die bepaalde fysieke voorwerpen gebruiken om hun muziekspeler te bedienen. In elk geval verleent het systeem de vrijheid om het best geschikte fysieke voorwerp te kiezen voor te gebruiken, zodat elke gebruiker het systeem kan gebruiken op een manier waarop het voor hun handig is, ongeacht mogelijke fysieke beperkingen.

Om een grote variatie aan gebruikers te ondersteunen, moet het systeem ook een grote variatie van te automatiseren acties ondersteunen. Aangezien elke gebruiker een ander doel zal hebben voor welke actie de fysieke voorwerpen moeten uitvoeren, is het niet mogelijk een verzameling van ondersteunde acties vast te leggen. Daarnaast zal elke gebruiker acties willen automatiseren die in verschillende applicaties plaatsvinden, zoals muziekspelers, browsers, en specifieke programma’s die aansluiten op de noden van de gebruiker. Hierdoor is het zelfs niet mogelijk om een
set van ondersteunde programma’s te definiëren.

Om dit te overkomen is het doel van het werk dat voorgesteld wordt in deze thesis om applicatie-onafhankelijk te werken, waardoor er geen gebruik kan gemaakt worden van applicatie-specifieke kennis. Recent werk in de literatuur [22, 90] hebben werk verricht in wat "pixel-gebaseerde methodes" wordt genoemd. Deze methodes gebruiken enkel de zichtbare pixels op een computer om acties te automatiseren, in plaats van applicatie- of besturingssysteem-specifieke zaken. Deze methodes vereisen momenteel programmeerkennis van de eindgebruiker om interacties te specificeren. Daarnaast zijn ze niet geschikt voor dynamische gebruikersinterfaces, tenzij de gebruiker dit expliciet specificeert in de logica die geprogrammeerd wordt. Aangezien pixel-gebaseerde methodes gebruik maken van zichtbare pixels, kunnen ze typisch niet in de achtergrond uitgevoerd worden, omdat in dit geval de pixels niet zichtbaar zijn. Dit beperkt de gebruiker in het aantal acties dat er parallel uitgevoerd kunnen worden op de computer, aangezien elke actie de voorgrond vereist. Omwille van deze reden is er in deze thesis gekeken hoe pixel-gebaseerde methodes kunnen gebruikt worden voor niet-zichtbare pixels. Tenslotte vereist het koppelen van extra hardware aan acties op een computer vaak elektronica-kennis van een gebruiker. Omdat we niet van elke gebruiker de nodige programmeerkennis of technische achtergrond kunnen verwachten, zijn volgende onderzoeksuitdagingen opgesteld voor het werk voorgesteld in deze thesis:

1. Onderzoeksuitdaging 1: eindgebruikers voorzien met een tool om acties te automatiseren op een desktop zonder programmeerkennis of andere technische kennis te verwachten.

2. Onderzoeksuitdaging 2: een robuuste pixel-gebaseerde aanpak voor dynamische gebruikersinterfaces te automatiseren.

3. Onderzoeksuitdaging 3: een gebruiksvriendelijk programmeer-paradigma waarmee niet-programmeurs de fysieke voorwerpen kunnen koppelen aan geautomatiseerde desktop acties, zoals gedefinieerd in (2).

4. Onderzoeksuitdaging 4: overkom de inherente limitatie van pixel-gebaseerde methodes die vereist dat pixels zichtbaar zijn op het scherm vooraleer ze herkend kunnen worden.

Om de onderzoeksuitdagingen aan te pakken, is er een gebruiksvriendelijke toolkit geïmplementeerd die het mogelijk maakt willekeurige acties op een desktop te automatiseren en hier een fysiek voorwerp aan te koppelen. Sectie A.2 legt uit hoe de eerste onderzoeksuitdaging wordt aangepakt door de gebruiker een actie de laten demonstreren. Sectie A.3 legt uit welke maatregelen er getroffen zijn om de robuustheid van pixel-gebaseerde methodes te verbeteren. Sectie A.4 behandelt hoe een gebruiker op een eenvoudige manier de fysieke objecten aan een gedemonstreerde actie kan koppelen. Sectie A.5 gaat in op hoe pixel-gebaseerde methodes gebruikt kunnen worden om niet-zichtbare pixels te automatiseren. Tenslotte voorziet sectie A.6 een conclusie van deze Nederlandstalige samenvatting.
A.2 Programmeren door te demonstreren

Om de eerste onderzoeksuitdaging aan te pakken, gebruikt de toolkit die wordt voorgesteld in deze thesis een "Programming-By-Demonstration" aanpak. Deze aanpak maakt het mogelijk dat de gebruiker de te automatiseren actie slechts één keer demonstreert om deze te leren aan het systeem. Tijdens het demonstreren van de actie volgt het systeem de acties die de gebruiker uitvoert, en houdt de betekenisvolle acties bij. Deze acties kan het systeem vervolgens later gebruiken tijdens het opnieuw afspelen van een actie.

Voor een actie te demonstreren aan de toolkit interageert de gebruiker enkel met de toolkit en met de applicatie(s) waarin de automatisatie plaats vindt. Om ervoor te zorgen dat de gebruiker acties kan automatiseren in willekeurige applicaties, is de toolkit in staat om alle geïnstalleerde applicaties op de computer te achterhalen en te openen. Om ervoor te zorgen dat de gebruiker enkel interageert met de toolkit en met de te automatiseren applicatie(s), is de toolkit ontworpen als een full-screen applicatie, zodat de Windows taskbar en het bureaublad worden verborgen. Dit is nodig zodat de gebruiker niet rechtstreeks applicaties opent, maar enkel door gebruik te maken van de toolkit. Hierdoor kan de toolkit steeds de applicatie opnieuw opstarten vooraleer de gebruiker begint met de demonstratie, zodat elke demonstratie begint vanaf een gekende veilige staat van de applicatie.

Eens de gebruiker een applicatie heeft laten openen door de toolkit, kan de actie gedemonstreerd worden. Tijdens het demonstreren worden de globale muis en toetsenbord acties van de gebruiker geanalyseerd om te achterhalen welke acties de gebruiker specificeert. Aangezien dynamische gebruikersinterfaces veranderen doorheen de tijd, bijvoorbeeld door het toevoegen van items aan een lijst of het veranderen van aanbevelingen, is het niet nuttig om de posities waar de gebruiker klikt op te slaan voor verder gebruik. In plaats daarvan is er een algoritme ontworpen dat een schermafbeelding neemt van het interface element waarmee de gebruiker interageert, gebaseerd op de positie van de muis. Deze schermafbeelding wordt bij het afspelen van een actie vervolgens gebruikt door de pixel-gebaseerde methode om de juiste positie terug te vinden. Naast muis acties worden er ook toetsenbord acties bijgehouden. Deze vereisen geen interpretatie, aangezien deze gewoon opnieuw kunnen afgespeeld worden. De reden hiervoor is dat de klik die de focus geeft aan het juiste tekstveld ook zal nagespeeld worden, waardoor de juiste tekstvelden op de juiste momenten de focus zullen hebben.

A.3 Robuuste pixel-gebaseerde aanpak

Huidige pixel-gebaseerde aanpakken [22, 90] zijn vaak niet robuust voor dynamische gebruikersinterfaces. Bijvoorbeeld, Sikuli [90] voorziet een visuele scripting API waarmee gebruikers een interface kunnen automatiseren aan de hand van screenshots en programmeer-instructies zoals lussen en conditionele statements. Om ervoor te zorgen dat een script niet enkel werkt op het moment van specificatie, maar blijft werken nadat er een bepaalde periode verstreken is en de interface dynamisch is aangepast, dient de gebruiker reeds tijdens specificatie alle mogelijke veranderingen af te handelen om ervoor te zorgen dat het script zowel nu als in de toekomst op een
robuuste manier kan worden afgespeeld. Het is echter zeer moeilijk om te anticiperen op alle mogelijke dynamische veranderingen van een interface. Daarnaast is het ook moeilijk om bepaalde dingen visueel te specificeren, omdat ze niet steeds zichtbaar zijn. Om deze problemen te illustreren beschouwen we een voorbeeld van een script om een van de liedjes uit de lijst van "favoriete liedjes" in een muziek applicaties te starten. Tijdens het opstellen van het script zijn er slechts enkele liedjes in de lijst, waardoor alles in een beeld past. De gebruiker stelt een visueel script op om de lijst van favoriete liedjes te openen en vervolgens om op de play knop langs het af te spelen liedje te klikken. De gebruiker test het script en het werkt. In de toekomst voegt de gebruiker echter een liedje toe uit een album met dezelfde naam als het geautomatiseerde liedje. Nu vindt het script tweemaal de afbeelding met het liedje, en tweemaal staat er een afspeel knop langs. Welke knop moet het systeem nu gebruiken? De gebruiker past het script aan om de ambiguïteit op te lossen. In de toekomst worden er meer liedjes toegevoegd aan de lijst, waardoor er dient gescrolld te worden voor het liedje in kwestie zichtbaar te maken. De gebruiker moet wederom het script aanpassen. Deze aanpassing is echter niet eenvoudig, aangezien de scroll bar enkel zichtbaar wordt tijdens het scrollen, wat vaker het geval is in moderne interfaces.

Zoals geïllustreerd in voorgaand voorbeeld, zijn er vele zaken waar een gebruiker rekening mee dient te houden. Het systeem voorgesteld in deze thesis neemt deze last weg van de gebruiker en lost de besproken problemen algoritmisch op. Ambiguïteiten worden opgelost door, tijdens het demonstreren, automatisch de context rondom een widget bij te houden. Als er tijdens het terugzoeken van een widget meerdere overeenkomsten zijn, wordt de context gebruikt om een score te stellen voor elke overeenkomst, en de beste te kiezen. Zo overkomt het systeem ambiguïteiten die zich pas voordoen na demonstratie, en niet enkel tijdens demonstratie. Als het systeem niet in staat is ambiguïteiten algoritmisch op te lossen, of andere problemen of optimalisaties identificeert, wordt er een opvolg-vraag aan de gebruiker gepresenteerd, gelijkaardig aan HILC [39]. Daarnaast zijn er verschillende flows geïmplementeerd in de toolkit die gevolgd kunnen worden als een element niet meteen wordt teruggevonden, zoals een applicatie resetten naar een gekende veilige staat, automatisch proberen in bepaalde delen van een interface te scrollen, enzoverder.

### A.4 Fysieke objecten koppelen

Na demonstratie van een actie kan de gebruiker een fysiek voorwerp koppelen aan de actie om een fysieke representatie van de actie te krijgen. In feite is een "Tangible Proxy Interface" geen fysieke representatie van de hele actie, maar enkel van de laatste widget. Alle andere interface elementen waarmee de gebruiker acties heeft uitgevoerd, dienen enkel voor navigatie doeleinden doorheen de applicatie(s) en worden ook zo beschouwd. Door dit onderscheid te maken, is het makkelijk om op een gebruikersvriendelijke manier de fysieke objecten te koppelen. De gebruiker kan namelijk simpelweg een van de ontwikkelde fysieke voorwerpen plaatsen op de finale widget van een actie om deze te koppelen. Hiervoor is uiteraard een touch-screen vereist, zodat de positionering waargenomen kan worden. Om dit mogelijk te maken zijn de ontwikkelde fysieke objecten uitgerust met geleidend materiaal aan de onderkant in een specifiek patroon, gelijkaardig aan PUCs [84] en PERCs [83]. Om de
limitatie van PUCs [84] te overkomen, waarbij slechts een gelimiteerd aantal verschillende patronen kan herkend worden, bevatten de fysieke objecten ook een capacitieve sensor, waardoor er gemeten wordt als ze al dan niet worden vastgehouden door de gebruiker. Door te weten welk object wordt vastgehouden wanneer het patroon op een touchscreen herkend wordt, weet het systeem welk object er gekoppeld dient te worden.

De ontwikkelde fysieke objecten kunnen zowel invoer van de fysieke wereld naar de digitale omgeving voorzien, als uitvoer van de digitale omgeving naar de fysieke wereld. Er zijn drie verschillende invoer objecten ontworpen, en twee verschillende uitvoer objecten. De verzameling van ontwikkelde voorwerpen is zo gekozen dat er voor elk ondersteunde digitale widget minstens een fysieke tegenhanger is. De set is echter uitbreidbaar, zoals besproken in de thesis tekst. Naast de rechtstreekse overeenkomst tussen fysieke componenten en digitale widgets, maakt de positionering van een fysiek voorwerp bovenop een digitaal interface element ook andere interacties mogelijk. Bijvoorbeeld, als een slider niet in staat is om met een fysieke slider te werken, door bijvoorbeeld een fysieke beperking, kunnen er drie knoppen op een slider geplaatst worden, welke overeenstemmen met 0%, 50% en 100% van de slider. Een ander voorbeeld is wanneer een gebruiker geen fysieke toggle kan bedienen. In dit geval kan de fysieke toggle bijvoorbeeld vervangen worden door een fysieke slider, waarbij 0% op de slider correspondeert met de uit-staat van de toggle, en 100% met de aan-staat.

A.5 Pixel-gebaseerde methodes in de achtergrond

Het koppelen van meerdere acties zorgt met traditionele pixel-gebaseerde systemen voor een visuele warboel, omdat elke applicatie in de voorgrond moet staan zodat de juiste pixels zichtbaar zijn. Aangezien dit een serieuze limitatie is bij het koppelen van meerdere acties, zeker acties die in de achtergrond kunnen uitgevoerd worden, zoals het bedienen van de muziekspeler en het monitoren van inkomende mails bijvoorbeeld, is er een manier geïmplementeerd om pixel-gebaseerde manieren te gebruiken op niet-zichtbare pixels.

De manier die hiervoor geïmplementeerd is plaatst elke applicatie die geautomatiseerd is in een aparte virtuele desktop op Windows. Vervolgens wordt er een specifieke API gebruikt waarmee een schermafbeelding van elke draaiende applicatie kan worden gemaakt om schermafbeeldingen te nemen van de applicaties. Deze applicaties worden vervolgens verwerkt met de pixel-gebaseerde aanpakken, in plaats van ze rechtstreeks op de pixels van het scherm uit te voeren. Op deze manier kan elke onzichtbare pixel op dezelfde manier geanalyseerd worden als elke zichtbare pixel, wat het mogelijk maakt bepaalde acties in de achtergrond uit te voeren.

A.6 Conclusie

Binnen deze thesis is een gebruiksvriendelijke toolkit voorgesteld die het mogelijk maakt een willekeurige actie in een willekeurige applicatie op een computer te automatiseren door de actie slechts één keer te demonstreren. Om het mogelijk te maken
de actie op een robuuste manier terug af te spelen wordt er een combinatie gebruikt van algoritmes en opvolg vragen aan de gebruiker om met alle mogelijke scenarios rekening te houden. Na demonstratie van een actie kan er een fysiek object gekoppeld worden om de actie vanuit de fysieke wereld te bedienen, wat vele verschillende scenario's mogelijk maakt. Om het afspelen op een zo gebruiksvriendelijk mogelijke manier te laten verlopen, en de gebruiker hier zo weinig mogelijk last van te bezorgen, is er een manier ontwikkeld waarmee dat acties in de achtergrond kunnen worden uitgevoerd.
Bibliography


[34] Raimund Hucke. RaiMan’s SikuliX. URL: http://sikulix.com (visited on 04/10/2019).


[64] NuGet Gallery | Home. URL: https://www.nuget.org (visited on 03/04/2019).


[67] OpenCV library. URL: https://opencv.org (visited on 04/05/2019).


[70] PInvoke.NET. https://www.pinvoke.net. (Visited on 04/05/2019).


[76] Spark - A micro framework for creating web applications in Kotlin and Java 8 with minimal effort. URL: http://sparkjava.com (visited on 04/10/2019).

[77] Paul Stoffregen. CapacitiveSensor Library: detect touch or proximity by capacitive sensing. URL: https://github.com/PaulStoffregen/CapacitiveSensor (visited on 05/16/2019).


[86] w3schools. Browser Display Statistics. URL: https://www.w3schools.com/browsers/browsers_display.asp (visited on 05/01/2019).

[87] Waveshare. 480x320, 3.5 inch Touch Screen IPS TFT LCD Designed for Raspberry Pi. URL: https://www.waveshare.com/3.5inch-rpi-lcd-b.htm (visited on 05/11/2019).


[89] Wiring 128x64 OLEDs. URL: https://learn.adafruit.com/monochrome-oled-breakouts/wiring-128x64-oleds (visited on 05/15/2019).
Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:

**Programming Tangible Proxy Interfaces using GUI task demonstrations and follow-up questions**

Richting: **master in de informatica**  
Jaar: **2019**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen -, aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

**Veuskens, Tom**

Datum: **5/06/2019**