TEXT2TEST: FROM NATURAL LANGUAGE DESCRIPTIONS TO EXECUTABLE TEST CASES USING NAMED ENTITY RECOGNITION

by
AHMET YASIN AKYILDIZ

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
July 2023

TEXT2TEST: FROM NATURAL LANGUAGE DESCRIPTIONS TO EXECUTABLE TEST CASES USING NAMED ENTITY RECOGNITION

Approved by:

Prof. Cemal Yilmaz .....
(Thesis Supervisor)

Prof. Hüsnü Yenigün .....

Prof. Hasan Sözer .....

Date of Approval: July 25, 2023

ABSTRACT

TEXT2TEST: FROM NATURAL LANGUAGE DESCRIPTIONS TO EXECUTABLE TEST CASES USING NAMED ENTITY RECOGNITION

AHMET YASIN AKYILDIZ

Computer Science and Engineering, Master's Thesis, 2023

Thesis Supervisor: Prof. Cemal Yilmaz

Keywords: Mobile automation, Mobile application testing, GUI testing, Natural Language Processing, Named Entity Recognition

In this work, we present text2test, an innovative approach for automated testing of mobile application user interfaces (UIs). As mobile applications become increasingly prevalent, ensuring robust and user-friendly UIs has become essential, leading to a greater need for efficient testing methodologies. However, testing UIs poses challenges due to varying screen sizes, evolving UI elements across versions, and the need for frequent test case revisions. To address these challenges, we propose text2test, which combines named entity recognition (NER) and semantic similarity computations in a framework using Android APIs to execute test cases from natural language descriptions. We bridge the gap between textual input and UI interactions by training a NER language model to identify UI elements and actions from natural language test case descriptions. Using the DOM structure of an application, containing XML metadata of UI elements, we accurately detect the appropriate UI element associated with the action. Finally, using the information extracted by the NER model and the elements detected using semantic similarity we developed a framework that can execute test cases on Android applications. Our experiments show that text2test achieves a 92% precision rate in identifying element-action pairs and an average accuracy of 88% in detecting expected UI elements and a 76% success rate to fully reproduce test cases on Android applications. Our approach streamlines automated UI testing, reducing manual intervention and the need for frequent script updates and promises a solution for efficient and reliable UI testing.

ÖZET

TEXT2TEST: VARLIK İSIMLERI TANIMA TEKNIKLERI KULLANARAK DOĞAL DIL'DE YAZILMIŞ TEST CÜMLELERINDEN YÜRÜTÜLEBILIR TEST SENARYOLARINA

AHMET YASIN AKYILDIZ

Bilgisayar Bilimi, Yüksek Lisans Tezi, 2023

Tez Danışmanı: Prof. Dr. Cemal Yılmaz

Anahtar Kelimeler: Mobil otomasyon, Mobil uygulama testi, GUI testi, Doğal Dil İşleme, Varlık İsmi Tanıma

Bu çalışmada, mobil uygulama kullanıcı arayüzlerinin (UI'ler) otomatik testi için yenilikçi bir yaklaşım olan text2test'i sunuyoruz. Mobil uygulamalar giderek daha yaygın hale geldikçe, kullanıcı dostu kullanıcı arayüzlerinin sağlanması önemli hale geldi ve bu da verimli test metodolojilerine daha fazla ihtiyaç duyulmasına yol açtı. Bununla birlikte, değişen ekran boyutları, sürümler arasında değişen UI öğeleri nedeniyle test durumu revizyonları gereksinimi UI testlerini zorlaştırır. Bu zorlukların üstesinden gelmek için, Varlık İsimleri Tanıma (NER) ve anlamsal benzerlik hesaplamalarını kullanarak doğal dil cümlelerinden test senaryolarını yürüten Android API'lerini kullanan bir yazılım text2test'i sunuyoruz. Doğal dil test senaryolarının açıklamalarından UI öğelerini ve eylemleri tanımlamak için bir NER modeli eğiterek UI öğeleri ve UI etkileşimleri arasındaki boşluğu dolduruyoruz. Bir uygulamanın UI öğelerinin XML meta verilerini içeren DOM yapısını kullanarak, belirlenen eylemle ilişkili uygun UI öğesini doğru bir şekilde tespitini sağlıyoruz. Son olarak, NER modelimiz tarafından çıkarılan bilgileri ve semantik benzerlik ile tespit edilen öğeleri kullanarak, Android uygulamalarında test senaryolarını yürütebilen bir yazılım geliştirdik. Deneylerimiz, text2test'in öğe-eylem çiftlerini belirlemede %92'lik bir kesinlik oranı ve beklenen UI öğelerini tespit etmede ortalama %88'lik bir doğruluk oranı ve Android uygulamalarında test senaryolarını tam olarak yeniden oluşturmak için %76'lık bir başarı oranı elde ettiğini göstermektedir. Yaklaşımımız, manuel müdahaleyi ve komut dosyası güncelleme ihtiyacını azaltır ve verimli ve güvenilir UI testi için bir çözüm olanağı sunar.

ACKNOWLEDGEMENTS

I would like to thank my advisor Profesor Cemal Yilmaz for his endless support, and my family and friends for supporting me throughout my work in Sabancı.

To my family & friends

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

User interface testing is time-costly due to the need to cover every UI element functionality and logical interaction flow. For example, a test suite developed for an Android application needs 75% code-based adaptations and 100% image-based test case adaptations because of the UI changes between the two versions. Coppola, Raffero & Torchiano (2016). The need for UI adaptation in case of UI changes is also true for other test suites developed by technical stakeholders for any Page Object Model (POM) interface that utilizes XML or HTML documents. As a result, involving non-technical stakeholders in test automation is of great practical importance in today's software industry, mainly due to the ever-increasing cost of developing executable test cases using technical stakeholders.

To this end, many approaches have been developed Liu, Lu, Cheng, Chang, Hsiao & Chu (2014); North & others (2006); SmartBear (2019); Solis & Wang (2011), some of which have, indeed, been quite frequently used in the field, including behaviour-driven development (BDD) Li, Escalona & Kamal (2016); Soeken, Wille & Drechsler (2012) and capture-and-replay tools Bernal-Cárdenas, Cooper, Moran, Chaparro, Marcus & Poshyvanyk (2020). However, due to their fragile nature, both approaches suffer when there is a change in the UI arrangement. There is still room for improvement in the effectiveness and efficiency of these approaches.

For example, BDD allows non-technical stakeholders to develop executable test cases using semi-structured natural language sentences. However, it only partially eliminates the need for technical stakeholders as sentence-specific scripts (e.g., step definitions) are still required to be developed to execute the sentences against the system under test (SUT). An alternative approach would be to use a capture-and-replay tool Halpern, Zhu, Peri & Reddi (2015). Although these tools are easy to learn/use and non-technical stakeholders can develop executable test cases without requiring any technical stakeholders, the resulting test scripts are typically fragile in the presence of user interface (UI) changes. Small changes in the UIs, such as changing the screen size, a change in resolution and layout, and changes in labels/icons, can break the pre-recorded scripts. Indeed, BDD also suffers from the same issue. More

specifically, in the presence of UI changes, the scripts responsible for executing the sentences may need to be modified to accommodate the changes.

Our ultimate motivation is to create a UI test framework that is resilient to UI changes, unlike BDD and capture-and-replay thus enabling non-technical stakeholders to significantly contribute to the testing efforts with minimal help from stakeholders.

As a general assumption, we assume that when a stakeholder prepares human-readable test cases for functional testing, these test cases provide sufficient information for automation, which are the UI element (entity) to be interacted with, the type of interaction (action) and the state or the value which the UI element can be assigned (parameter).

Our ultimate goal in this field of research is to enable non-technical stakeholders to significantly contribute to the test automation efforts by developing executable test cases without requiring any technical stakeholders, which are more resilient to the changes in the UIs.

To this end, we, in this work, present an approach called *text2test*. At a very high level, *text2test* takes as input a screen represented in the form of a page object model (POM) and a sentence written in a natural language (in our case, in English) describing an interaction with a UI element on the screen. The output is the word(s) describing the action, the actual UI element on the screen on which the action needs to be carried out and the parameter that the UI element can be assigned. Figure 1.1 presents an example. Given the sentence *tap to the call friends icon in the menu* (Figure 1.1a), which needs to be executed on the screen in (Figure 1.1b(, the proposed approach figures out that the *invite friends* button needs to be clicked since it is the most semantically similar UI element on the screen to one described in the sentence. Then the test case execution tool developed by us performs the *click* action on the *invite friends* element that is present on the application user interface. In order to check if we performed the correct action on the correct UI element we check it manually. For this purpose, a test oracle can be developed as a future work idea.

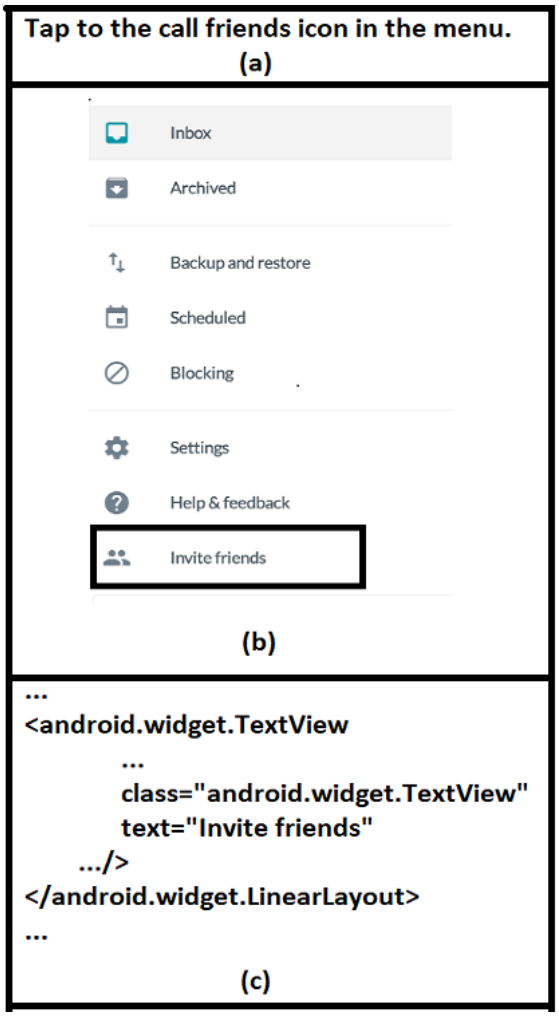

Figure 1.1 An example sentence to execute (a), the corresponding screen (b), and its POM representation (c).

We, in particular, use a well-known natural language processing (NLP) approach called *named entity recognition* (NER). Researchers and data analysts widely use NER in many domains for information retrieval Limsopatham & Collier (2016); Mahalakshmi, Vijayan & Antony (2018); Weber, Sänger, Münchmeyer, Habibi, Leser & Akbik (2021). Our work is different in that we use it (and, to the best of our knowledge, for the first time) as an aid to develop executable test cases by compiling sentences in natural language into executable user interactions. Indeed, the results of our initial set of experiments suggest that we can effectively use NER for this purpose.

We organized the remainder of the paper as follows: Section 2 summarizes the related work and provides background information; Section 3 presents the proposed approach; Section 4 shares our empirical evaluations of the proposed approach; and Section 5 concludes with a discussion of the success of our intended approach.

2. RELATED WORK & BACKGROUND

2.1 Software Systems Testing

The ever-increasing need for high-quality software systems necessitates the development of cost-effective testing tools and processes Gao, Bai, Tsai & Uehara (2014); Kirubakaran & Karthikeyani (2013); Muccini, Di Francesco & Esposito (2012). In their work Muccini et al. state that the domain of mobile devices is growing exponentially, which also comes together with exponential growth in the mobile application domain. The biggest reason for this growth is that, during recent years, software systems started to take part in crucial sectors such as banking, security and even human health. Therefore, the necessity of software systems testing is increasing massively. However, testing software systems is costly.

One way to reduce the cost of testing is to enable the non-technical stakeholders to contribute to the test automation efforts significantly Liu et al. (2014); Northet al. (2006); SmartBear (2019); Solis & Wang (2011). Behaviour Driven Development (BDD) offers automated testing capabilities that let non-technical stakeholders verify the software's behaviour without needing in-depth technical knowledge by utilizing BDD tools like Cucumber. Liu et al., in their research of Capture-replay testing, show that it enables testers to quickly generate test cases by recording user interactions and replaying them, reducing the time and effort required for manual testing. This work presents a different approach to help non-technical stakeholders develop executable and *resilient* test cases.

Many researchers have proposed approaches for end-to-end testing of software systems Amalfitano, Fasolino, Tramontana, De Carmine & Imparato (2012); Anand, Naik, Harrold & Yang (2012); Costa, Paiva & Nabuco (2014); Gao et al. (2014); Moreira & Paiva (2014); Song, Qian & Huang (2017); Su, Meng, Chen, Wu, Yang,

Yao, Pu, Liu & Su (2017). According to Gao et al., testing mobile applications is challenging because of the wide range of device and platform configurations, the dynamic nature of mobile environments, and the intricate relationships between mobile apps and back-end systems. The other authors suggested solutions to these problems. For instance, Amalfino et al. provided a program that builds test cases using the extracted XML format hierarchy and extracts the GUI hierarchy of an Android application. Costa et al. capture the mobile application's GUI events and transform them into a series of patterns. Then the authors use these patterns to create test cases. Likewise, Song and colleagues model the components and interactions of the application, which is then applied to simulate the application's behaviour and produce test cases.

Similarly, Moreira et al. suggest the PBGT tool, which automatically creates test cases based on the model of the application using pattern recognition algorithms. On the other hand, Anand et al. contend that traditional black-box testing is insufficient to test smartphone applications and use concolic execution to explore the program's execution paths and generate inputs that meet specific coverage criteria. Finally, Su et al. combine stochastic model-based testing with guided exploration to generate test cases. Our work differs from those works mentioned above in that we ultimately aim to compile instructions given in natural language to UI interactions.

A broad spectrum of domains uses NER and other NLP approaches extensively for information retrieval purposes Mahalakshmi et al. (2018); Rocktäschel, Weidlich & Leser (2012), including software engineering Ernst (2017); Fischbach, Vogelsang, Spies, Wehrle, Junker & Freudenstein (2020); Granda, Parra & Alba-Sarango (2021); Tao, Gao & Wang (2017); Zhou, Li & Sun (2020). Furthermore, there have been significant advancements in training specialized NER models and their assessments in recent years. Lin et al. suggested adding a neural adaptation layer to a pre-trained NER model to adapt it to various domains Lin & Lu (2018) and Liu et al. worked on the CrossNER framework for assessing the effectiveness of NER systems in cross-domain scenarios. Today, several open-source libraries are available for this purpose, including Stanford NER, NLTK, SpaCy and the BERT transformers.

NER has great potential in our application domain. For example, not only the actions and the UI elements but also the arguments passed to the actions (e.g., enter the text ... in the text field ...), and even the domain-specific terms (e.g., choose ... as the departure airport) can be tagged using NER, significantly increasing the applicability of NER in UI testing domain.

Using NER in application testing is not an entirely new idea Lin, Wang & Chu (2017a); Mahalakshmi et al. (2018); Tao et al. (2017). Tao et al. mapped natural

language descriptions to BDD step definitions. Their approach, however, requires technical stakeholders for the development of the step definitions Tao et al. (2017). Lin et al. used semantic similarity to improve the effectiveness of crawling-based web application testing Lin et al. (2017a). Mahalakshmi et al. used NER to identify domain-specific terms in use cases so that they could infer abstract test scenarios Mahalakshmi et al. (2018). However, they are not concerned with actual screens or UI elements on these screens and the actions required for generating executable test cases. Our work is different in that we use NER to compile natural language descriptions for interactions eventually.

2.2 Named Entity Recognition

NER is a sub-task of natural language processing. A substantial task of NLP is understanding a natural language text without losing any information. In this task, NER is an information extraction method that locates and classifies named entities into predefined categories. A simple example for NER would be "Mark Zuckerberg is one of the founders of Facebook, a company from the United States" containing the entities Person – Mark Zuckerberg, Company – Facebook and Location – United States.

The improvement of NER systems began with rule-based approaches. Supervised learning Palmer & Day (1997), semi-supervised learning Brin (1998), and unsupervised learning Etzioni, Cafarella, Downey, Popescu, Shaked, Soderland, Weld & Yates (2005) were different approaches to training NER models. However, these early approaches needed more flexibility and performed poorly on complex data sets. Later research worked on machine-learning approaches that worked with statistical models to learn from annotated data. McCallum et al. implemented maximum entropy Markov models for McCallum, Freitag & Pereira (2000) and Laffrey et al. introduced conditional random fields (CRFs) Lafferty, McCallum & Pereira (2001) both are probabilistic approaches to sequence labelling tasks and achieved more robust scores that had better accuracy and recall over the rule-based approaches.

Although probabilistic approaches produced higher success rates at NER tasks, with the popularisation of neural networks in machine learning, researchers started exploring various network architectures, which indeed did improve the scores of the probabilistic approaches. Lample et al. evaluated convolutional neural networks (CNNS), recurrent neural networks (RNNs) and the combination of both for NER

tasks. Lample, Ballesteros, Subramanian, Kawakami & Dyer (2016) A remarkable result they emphasize is that when they use neural networks in a cascade, they outperform sequestered counterparts. Since the cascading of neural networks yielded better recognition results, Ma et al. introduced a Bi-directional Long Short Term Memory LSTM-CNN-CRF model for sequence labelling. Ma & Hovy (2016)

Long Short Term Memory (LSTM) is explicitly designed to avoid long-term dependency problems of recurrent neural networks (RNN), which is a problem when the gap between the relevant information and the predicted term in a sentence is long. Karpathy, Johnson & Fei-Fei (2015) Implementing a state vector achieves this memory. The state vector s_t consists of 2 parts: cell state c_t, which is the long-term memory where long-term that preserves dependencies and hidden state h_t, the short-term memory that controls the gates in a cell and decides what will go into the long term memory. The gates controlled by the hidden state are the input gate that is updated using i_t and C'_t, where it decides which new information it is going to store in the cell state, forget gate f_t decides which information is going to be thrown away and output o_t gates. An LSTM cell can be seen in figure: 2.1

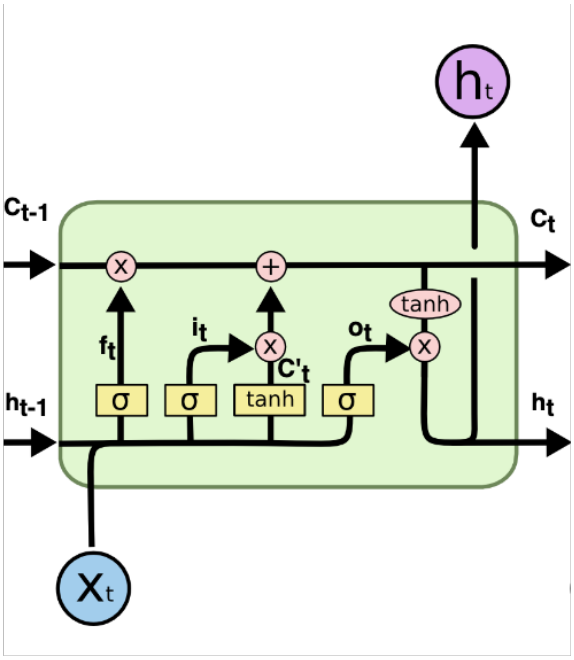

Figure 2.1 An LSTM cell showing the four interacting layers

$$f_t = \sigma(W_f \cdot [h_{t_{-1}}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [h_{t_{-1}}, x_t] + b_i)$$

$$o_t = \sigma(W_o \cdot [h_{t_{-1}}, x_t] + b_o)$$

$$C'_t = tanh(W_C \cdot [h_{t_{-1}}, x_t] + b_C)$$

$$C_t = f_t * C_{t_{-1}} + i_t * C'_t$$

$$h_t = tanh(c_t)$$

Conditional Random Fields are used for building probabilistic models to segment and label sequence data. (Lafferty et al., 2001) For an input sentence $X = (x_1, x_2, ..., x_n)$and corresponding predictions of labels $y = (y_1, y_2, ..., y_n)$, the CRF score for this sequence can be calculated as:

$$s(X, y) = \sum_{i=1}^{n} T_{y_{i-1}, y_i} + P_{i, y_i}$$

Where T is the tagging transition matrix which shows the likelihood of transitioning from tag y_{i-1} to y_i. Then a softmax function is used to extract a conditional probability of the path y by normalizing the above score over all possible tag paths y':

$$p(y|X) = \frac{e^{s(X_i, y)}}{\sum_{y'} e^{s(X_i, y')}}$$

As a last step, predict the best tag path that obtains the maximum score calculated using the Viterbi (Viterbi, 1967) algorithm:

$$argmax_{y'} s(X, y')$$

Another significant improvement in named entity recognition tasks is the utilization of transformers, especially Bidirectional Encoder Representations (BERT). Devlin et al. fine-tuned the BERT model on the CoNLL-2003 dataset. Devlin, Chang, Lee & Toutanova (2019) CoNLL-2003 is a standard benchmark test for NER. Tjong Kim Sang & De Meulder (2003) The authors reported that the BERT-based NER model outperformed the previous state-of-the-art approaches on this task.

BERT is a powerful model that can be used successfully for text-based learning. The bi-directional nature of BERT allows learning the word's context from its surroundings (both words from its right and left). Base BERT models are trained with two objectives masked language modelling (MLM) and next sentence prediction (NSP). MLM objective masks some percentage of the input tokens at random and updates model weight by making predictions on these masked tokens, thus learning contextual relationships between words. NSP objective, on the other hand, allows the model to understand relationships between two sentences which the MLM

cannot capture. (Devlin, Chang, Lee & Toutanova, 2018). However, it is very computationally expensive to train a new BERT model from blank for any task in hand and requires enormous data. The effective approach of transfer-learning on BERT models (Agrawal, Tripathi, Vardhan, Sihag, Choudhary & Dragoni, 2022) allows BERT-based NER models to perform with higher scores, particularly when the training data is limited.

2.3 Domain Specific Named Entity Extraction

At first glance, the main objective of named entity extraction may only seem to detect entities such as persons, locations, and companies. Recently it has been used in domain-specific areas. The language employed in a particular field or subject area is frequently technical or specialized, with a vocabulary not used in other settings. Applying general-purpose NER models to extract named entities in such domains may be challenging. Domain-specific NER models are trained on a specialized dataset that contains examples of named entities and the corresponding labels within the particular domain or topic. The specialized datasets might contain books on biomedicine, court cases, financial statements, and more. A domain-specific NER model can attain higher accuracy and performance in detecting named entities within that area or subject.

Zhang et al. fine-tuned a BERT model on a financial NER dataset with entities such as company names, stock codes and general financial terms. Zhang & Zhang (2022) Choudhary et al. showed the effectiveness of domain-specific NER models by comparing them to the performance of base models on tweets. Ritter, Clark, Mausam & Etzioni (2011). In order to extract menu items from online user reviews for restaurants, Syed et al. propose the "MenuNER" method. To optimize performance on the domain-specific job of detecting menu elements, the authors combine extended feature vectors created by concatenating domain-adapted **BERT embeddings**, **character embeddings**, and **part-of-speech (POS)** tag features with a Bi-LSTM+CRF model. Syed & Chung (2021)

Word Embeddings used by the authors are vector representations of words in a corpus of text. They are numerical representations that capture the semantic meaning of words and their relationships to other words in the dataset. Usually, word embeddings are learned by an unsupervised learning method using a large corpus of text data. Although more conventional methods of learning word embeddings are

word2vec Mikolov, Chen, Corrado & Dean (2013a) and Glove Pennington, Socher & Manning (2014), vector representations used in the last layers of BERT models can also be helpful as word embeddings for downstream tasks. POS provides linguistic information on how a word is being used within the scope of a sentence. They can also distinguish the meaning or explain a word's syntactic role, allowing a language model to infer semantic information from how a syntactic role is commonly used semantically. Character-Level Embeddings is a way to capture morphological and structural information about words. Using CNNs, Ling et al. generated character-level embedding representing words as sequences of vectors that capture the word's internal structure. Ling, Luís, Marujo, Astudillo, Amir, Dyer, Black & Trancoso (2015) They contribute to performance improvements, especially in morphologically rich languages. Combining different embedding types as input to the downstream Bi-LSTM-CRF task improves the accuracy of the named entity recognition.

Although NER is a popular tool to detect entities, it has not been used in the software testing domain. Other approaches to automated test validation use different methods to extract entities from bug reports. Fazzini et al. automated method named Yakusu to convert bug reports into test cases for mobile apps. Fazzini, Prammer, d'Amorim & Orso (2018). This process involves lexicon normalization and object standardization on the textual bug reports. Lexicon normalization replaces non-standard words with standard action words, making it easier to parse and understand actions. Object standardization replaces phrases referring to UI elements with their corresponding IDs in the app's ontology. Yakusu then examines each clause in the bug report: if the root of the clause is a predefined action word, it extracts an abstract action based on the clause's dependency tree.

Another approach to extracting domain-specific named entities is to identify relevant components and their dependencies using a dependency parser. Zhao et al. used dependency parsing on bug reports to reproduce Android application crashes. They extract the actions and the related elements from the bug report sentences using a combination of dependency parsing and a rule-based approach and generate a set of execution paths that lead to a crash. Zhao, Su, Liu, Zheng, Wu, Kavuluru, Halfond & Yu (2022); Zhao, Yu, Su, Liu, Zheng, Zhang & G.J. Halfond (2019). Yakusu and ReCDroid have demonstrated high success rates in extracting elements from issue reports. However, both approaches rely on a predefined vocabulary to identify and classify actions in issue reports.

2.4 Element Detection

In order to address the manual, labour-intensive configuration requirements of crawling-based application model generation Lin, Wang & Chu (2017b) proposes an unsupervised approach to detect the purpose of applications and UI elements in the application using a semantic similarity-based method. Application page object model (POM) is used to extract a vector representation of related text, and it is matched against a labelled corpus to determine the topic and action. A TF-IDF-based vector is used for semantic similarity calculations, which precedes the more recent word vector-based methods. However, in the work of Lin et al. (2017b), the general purpose of using semantic similarity is to identify the input topics of Web application elements such as a password or email text fields, while our purpose for semantic similarity is to detect the particular UI element for the given sentence.

A **page object model or POM** is a fundamental unit of test automation frameworks like Selenium and Appium. It can be an HTML or XML file, depending on the application under test, and it maps out an entire page as a document composed of a hierarchy of nodes.

A well-known semantic-similarity calculation method is the Google Universal Sentence Encoder. Cer, Yang, yi Kong, Hua, Limtiaco, John, Constant, Guajardo-Cespedes, Yuan, Tar, Sung, Strope & Kurzweil (2018). The main idea is to encode text into high-dimensional vectors that capture the semantic meaning of a given sentence. Google's AI research team introduced two versions of the encoder: the deep averaging networks (DAN) and the transformer model. DAN averages the embeddings of the words in a given phrase and computes a sentence embedding; it is a quick and practical approach to computing sentence embeddings. The transformer-based approach takes advantage of the self-attention mechanism that captures long-range information in the sentence and focuses on different parts of a given phrase in each step, resulting in highly effective sentence encoding.

3. APPROACH

This chapter describes the architecture of *text2test* shown in Figure 3.1. *text2test* consists of three major parts - element extraction, element detection and test case execution. To perform element extraction we employ a well-known NLP technique *named entity recognition* to extract *action*, *entity* and *parameter* information from test case sentences. We are utilizing Universal Sentence Encoder's word2vec representation to detect entities on an application UI and calculate similarity scores. Finally, we use information extracted from the sentences and the detected elements from the application in our test case execution approach. Test case execution is based on a customized depth-first search algorithm of the application from launch till the execution of all the test cases.

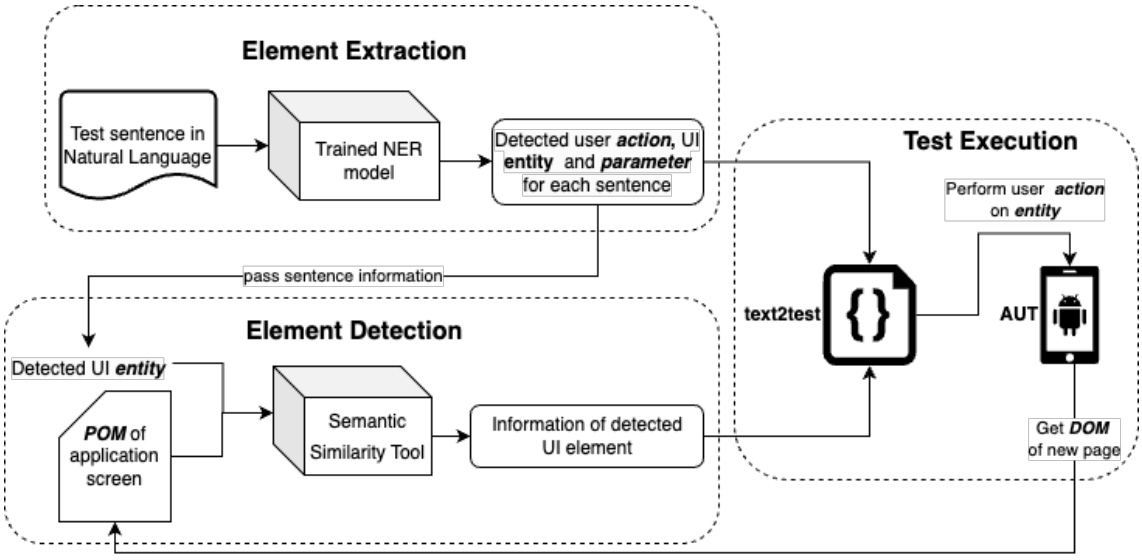

Figure 3.1 General Structure of the Proposed Approach

3.1 Element Extraction via NER Model

Element extraction from UI test sentences is a challenging task. Given a sentence, we require to recognize an *action* and a corresponding *entity* and, in some cases, *parameters*, which are relatively unseen tokens. Overcoming this challenge requires first adapting a base BERT model to the specific domain of applications, followed by creating a NER approach that can handle the low data limitation of the user-generated UI test case sentences.

3.1.1 Domain-Specific BERT

We are pre-training the BERT-base cased model from HuggingFace for our task. (Wolf, Debut, Sanh, Chaumond, Delangue, Moi, Cistac, Rault, Louf, Funtowicz & Brew, 2019) To pre-train the BERT-base cased model, we are using the MLM task with sentences from our domain-specific corpus. Using MLM task on the test sentences is basically giving the model a fill-in-the-gaps task. In our case, this helps the model learn relationships between words seen together in UI testing sentences. Towards this task, first, we pre-process domain-specific corpus data into training examples following the methodology used for LM training in the original BERT paper, then train the BERT base with the regenerated data. Pre-training allows the model to make improved predictions on a specific domain and task while retaining its general language understanding. (Huggingface, 2019) The domain we mention here is the UI of a mobile platform, web or stand-alone application, and the sentences are UI functionality test sentences specific to these applications. Moving forward, we will call this domain-specific BERT model as *UI-BERT* since the fine-tuning data is from test sentences of application UI testing. We describe the detailed information on the domain corpus in Section 4.3.

An example from a UI test sentence is *Open a new [MASK].* The pre-trained BERT model on Books and Wikipedia corpus' first two predictions for the masked word are *door* and *book*, when we perform the same unmasking task with *UI-BERT* the first two predictions become *website* and *page* which are the better predictions for our case.

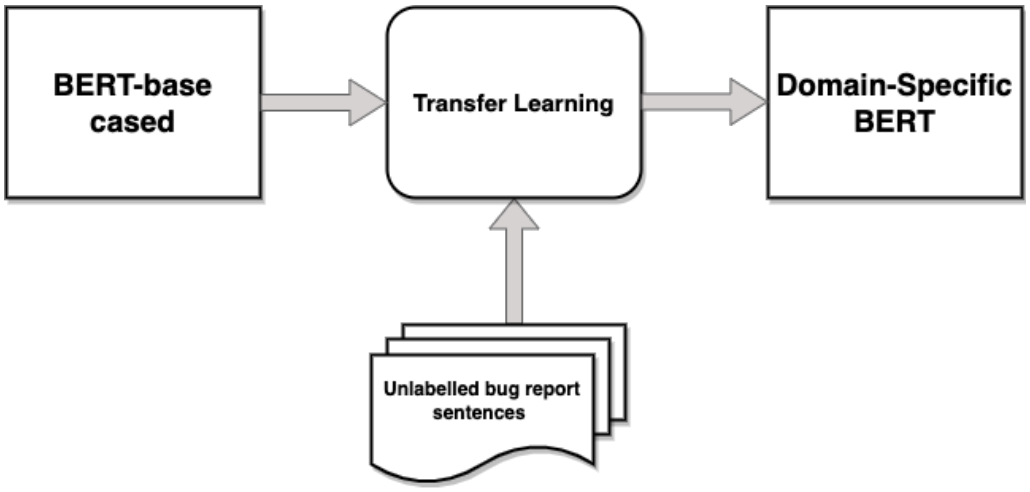

Figure 3.2 BERT Pre-training

3.1.2 NER Model

Our implementation of the NER approach to extract elements from user test sentences uses the well-tested architecture of using a Bi-LSTM-CRF model as a downstream task to the fine-tuned *UI-BERT* (Dai, Wang, Ni, Li, Li & Bai, 2019) (Souza, Nogueira & de Alencar Lotufo, 2019). Also, we are using the embedding layer technique adapted by MenuNER concatenating word embeddings with the part-of-speech (POS) tag embedding and character embeddings (Syed & Chung, 2021), which also utilizes a Bi-LSTM-CRF model as a downstream. See Figure3.3 for the proposed architecture.

3.1.2.1 Input Layer and Embeddings

Word embeddings aim to represent words in a lower dimension using vectors, allowing higher accuracy on NLP tasks with a much lower computational cost. (Mikolov, Chen, Corrado & Dean, 2013b) In our work, we represent word embedding with a concatenation of 3 different levels of embeddings: POS-tag, word and character embeddings.

In a given sentence of word sequence $S = \{w_1, w_2...., w_n\}$, feature vector is composed of the word-level embedding $e_i^w \in \mathbb{R}^{d_w}$, POS-feature embedding $e_i^p \in \mathbb{R}^{d_p}$ and character-level embedding $e_i^c \in \mathbb{R}^{d_c}$. The final word embedding is $V_i = e_i^w \oplus e_i^p \oplus e_i^c$, where e_i^w, e_i^p and e_i^c are embeddings of dimensions d_w, d_p and d_c vector space and $i \in \{1, 2, ..., n\}$.

We obtain **Word embeddings** from *UI-BERT*. Specifically, we concatenate the embeddings of the last four layers of *UI-BERT*. In their research, the authors of BERT (Devlin et al., 2018) reported that concatenating the embeddings of the last four layers yields the best results for a feature-based approach. This concatenation can be achieved by mean pooling the hidden states of these layers, allowing us to capture a given word's semantic and syntactic features. We enhance the performance of our NER task by creating word embeddings to properly reflect the complex and varied meanings of words in their context.

As part of our approach, we utilize **Part-Of-Speech (POS)** embeddings to provide linguistic information on how words are used within a sentence's context. These embeddings enable us to distinguish a word's meaning and syntactic role, allowing our language model to infer semantic information from how a syntactic role is commonly used semantically.

To improve the performance of *UI-BERT*, we also leverage **character-level embeddings** in our embedding layer. Our primary motivation is to use them to improve the learning of out-of-vocabulary words. For example, if an input field is labelled as *credit_card_num* in the POM of a UI but if the test sentence includes *credit card*, then we believe that character level embeddings will help us capture the relationship between the two. We are directly implementing the same approach to generate char embedding from the CharCNN feature of the MenuNER, which is a 1-dimensional ConvNets (Zhang, Zhao & LeCun, 2015).

3.1.2.2 Bi-LSTM CRF Downstream task

The Bidirectional Long Short Term Memory (Bi-LSTM) and Conditional Random Fields (CRF) architecture is a commonly used model for NER tasks. We use this architecture as a downstream task of the *UI-BERT* to fine-tune our NER approach, which takes in the input embeddings explained in the previous subsection 3.1.2.1.

Long Short Term Memory (LSTM) In our implementation, we use a Bi-LSTM network concatenating forward and backward state vectors to extract short and long-term dependency information from a sentence in both directions to improve the quality of our entity predictions by capturing long-term dependencies. As we would like to identify the boundaries of named entities (which vary in length and structure), we are utilizing the ability of LSTMs to maintain context information over sequences of text and the ability to remember and forget information selectively.

In addition to their ability to hold context information, they can also handle input sequences of variable lengths. Handling variable length sequences is necessary for our application, where the length of named entities can vary significantly.

Conditional Random Fields (CRF) This study uses Conditional Random Fields (CRFs) as a significant component of our NER approach. We select CRFs because they can model the relationships between nearby labels, enabling more precise and reliable predictions of named entities. CRFs explicitly describe the likelihood of each label given its neighbours' labels and the probability of each label given the input. CRFs can manage numerous label types at once, making them an ideal choice for classifying both named entities and their corresponding labels. With the use of CRFs, we aim to enhance the precision and functionality of our NER system on a global scale.

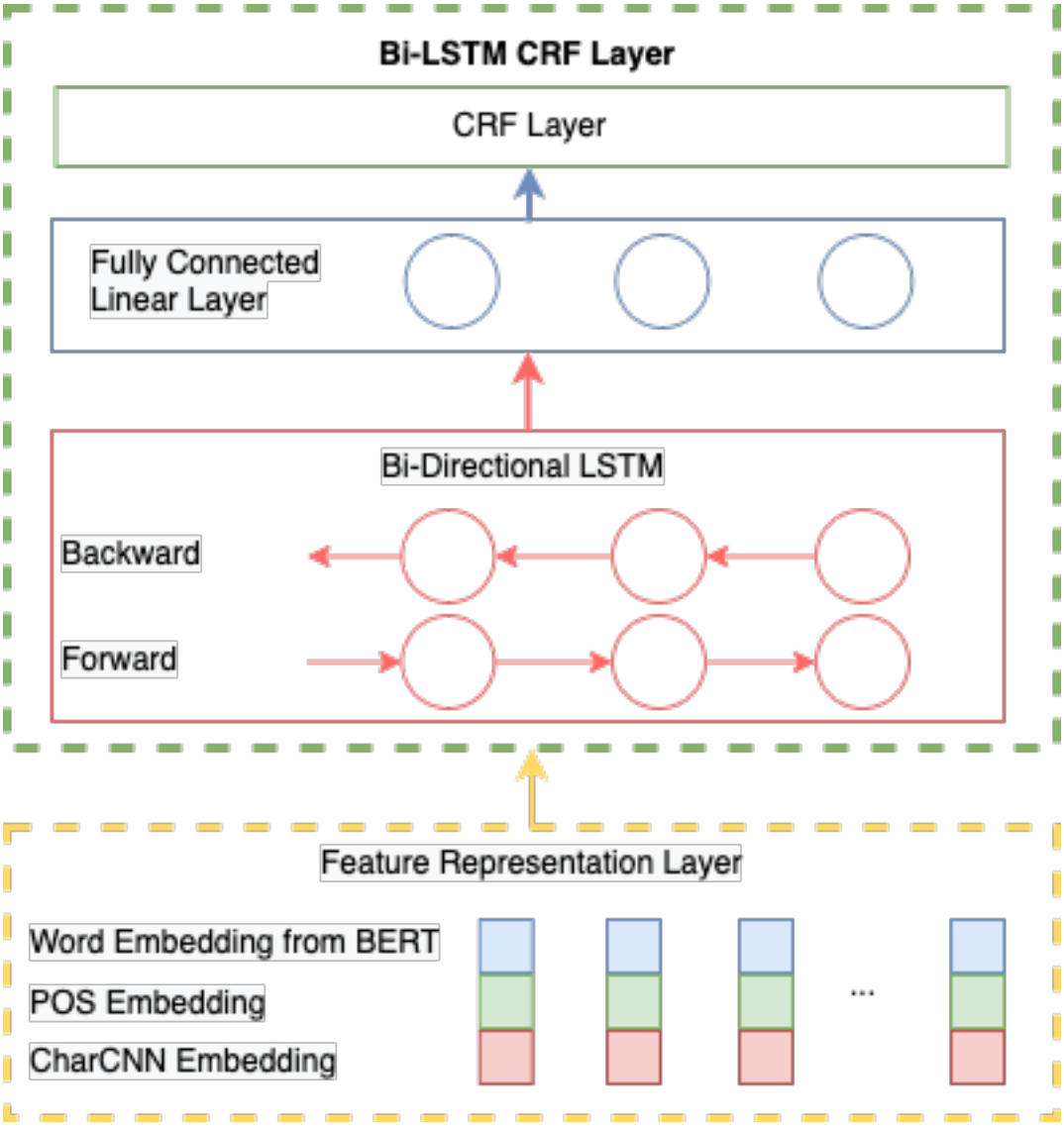

Figure 3.3 Proposed architecture for android sentences action and entity recognition

3.2 Element Detection with Similarity Score Calculation

After the NER model tags the words, the next step is to use the words tagged with the element labels (i.e., the element words) to identify the actual UI element on the screen. In the identification step, we compute the semantic similarities between the element words extracted from the test sentence and the attributes extracted from the POM structure of the given screen. We opted to evaluate the proposed approach on the Android platform for this work. As such, we obtain the POM models of the screens by using Android Debugging Bridge Developers (2021). However, the proposed approach readily applies to other POM models, which can be obtained from different mobile and Web platforms.

To compute the semantic similarities, we, in particular, utilize the *text* and *content-description* attributes (if present) of the elements in the POM model. To compare semantic similarity approaches, we first utilize *UI-BERT* to compute the word embeddings of the element words from test sentences and each attribute present in the POM and calculate the cosine similarity between each embedding. Secondly, we extract the embeddings utilizing the DAN and transformer version of the Google Universal Sentence Encoder and again calculate cosine similarity.

One observation we make is that the semantic similarity models are typically trained by using proper natural language sentences, such as the ones with verbs, objects, and subjects, rather than the segments of sentences, which is the case in this work. We, therefore, compute the semantic similarity in two different manners; *as-is similarity* and *action-added similarity*.

The former approach directly compares the element words extracted from the test sentence with the attributes extracted from the POM model using cosine similarity of their average word embeddings.

On the other hand, the latter approach concatenates the element words and the POM attributes with the action word before seeking semantic similarity. Concatenation gives us two generated sentences: the concatenation of action and element words and the concatenation of action and POM attribute. Given these two sentences, we calculate each sentence's average word embedding and compute the cosine similarity between the resulting vectors.

Each approach obtains the maximum similarity measure by the element word and an attribute pair. As mentioned, a POM element can have *text* and *content-description* attributes. We calculate the similarity scores for both and take the maximum of

these two. Once the similarity score for each POM element is calculated, the higher the score between the pairs, the more likely it is for the POM attribute to be the extracted UI element from the sentence.

3.3 *text2test* Test Case Execution

The goal of this section is to fully and automatically execute test cases for an application under test (AUT) using the *action*, *entity* and *parameters* that are extracted using the element extraction approach in section 3.1 and matching the extracted *entity* to the elements on the application's UI using element detection approach explained in section 3.2 and performing the *action* on the corresponding UI element.

To execute test sentences *text2test* implements a customized depth-first search (DFS) algorithm to traverse through the elements of AUT using Appium. It explores the elements based on similarity scores between extracted *entity* name and element names on the AUT UI, performing specified actions on the found element of each test sentence.

Algorithm 1 outlines the algorithm of *text2test*'s DFS exploration. The algorithm begins by launching the AUT (Line 1) and then reads the first test sentence and extracts the *action*, *entity* and *parameter* (Lines 4-5). Here it is important to note that we expect each test sentence contains a pair of *action* and *entity* words except for *rotate* action and if applicable the sentence may also have a *parameter* to apply. It then retrieves the current POM of the AUT's UI and finds the top 3 similar elements by comparing all the elements to the *entity* extracted from the test sentence (lines 7-8).To determine whether a UI component matches a test case entity, *text2test* utilizes Word2Vec Mikolov et al. (2013a), a word embedding technique, to check if the name (i.e., the displayed text) of a UI component is semantically similar to the extracted *entity* name. The model uses a score in the range of [0,1] to indicate the degree of semantic similarity between words (1 indicates an exact match). We used a relatively low score, 0.3, as the threshold. We observed that using a high threshold may cause no similar entities to be found. For example, the similarity score of "add new item" an entity from a test sentence and *fab_add_new_item* the description of the element found on the POM is 0.311 due to the addition of the "_" in between the words.

Once we read the initial sentence corresponding to the AUT's landing page and

Algorithm 1: *text2test* DFS Algorithm

Data: AUT, test sentences
Result: number of correct actions on elements equals the total number of test sentences

```
1  Launch AUT;
2  visited ← an empty set to hold visited elements;
3  path ← an empty array to hold the correct path of actions and elements;
4  sent ← sentence[0] from test sentences;
5  act, ent, param ← elementExtraction(sent);
6  POM ← currentPOM() AUT landing page;
7  top3Similar ← elementDetection(ent, POM);
8  DFSTree add top3Similar;
9  while DFSTree isn't empty do
10     if goToPath then
11         Launch AUT;
12         executePath(path);
13     if numberOfCorrectEntities = numberOfSentences then
14         return Completed;
15     else
16         node ← DFSTree.pop();
17         Add node to visited;
18         if performAct(act, node) then
19             addToPath(act, node);
20             numberOfCorrectEntities += 1
21         sent ← next sentence from test sentences;
22         act, ent, param ← elementExtraction(sent);
23         POM ← currentPOM();
24         if elementDetection(ent, POM) = 0 then
25             goToPath = True;
26             numberOfCorrectEntities -= 1
27         else
28             DFSTree add top3Similar;
29 return numberOfCorrectEntities;
```

extract the top 3 similar elements and the action to be performed, *text2test* enters a loop to perform a depth-first search iteratively. (lines 9-28). At each iteration, *text2test* selects the most relevant UI element and performs the corresponding action. Once the relevant UI element is found and the interaction with it is successful we add that UI element to the correct path of execution with its action pair and increment the number of correctly interacted entities (lines 18-20). If none of the UI elements match the entity of the test case sentence we decide that the last interacted element was an incorrect element and we set a *goToPath* flag as *True* and decrement the number of correctly interacted elements (lines 24-26). If the *goToPath* flag is set to True, that means that we need the take the next possible element from our stack and interact. So *text2test* relaunches the AUT and executes all its previous correct actions on the UI elements and continues from the next element in the stack lines(10-12). The backtracking ensures the top 3 similar elements we detected on the AUT POM are interacted from most similar to least similar until a correct element is found. This loop executes until the number of correctly interacted entities reaches the total number of sentences and this is considered as the test execution steps are completed successfully (lines 13-14) otherwise it continues to read a test sentence to find its *action*, *entity* and *parameter*, get the current AUT POM and try to interact with similar UI entities.

4. EXPERIMENTS

This chapter describes the data sets used in the experiments and the system configuration. Then we will present the results of our element extraction NER approach and discuss the effects of different layers and features used. On top of this, we will share the results of our proposed element detection approaches. Moreover, we will present the results of the *text2test* complete approach for test case generation. Finally, we will be comparing the entity recognition capabilities of our NER approach with Zhao et al. (2019)'s work since it is the closest approach that executes test cases from natural language sentence descriptions to ours.

4.1 Operational Framework

We perform the NER model training and evaluation using Google Colab. The machine allocated to us from Google Colab machine has a 1 socket connected to two 2 cores of an Intel(R) Xeon(R) CPU running at 2.20 GHz with 2 threads per core. For both applications on Google Colab, we are also utilizing an NVIDIA Tesla T4 GPU. The Colab notebook uses 18.04.6 LTS (Bionic Beaver) and has a Python version of 3.8.16. We use PyTorch as our machine learning framework for both tasks with the following Python package versions transformers=4.12.3 allennlp=2.8.0 PyTorch-lightning=1.3.8 and all the dependencies that come with them. Also, for both tasks, we use the BERT base cased model and OpenNLP for POS tagging in NER.

The element detection process and the *text2test* test case execution are performed on a MAC M1 Pro with Apple Silicon, with 16 GB of RAM running the macOS operating system. In the element detection process, we calculate semantic similarities in two folds, using the fine-tuned NER model we trained and the Google Universal Sentence Encoder. In the *text2test* test case execution process we use the appium-python-client=1.0.2 Python package to automate the execution of test cases to test

various mobile applications.

4.2 Evaluation Framework

Our experimentation aims to evaluate the following:

- the success of extracting the *action*, the *element* and the *parameter* words for a given test sentence.
- rank the POM attributes on a given screen as the most similar to the extracted UI entity .
- how well can we combine element extraction and element detection in order to execute test cases from given sentences and POM pairs?
- comparison of the effectiveness of our element extraction approach compared to the ReCDroid framework.

As explained in Section 3.1 and Section 3.2, we can describe our approach with two steps following each other. We carried out separate evaluations for these steps. Initially, for the information extraction step, we measured the performance of our NER model depending on how accurately our model detects the *action*, *element*, and *parameter* words for a given test sentence. We obtained the ground truth for element extraction and approaches by human annotation. The annotation identified and labelled the *action*, *element*, and *parameter* words. For the element identification step, we measured the performance depending on the calculated similarity rank of the pre-determined correct element for detected element words (we expect the correct element to have the highest similarity score).

To evaluate the success of the element extraction step (Section 4.4.1), we use *precision*, *recall*, and *F-measure* metrics, which are well-known metrics frequently used for evaluating similar machine learning models:

$$Precision(P) = \frac{TP}{TP + FP}$$

$$Recall(R) = \frac{TP}{TP + FN}$$

$$Fscore = \frac{Precision \times Recall}{Precision + Recall}$$

More specifically, the precision of a prediction made for label l is computed as the ratio of the correctly labelled words to all the words labelled with l by the NER model. In contrast, the recall is computed as the ratio of the correctly labelled words to all the words that needed to be labelled with l. Moreover, the F-measure is computed, giving equal importance to both precision and recall. Note that all of these metrics assume a value between 0 and 1 inclusive. The higher the value, the better the proposed approach is. Furthermore, we report the precision, recall, and F-measures for two types of element extraction calculations.

The first approach, complete matching, accepts a true positive on an element if the predicted and ground truth elements match precisely. The second approach, partial matching, determines a true positive if the predicted entity span from the model overlaps with the entity span of the ground truth entity in the sentence. To explain this further, when we are using partial matching, we consider a predicted entity to be a true positive if there is any overlap between the predicted entity span (the range of words identified by the model as the entity) and the ground truth entity span (the range of words that actually represent the entity in the sentence). To illustrate this we can consider the ground truth entity *set as default browser*, which we will see as a UI item, and if the predicted entity is *default browser*. In this case, although the ground truth and the predicted entity are not an exact match, there is an overlap in *default browser* in both spans. Therefore, using a partial matching approach we count this example as a true positive.

To evaluate the success of the element detection step (Section 4.4.2), we compute the accuracy of the predictions. Note that the output of this step is a ranked list of UI elements ordered by their semantic similarity scores. To compute the accuracy, we check whether the actual UI element addressed by the test sentence is in the top k of the reported list. More specifically, given k (in our case, $1 \leq k \leq 3$), the accuracy of the predictions is computed as the percentage of the test sentences, the UI element of which appears in top k predictions.

$$Nk = \frac{(\#\, occurrences\, of\, the\, precise\, element\, in\, top\, x)}{(Total\, number\, of\, elements\, in\, test\, set)} \tag{4.1}$$

The formula 4.1 explains the calculation of an N score of a NER model-similarity calculation pair used in the element detection step.

Also, we execute test sentences on an AUT with our test reproduction framework *text2test*. We evaluate the success of the test case as how many successful test sentences are executed. This means that from each test sentence in a test case, first, we identify the correct action, element and parameter, secondly, we identify the correct element on the corresponding screen and lastly interact with that element successfully. We calculate the individual success ratio for each application as the percentage of successfully executed sentences out of the total number of sentences for a particular application. Finally, give a total success ratio as the number of completely executed tests for an application to the total number of applications.

Finally, in the last part of our evaluations, during the evaluation against the ReC-Droid framework, we are using the same metrics that we are using for the element extraction step. Then, we compare how well both approaches perform against each other.

4.3 Dataset and Subject Applications

4.3.1 Dataset used in Element Extraction and Element Detection

We used a well-known open-source instant messaging application *Signal*, a prevalent free web browser backed by Mozilla *Firefox* and cross-platform media player software *VLC* in these experiments as our subject applications. Getting enough sample sentences that contain action and elements was the main challenge to procure our dataset. Towards this goal, we choose these applications because they are all open-source with considerable amounts of reported issues and we can fetch user-reported issues using *GitHub Issues API*. A well-written issue report contains reproduction steps to describe the problem at hand and reproduction steps are the point of interest as they contain similar sentences to test cases. All these 3 applications mentioned above contain ample amounts of reproduction steps that we can collect using an automated script. The sentences in these reproduction steps are similar to test case sentences for UI verification and they include sufficient elements and actions in many different screens, which helped us obtain comprehensive evaluations.

Bert-Finetuning: In the first step of our NER approach, we pre-train BERT base

cased model with sentences from our application domain. We call this step BERT-finetuning to the application domain. We used all 400000 issue sentences from GitHub without any modification to be used in a masked language modelling (MLM) task finetuning the base model. This training allows the base model to generalize better on the domain at hand while retaining its general language understanding.

Sentence Classification to Gather Applicable Test Sentences: Moving forward from all the collected issue sentences from GitHub we need to create data sets. Initially, we started manually combing through all the sentences in the reproduction steps. As mentioned before the sentences in the reproduction steps of an issue contains similar test sentences that we are trying to execute. However manually creating data sets was time-consuming, error-prone and tedious. Thus we opt to manually analyze 1821 issue reports and then use them to train a text classification model. Then use this text classification model to automatically create the remaining data sets. We classified the reproduction steps into five classes in Table 4.1. The first three classes *Green*, *Yellow*, *Orange* are sentences that we can use to train, validate and evaluate our NER model.

In particular, utilizing our text classifier we determined the sentences that indicated action on a UI element. The text classifier analyzed about 4000 issue reports for Signal; about 6761 issue reports for Firefox; and about 2000 for VLC and filtered the issue reproduction steps. We performed this step to identify sentences that contain *action* and *entity* elements in an automated manner. Please note that the sentences that do not contain *action* and *entity* elements do not contribute to the purpose of this approach. We are trying to detect *action* and *entity* elements from a given test sentence and match the *element* found in that sentence to a UI element of the application. Thus we removed sentences containing prerequisites or observations that do not contain *action* and *entity* pairs and also eliminated the code samples and crash outputs. This classification process makes sure we are only using data with *action* and *entity* pairs and it does not affect the evaluation process and the generality of the results.

Class	Information	Distribution
Green	simple sentences that include a verb from the given action list and a noun or a noun phrase from the given elements list that the action will be use used on	13.3%
Yellow	complex sentences. some part of the sentence contains a verb from the action list and a noun or a noun phrase from the elements list that the action will be use used on	11.0%
Orange	complex sentence. does not necessarily contain an action verb or an element noun. With some sentence modification, it can be turned into a green or a yellow sentence	35.6%
Red	a sentence that contains a prerequisite or observation or additional information and does not contain an action verb or element pair	14.7%
Black	code or crash outputs	25.5%

Table 4.1 Classes used for text classification and their distributions

Element extraction: The sentence classification model identified a total of 4581 sentences, 689 from Signal, 2884 from Firefox and 1008 from VLC to be used for the NER model for element detection. Out of which 3918 (85%) sentences (2101 Firefox sentences, 583 Signal sentences, 852 VLC sdentences) were used as the training set, and the remaining sentences (15%) were used as the test set. We labelled the dataset using IOB (Inside, Outside, Beginning) format first presented by Ramshaw and Marcus. Ramshaw & Marcus (1995) These sentences that we collected had *action* and *entity* labels, how ever did not contain enough samples of *parameter* labels. We are using issue report sentences, where usually, the failure happens if there is a faulty functionality with the UI element. So the problem can be reproduced before the user can input a parameter. That is why we are not able to collect enough *parameter* labels. So we decided to generate sentences that contain *parameter* labels, and towards this goal, we introduced 404 new sentences to the Firefox application and 50 new sentences to the VLC application similar to the *parameter* examples in the collected issue reports. We did not introduce new sentences to the Signal application, we did not generate sentences for Signal, because in the test sentences that we collected from the issue reports there were no *parameter* examples that can be used to generate further sentences. Table 4.2 shows the number of total and unique *action*, *entity* and *parameter* elements in our data, which shows a high number of unique elements for each class which is important to help the NER model to disambiguate entities and reduce confusion.

Number of Label Types	Firefox	Signal	VLC
# of Action Labels	3964	679	1003
# of Unique Action Labels	545	132	336
# of Element Labels	3703	675	974
# of Unique Element Labels	2247	435	785
# of Parameter Labels	412	-	78
# of Unique Parameter Labels	373	-	70

Table 4.2 Number of total elements and number of unique elements

Element detection: In this part of the experimentation we picked 200 sentences from the data sets we created. The distribution of these sentences is 40 sentences from Signal, 80 from Firefox and 80 from VLC. Then we manually collected the POM models of the corresponding application screens.

4.3.2 Dataset used in *text2test* Test Case Execution

For the complete approach *text2test*, we used 20 different applications mentioned in the ReCDroid paper that we were able to launch in our system configuration described in Section 4.1. For these 20 applications, the authors provided 24 issue reports that have reproductions steps. We use these issue reports as test cases for each application for the purpose of *text2test*. Since we aim to detect an element and interact with it given a sentence and a POM pair, we modified some test sentences. The reason for this modification is due to some technical reasons our system not being able to run the exact version of the mentioned application and we needed a newer version of the subject application, where the item name is modified. Also, to get the application to the correct screen mentioned in the issue report, we added some preliminary sentences to these issue reports (usually 1 or 2 sentences at the beginning). In particular, our approach takes as input a screen and the instructions to be executed on them. However, issue reports almost always do not start from the landing page of an application. So these preliminary sentences make sure *text2test* can reach the screen related to the beginning of the issue report. Table 4.3 reports the applications we are using. For only 3 applications, we had to modify the item that was mentioned in the ReCDroid issue report.

Application Name	**# of Test Sentences**	**Application Description**	**Contains Exact Item**
ACV	6	Comic Book Reader	Modified
anymemo	7	Flash Card Study App	YES
anymemo2	9	Flash Card Study App	YES
asciicam	9	ASCII Webcam	YES
birthdroid	7	Birthday Reminder	YES
car_report	14	Car Cost Tracker	YES
dagger	1	Dagger Android Extension	YES
fastadapter	1	RecyclerView Creator	YES
fastadapter2	8	RecyclerView Creator	YES
flashCards	7	Flash Card Study App	Modified
librenews	5	News Notification App	YES
librenews2	6	News Notification App	YES
librenews3	5	News Notification App	YES
markor	8	Text Editor	NO
memento	6	Organizer App	YES
news	3	News Reader	YES
odb	3	Onboard Diagnostic Reader	YES
openhab	6	Home Automation App	YES
openSudoku	11	Sudoku Game	YES
qksms1	8	Messaging App	YES
qksms2	4	Messaging App	YES
screenRecord	8	Screen Recorder App	Modified
shuttle	6	Music Player	YES
tagmo	3	NFC tag manager	YES
transistor	3	Radio App	YES

Table 4.3 Applications used in testing *text2test*, the number of sentences in each test case and information from the reported ReCDroid issue report.

An example test case for the ACV application is as follows:

- Sentence 1: click continue - act: click - ent: continue
- Sentence 2: click ok - act: click - ent: ok
- Sentence 3: click menu - act: click - ent: menu
- Sentence 4: choose "open" - act: choose - ent: "open"

- Sentence 5: go to directories like Android - act: go - ent: directories, Android
- Sentence 6: long-press a folder, like "data" - act: long-press - ent: folder, data

Sentences 1 and 2 are preliminary sentences added to the test case. These sentences are used to get the application to the initial state in the issue report. Sentences 3, 4 and 6 are sentences that we used as-is. Sentence 5 is an example where we modified the sentence. The original sentence was *go to directories like /tmp*, however as mentioned before the application version that ran on our system did not have a */tmp* folder, so we replaced the folder with an existing one. Also, sentence 5 is a good example where the NER approach guessed two entities in a sentence. It guessed *directories* and *Android* as entities in this sentence, where *Android* is the correct one. We designed an algorithm to handle these types of situations.

4.3.3 ReCDroid Comparison:

Finally, we are comparing our element extraction approach 3.1 to the element extraction of ReCDroid. We are doing this comparison to understand the effectiveness of our approach on the RecDroid test sentences. Also, we are trying to see the effectiveness of the dependency parser used by ReCDroid to identify elements. The comparison between our *text2test* element extraction approach and ReCDroid comprises 2 steps. In the first step, we used the test sentences from the Signal, Firefox and VLC, ran them through the ReCDroid framework with the help of a wrapper developed by us and collected the extracted *action* and *entity* words. In the second step, we performed element extraction with our trained NER approach on the 94 sentences that ReCDroid is using as test sentences from crash reports. We discuss the results of these comparisons in Section 4.4.4

4.4 Experimental Results

This section starts with discussing experiments conducted to improve element extraction in user interface (UI) applications using BERT fine-tuning and neural network architectures. Initially, we compare the BERT-base cased model from Huggingface with our fine-tuned model *UI-BERT*. Then we perform an ablation study

on NER architecture and analyze the effects of adding new features to the embedding layer. The results show that fine-tuning BERT models increases the success of element extraction, but adding POS and char embeddings does not improve entity recognition, which we will discuss in detail later in the section. In this section, we also compare our NER approach with ReCDroid and analyze the entity detection and action detection scores. As a final experiment for element extraction, we discuss adding a new element, *parameter*, to the training data and evaluate the NER model's performance. After extracting element words from test sentences, we share the results of element detection between extracted words and UI element attributes from the screen's POM model using two similarity calculation methods: "as-is" and "action-added." We experiment with Universal Sentence Encoder and the *UI-BERT* models and report our detection results. Finally, we will share the experimental results of *text2test* and discuss this approach's success on several Android Applications.

4.4.1 Element Extraction

In our experiments, the first item we look into is the effects of BERT fine-tuning. Specifically, we are comparing the predictions made using the BERT-base cased model from Huggingface with the fine-tuned model *UI-BERT*. As explained in Section 3.1.1, training the base model with unlabelled issue report sentences with an MLM objective will improve the adaptation of the NER approach to the domain, and it will have a better understanding of the word features and the semantic relationships. At this point, we are only training a model with BERT and an ArgMax layer to make predictions; we are not utilizing the Bi-LSTM-CRF architecture to see the effects of *UI-BERT* without any other layers in the NER approach.

		BERT-base-cased			*UI-BERT*		
		precision	**recall**	**F-score**	**precision**	**recall**	**F-score**
Signal	**ACT**	0.9608	0.9899	0.9751	0.9608	0.9899	0.9751
	ENT	0.6186	0.6697	0.6432	0.6911	0.7798	**0.7328**
Firefox	**ACT**	0.865	0.9423	0.9020	0.9288	0.9721	**0.9500**
	ENT	0.6299	0.7219	0.6728	0.6899	0.7415	**0.7148**
VLC	**ACT**	0.8703	0.9321	0.8917	0.9227	0.9009	**0.9117**
	ENT	0.6088	0.7136	0.6597	0.7336	0.6709	**0.7009**

Table 4.4 BERT-base-cased vs *UI-BERT* results

Overall, Table 4.4 shows that fine-tuning of BERT models *UI-BERT* increases the success of the element extraction task. We are seeing a profound increase in the entity recognition F-scores for Signal, Firefox and VLC applications which have a 9, 4 and 4 point increase, respectively. When we compare the action recognition F-scores, we see an increase in Firefox and VLC. Our investigation shows that the Signal application's training sentences are usually shorter and contain only one verb. The average number of words in the Signal application is 4 words per sentence, compared to the 7 words per sentence for Firefox and 8 words per sentence for VLC. Since Signal application sentences contain only one action, it is easier for both models to make correct predictions for the action labels.

Contrary to this, training sentences of Firefox and VLC have longer sentences that sometimes have more than one verb and applying fine-tuning helps to increase action recognition F-score by 5 and 2 points. We can further investigate the effect of domain adaptation in Table 4.5. Sentences 1, 2, and 3, the base BERT model cannot make an entity prediction on the sentence or misses an entity completely since these entities are just ordinary words; however, after fine-tuning *UI-BERT* can extract an entity or both entities from the sentences. Sentences 4, 5, 6, and 7 are examples of entities containing more than one word. The base BERT model cannot label every word in an entity, because of the data it is originally trained on it cannot correctly decide on domain-specific multi-word entities. Fine-tuning of *UI-BERT* allows it to label these multi-word entities correctly since it has seen not identical but a similar sequence of words in its fine-tuning where we are adapting the base-BERT to *UI-BERT*. Finally, sentence 8 is an example of base BERT model false prediction on the work *tab*, a frequent word in Firefox and usually tagged as an entity. However, in this case, it should not be an entity since it just gives extra information on the needed action.

ID	Sentence	BERT-base-cased		*UI-BERT*	
		act	**ent**	**act**	**ent**
1	**Tap** on *Saved logins*, unlock it if biometrics or pin is set.	tap	-	tap	saved logins
2	**Enable** *files permissions.*	enable	-	enable	files permissions
3	**Turn on** *Wifi.* **Press** *Try again* in the system notification.	turn, press	wifi	turn on, press	wifi, try again
4	**Select** the option to *Open links in apps.*	select	open link	select	open links in apps
5	**Tap** on *Show more* on any social media icon.	tap	show	tap	show more
6	**Disable** the *bookmarks sync* in Sync settings. Sync, so the setting takes effect on all 3 devices.	disable, Synch	bookmarks	disable	bookmarks sync
7	**Tap** again on the *3dot menu* and **select** *Add to Home screen.*	tap, select	3dot, home	tap, select	add to home screen
8	**Hit** *undo* to restore the tab.	hit	undo, tab	hit	undo

Table 4.5 Action and Entity predictions by Bert-base-cased model and *UI-BERT* given a test sentence. **bold** words represent the ground truth action in the sentence, while *italic* word represents the ground truth entity.

Moving forward, using domain adapted *UI-BERT* and will be performing an ablation test. As discussed in Section 3.1 introducing new features in NER systems increases the effectiveness. This test aims to see the effects of the downstream bi-lstm and CRF layer and, secondly, the effects of adding new POS and char features to our embedding. Table 4.6, Table 4.7 and Table 4.8 summarize the experimental ablation results. The impact of the bi-lstm and CRF layer addition to the BERT

model as a downstream task is evident; long-short term memory coming from the bi-lstm layer combined with the CRF layer's decision-making capabilities increases the entity recognition in the Signal, Firefox and VLC applications with 8, 2.5 and 2.6 points respectively. This increase in F-scores for Signal application is again due to Signal training and more concise test sentences. However, even with longer, more complex sentences and multi-word entities, which Firefox and VLC have, we are seeing a significant increase in the element detection capability of the model. Here we would like to point out that the scores for action recognition are similar in all steps ablation test. The similar action scores are because mobile applications have a limited number of actions repeated in all sentences, giving the model a solid performance. We designed our approach hypothesizing that adding new features to our embedding layer would improve NER tasks. We expected to see this because, with a relatively small set of sentences from the application, every new information added would be a benefit.

Nevertheless, in our case, we did not improve the element detection results by adding POS and char CNN features to the embedding layer. Using POS and char CNN embeddings did not positively affect entity recognition compared to using BERT embeddings. Since the sentences extracted from issue reports are irregular sentences, which usually do not have both a subject and a complete predicate, they are used in writing or speech as complete sentences that stand on their own. POS tagging these irregular sentences with *OpenNLP* does not yield precise and dependable information. Discrepancies between the POS tags assigned by the OpenNLP and the expected POS tags in the irregular sentences extracted from issue reports mean that certain words may not accurately reflect their grammatical role or function in the sentence. Some verbs in imperative sentences can be interpreted as nouns. These differences and inaccuracies in POS tagging hurt the performance of the entity recognition system.

On the other hand, using charCNN alone did not improve the score either; our investigation shows that it yielded a higher number of false positive predictions while making fewer false negative predictions. Also, while (Syed & Chung, 2021) shows that using POS and char CNN together causes a synergy and increases NER performance, we did not see the same effect in our testing. No effect in testing means that POS and char embeddings did not add useful information in our case. We believe that the syntactic and morphological information they provided was only sometimes relevant or informative for entity recognition, and the information provided by BERT embeddings takes precedence in our case.

		Signal *UI-BERT*		
		precision	**recall**	**F-score**
argmax	ACT	0.9608	0.9899	0.9751
	ENT	0.6911	0.7798	0.7328
bert_CRF	ACT	0.9800	0.9703	0.9751
	ENT	0.8209	0.8088	**0.8148**
bert_CRF_ pos	ACT	0.9800	0.9703	0.9751
	ENT	0.7951	0.8083	0.8017
bert_CRF_ CNN	ACT	0.9510	0.9604	0.9557
	ENT	0.7946	0.7417	0.7672
bert_CRF_ pos_charCNN	ACT	0.9423	0.9703	0.9561
	ENT	0.8083	0.8083	0.8083

Table 4.6 Signal app. experimental results for NER approach utilizing *UI-BERT* model with extended features

		Firefox *UI-BERT*		
		precision	**recall**	**F-score**
argmax	ACT	0.9288	0.9721	0.9500
	ENT	0.6899	0.7415	0.7148
bert_CRF	ACT	0.9515	0.9444	0.9480
	ENT	0.7523	0.7257	**0.7387**
bert_CRF_ pos	ACT	0.9241	0.9463	0.9350
	ENT	0.7206	0.7257	0.7231
bert_CRF_ CNN	ACT	0.9416	0.9556	0.9485
	ENT	0.7032	0.7381	0.7202
bert_CRF_ pos_charCNN	ACT	0.9201	0.9593	0.9393
	ENT	0.7172	0.7451	0.7309

Table 4.7 Firefox app. experimental results for NER approach utilizing *UI-BERT* model with extended features

		VLC *UI-BERT*		
		precision	**recall**	**F-score**
argmax	ACT	0.9227	0.9009	0.9117
	ENT	0.7336	0.6709	0.7009
bert_CRF	ACT	0.894	0.9151	0.9044
	ENT	0.7794	0.6795	**0.7260**
bert_CRF_ pos	ACT	0.9163	0.9292	0.9227
	ENT	0.7143	0.6624	0.6874
bert_CRF_ CNN	ACT	0.9113	0.9134	0.9227
	ENT	0.7243	0.6811	0.7072
bert_CRF_ pos_charCNN	ACT	0.9116	0.9245	0.918
	ENT	0.7220	0.6880	0.7046

Table 4.8 VLC app. experimental results for NER approach utilizing *UI-BERT* model with extended features

As an end product, we would like our element extraction NER approach to generalize to all types of mobile applications. So we believe that if we train a NER system using training data from multiple applications, the success of extraction should increase. In our next test, toward this goal, we trained another NER model with a combination of test sentences from Signal, Firefox and VLC. We achieved this by combining each application's training, validation, and test datasets and trained another NER model with *UI-BERT*. The results of Table 4.9 show that combining different sentences positively affects element extraction. Although the combined model has a lower score than the Signal model, we have a more significant improvement than our Firefox and VLC applications predictions.

Further examination of the predictions shows that the improvements are due to two factors. First, combining both training sets helps improve multi-word Firefox and VLC entities' prediction. Secondly, in our sentences, we have words such as *option*, *menu* and *button* at the end of entities, which we did not mark as a part of an entity. For example, in the sentence *Open bookmarks menu* the expected entity is *bookmark*; however, we realized that both only Signal and only Firefox models sometimes predict the entity as *bookmarks menu* but using all training sets improves on this behaviour.

		precision	recall	F-score
Signal	**ACT**	0.9800	0.9703	0.9751
	ENT	0.8209	0.8088	0.8148
Firefox	**ACT**	0.9515	0.9444	0.9480
	ENT	0.7523	0.7257	0.7387
VLC	**ACT**	0.894	0.9151	0.9044
	ENT	0.7794	0.6795	0.726
Signal & Firefox & VLC	**ACT**	0.9387	0.9594	0.9489
	ENT	0.7308	0.7604	0.7453

Table 4.9 BERT bi-LSTM CRF Model results with Signal, Firefox and VLC application training and test sets individually versus the combination of these training and testing sets. The entity extraction results are exact-match results.

Investigation of BERT bi-LSTM CRF trained on Signal & Firefox & VLC sentences showed that the false positive results usually occurred on multi-word entities, either the model misses predicting one of the words or predicts extra words such as *option, menu, button* that usually occur at the end of the element.

Our final goal is to match elements extracted from a sentence with the elements we get from the application POM. Towards this goal, we decided that if we can extract some of the words from multi-word elements, then this information is still helpful in detecting elements from the POM. This thought led us also to calculate partial element extraction results given in Table 4.10. Here the true positive is calculated if the span of the ground truth entity in the sentence intersects with the span of the entity prediction from the model. For all models, significantly higher scores indicate that we are at least finding one word of multi-word elements, which was the expected result.

		precision	recall	F-score
Signal	**ACT**	0.9515	0.9800	0.9655
	ENT	0.8655	0.8655	0.8655
Firefox	**ACT**	0.9387	0.9738	0.9560
	ENT	0.8761	0.9089	0.8922
VLC	**ACT**	0.9260	0.9431	0.9343
	ENT	0.9450	0.8112	0.8730
Signal & Firefox & VLC	**ACT**	0.9544	0.9733	0.9638
	ENT	0.8870	0.9200	0.9032

Table 4.10 BERT bi-LSTM CRF Model results for the same approach in Table 4.9 but partial action and element detection calculated as true positive.

Finally, we added a new element *parameter* to our training data on the element extraction testing. The *parameter* elements contain values that the *entity* elements can hold. In this test, we combined the Signal & Firefox & VLC concatenated training, validation, and test data, and we divided the new sentences with *parameter* elements (the 454 sentences mentioned in Section 4.3 with a 70/10/20 ratio to the data respectively). The results of the test sentences containing *parameters* are in the Table. 4.11. Overall, we can conclude that the new NER model performs well on the new *parameter* elements while retaining its element extraction capabilities for the *action* and *entity* elements.

		precision	**recall**	**F-score**
exact matching	**ACT**	0.9612	0.9637	0.9625
	ENT	0.7477	0.7606	0.7541
	PARAM	0.9390	0.8652	0.9006
partial matching	**ACT**	0.9740	0.9753	0.9747
	ENT	0.9151	0.9272	0.9212
	PARAM	0.9753	0.8977	0.9349

Table 4.11 'Signal & Firefox & VLC" *UI-BERT* _bi-LSTM_CRF model predictions with the new *parameter element*

4.4.2 Element Detection

After the NER model tags the words, the next step is to use the words tagged with the element labels (i.e., the element words) to identify the actual UI element on the screen. In the identification step, we compute the semantic similarities between the element words extracted from the test sentence and the attributes extracted from the POM structure of the given screen. We opted to evaluate the proposed approach on the Android platform for this work. As such, we obtain the POM models of the screens by using Android Debugging Bridge Developers (2021). However, the proposed approach readily applies to other POM models, which can be obtained from different mobile and Web platforms.

To compute the semantic similarities, we, in particular, utilize the *text* and *content-description* attributes (if present) of the elements in the POM model. Specifically, we compute the semantic similarity between the element words from test sentences and each attribute present in the POM using their word embeddings extracted from the NER model.

One observation we make is that the semantic similarity models are typically trained by using proper natural language sentences, such as the ones with verbs, objects, and subjects, rather than the segments of sentences, which is the case in this work. We, therefore, compute the semantic similarity in two different manners; *as-is similarity* and *action-added similarity*.

The former approach directly compares the element words extracted from the test sentence with the attributes extracted from the POM model using cosine similarity of their average word embeddings. On the other hand, the latter approach concatenates the element words and the POM attributes with the action word before seeking semantic similarity. Concatenation gives us two generated sentences: the concatenation of action and element words and the concatenation of action and POM attribute. Given these two sentences, we calculate each sentence's average word embedding and compute the cosine similarity between the resulting vectors.

Each approach obtains the maximum similarity measure by the element word and an attribute pair. As mentioned, a POM element can have *text* and *content-description* attributes. We calculate the similarity scores for both and get the similarity scores and calculate a top 3 similarity described in 4.2. Once the similarity score for each POM element is calculated, the higher the score between the pairs, the more likely it is for the POM attribute to be the extracted UI element from the sentence.

Table 4.13 indicates the element detection results for three similarity calculation approaches. We calculated the average number of elements in the analyzed screens as ten elements.

		as-is Similarity	action-added similarity
BERT	N1	65.83	60.83
	N2	81.67	77.50
	N3	84.17	88.33
Universal Sentence Encoder Transformer	N1	72.50	73.67
	N2	79.17	82.33
	N3	85.00	87.83
Universal Sentence Encoder DAN	N1	74.17	74.17
	N2	89.17	88.33
	N3	91.67	91.67

Table 4.12 Element Detection Results

		accuracy
UI-BERT	N1	65.83
	N2	81.67
	N3	84.17
Universal Sentence Encoder Transformer	N1	72.50
	N2	79.17
	N3	85.00
Universal Sentence Encoder DAN	N1	74.17
	N2	89.17
	N3	91.67

Table 4.13 Element Detection Results

We calculated the scores of all NER model-similarity calculation approach pairs with a rank-based order depending on whether the relevant element occurred in the most similar 3 UI elements from the screen after the comparison. This calculation depends on the occurrence of the distinct element in the top 3, top 2, or top 1 of the compared elements. These top 3, top 2, and top 1 scores are N1, N2, and N3 in Table 4.13.

In the BERT model similarity calculation, we utilized the BERT_bi_LSTM_CRF model, which performs best in element extraction. We can note that all our similarity calculation methods provide acceptable results checking the N3 success rates; they are performing over 84% N3 element detection score.

It is important to note, that both universal sentence encoder models (trained with transformers and DAN) perform better than our BERT model. We expected the BERT model with fine-tuned issue report sentences would provide better features via its word embeddings and thus provide better similarity. However, the results show that Universal Sentence Encode will be the better choice, we believe that this is due to the low domain knowledge requirements of similarity calculation.

Both the transformer and the DAN models of Universal Encoder got over 90% N3 success. While the transformer model is expected to create better results than the DAN model, it could not make a difference since our sentences were not that long to understand a structure. The average number of words in test sentences is 5.3. Therefore, the transformer model could not show better results.

Appending the detected action word to the entity words did not create a valuable change. While as-is element detection obtained 91.67% average N3 success, action-added element detection also had the same score. Moreover, adding the detected

action word to information of all elements from the DOM structure creates a fake similarity because of the appended common words.

Moreover, we observed that the selected user's readable information might not be unique on an application screen. For example, in the Signal application while on a screen of multiple messages, all the message elements had the content description *media message*. It does not contain any message-specific information on the DOM structure. Therefore, it was impossible to differentiate which message was mentioned in the test sentences.

4.4.3 *text2test*- Test Case Execution

In this section, we combine information extraction and element detection to execute the test cases from given sentences and POM pairs. Table 4.14 summarizes the results of applying *text2test* on the 25 test cases out of 51 issue reports identified by the authors of the ReCDroid paper. We did not include the remaining issue reports due to the limitations of our system configuration.

Application Name	**Number of sentences**	**Successfully executed sentence count**	**Successful Execution**	**Sentence Success Ratio**
ACV	6	6	YES	100.0
anymemo	7	7	YES	100.0
anymemo2	9	9	YES	100.0
asciicam	9	2	NO	22.2
birthdroid	7	7	YES	100.0
car_report	14	14	YES	100.0
dagger	1	0	NO	0.0
fastadapter	1	1	YES	100.0
fastadapter2	8	8	YES	100.0
flashCards	7	7	YES	100.0
librenews	5	5	YES	100.0
librenews2	6	6	YES	100.0
librenews3	5	3	NO	60.0
markor	8	5	NO	62.5
memento	6	5	NO	83.3
news	3	3	YES	100.0
odb	3	3	YES	100.0
openhab	6	2	NO	33.3
openSudoku	11	11	YES	100.0
qksms1	8	8	YES	100.0
qksms2	4	4	YES	100.0
screenRecord	8	8	YES	100.0
shuttle	6	6	YES	100.0
tagmo	3	3	YES	100.0
transistor	3	3	YES	100.0

Table 4.14 *text2test* Test Case Reproduction Results

Table 4.14 presents a summary of the experiment results. Each row corresponds to a specific application name, along with the following information:

- **Number of sentences**: This indicates the total number of sentences extracted from the issue report for a particular application.
- **Successfully executed sentence count**: This denotes the number of sentences that were successfully executed as test cases using the *text2test* system.

- **Successful Execution**: This field indicates whether the execution of all test cases for a given application was successful. It is marked as "YES" if all test sentences were executed without errors and "NO" if any of the test sentences failed to execute successfully.
- **Sentence Success Ratio**: This metric represents the percentage of sentences that were successfully executed out of the total number of sentences for a particular application.

Looking at the results, we observe that the *text2test* system achieved a 76% success rate in executing all test cases of the applications. Applications like ACV, anymemo, anymemo2, birthdroid, car_report, fastadapter, fastadapter2, flashCards, librenews, news, odb, openSudoku, qksms, screenRecord, tagmo, and transistor achieved a 100% sentence success ratio. This indicates that all the extracted sentences were successfully executed as test cases without any errors for these applications.

However, some applications showed lower success rates. For example, asciicam had a sentence success ratio of 22.2%, indicating that only a small fraction of the sentences could be executed successfully. Similarly, applications like librenews3, markor, memento, and openhab also had lower sentence success ratios, indicating that a significant number of test cases could not be executed successfully. Here are specific reasons for the lower success rates in executing test cases for certain applications:

asciicam: The item in this application does not have a text or description field. As a result, using the resource ID alone was not sufficient to find the correct flow and execute the test cases accurately.

dagger: The application crashes before executing the sentence, which prevented the successful execution of test cases.

librenews3: The entity mentioned in the sentence could not be matched with any element on the screen. This led to difficulties in identifying and interacting with the correct elements during test case execution.

markor: The element's resource ID, content description, and text fields did not contain any information. This lack of identifying information made it challenging to accurately execute the corresponding test cases.

memento: The date-set field in this application did not have any associated text, content description, or resource ID that could be used to interact with it during test case execution. This limitation hindered the successful execution of test cases.

openhab: The application did not work properly, and there was no page available

after the initial load. This issue prevented the successful execution of test cases in this particular application.

Out of the six applications that we could not execute test cases successfully, two of them were due to the applications not working as expectedly. Three of the applications did not have any descriptive information on the element that can be used in the element detection step. And lastly, there was only one application where the NER approach failed and predicted an incorrect element. It is important to address these issues to improve the overall effectiveness and reliability of the *text2test* system across a wider range of applications.

These results highlight the varying effectiveness of the *text2test* system across different applications. While it demonstrated high performance for most applications, there were cases where it faced challenges in accurately executing test cases. Further analysis and improvements could be explored to enhance the system's performance and address the limitations observed in specific applications.

4.4.4 *text2test* Element Extraction Comparison to ReCDroid

In this sub-section we are comparing our *text2test* element extraction approach with Zhao et al. (2022,1). ReCDroid approach uses SpaCy (Honnibal & Montani, 2017) dependency parser to identify grammatical patterns that describe user actions, target GUI elements from issue reports, and automatically reproduce crashes for Android apps. Once they identify the user action and target GUI element, they utilize word embeddings to determine semantic similarity with the POM element. In the scope of element extraction, we performed two tests with ReCDroid. The first test determines how well our Signal & Firefox & VLC BERT_bi-LSTM_CRF model can detect the elements from the ReCDroid test sentences. The second test is to feed the Signal & Firefox & VLC test sentences to the ReCDroid dependency parser, analyze the patterns it finds, and extract the action and the entities.

		precision	**recall**	**F-score**
element extraction	**ACT**	0.8022	0.9125	0.8538
	ENT	0.6705	0.6413	0.6556
partial extraction	**ACT**	0.9111	0.9425	0.9266
	ENT	0.8687	0.87755	0.8730

Table 4.15 Signal & Firefox & VLC *UI-BERT* _bi-LSTM_CRF model prediction results on ReCDroid test sentences

The 4.15 Table shows the element and action detection from the 96 sentences ReCDroid testing includes. These 96 sentences are from crash reports of 30 different applications. The action label F-score shows us that independent of applications since the set of possible actions on the mobile phone UI is limited, we can achieve an 85% score. The promising aspect of this test is the entity detection F-score of 65.6%. The NER model is not trained with entities on the 30 different applications, but we believe that it can generalize to other applications. When we check the partial entity matching score, it is at 87%. The 87% F-score means the model, even though unable to detect the exact element, is aware that some part of it must be an entity.

		precision	**recall**	**F-score**
element extraction	**ACT**	0.8293	0.8097	0.8194
	ENT	0.4231	0.4195	0.4213
partial extraction	**ACT**	0.8290	0.8095	0.8191
	ENT	0.500	0.4845	0.4921

Table 4.16 ReCDroid NLP model on Firefox and Signal test sentences

The results of ReCDroid dependency parser detection actions and entities on Signal, Firefox and VLC sentences are in Table 4.16. The action detection of the dependency parser user by ReCDroid is commendable. It can detect single-word actions like a tap and click select with high accuracy and is also good at understanding multi-word actions like a double tap and long click. However, it cannot detect actions like a swipe or a scroll, and this is due to the authors not providing grammatical patterns including these actions. The problem with the dependency parser approach becomes evident when we look at the entity recognition F-score of 42%. Even if we apply the partial entity recognition approach, it achieves 49%. We believe this low score is due to the dependency parser being only successful if the user provides sentences with specific grammatical patterns. When the sentence does not have the expected grammatical pattern, which user-generated issue reports sentences with various grammatical patterns not always meeting the expected patterns, this causes a significant issue for the system and results in lower scores.

5. CONCLUSION

This work aimed to create a NER model for implementing named entity recognition over user actions with applications and element detection over given sentences and finally executing test cases on applications under test (AUT). We prepared a training data set containing user action sentences and designed a labelling system for application test cases. We tested the trained model with different parameters, yielding acceptable results for future development. This novel model can be used for automated user interface testing, a domain with a high testing requirement. Moreover, in the literature, to the best of our knowledge, this is the first time a NER approach has been used to aid test automation for application UI testing.

Given a POM model of a screen and a free-form sentence written in a natural language specifying an action and a related UI element on which the action needs to be carried out, we have developed an approach based on named entity recognition and semantic similarity computations to automatically determine the actual UI element of interest on the screen and the action to be performed on this element and the parameter that this element can take.

After detailed evaluations and comparisons, we observed that the proposed element extraction approach method is accurate and open to be implemented in further work for automation. This line of research is exciting and quite promising. We have arrived at this conclusion by noting that in our empirical evaluations, the proposed approach correctly determined the action words with an average F-measure of 0.97, the actual UI elements of interest with an accuracy of 0.92 and the parameter that the UI elements can be assigned with an accuracy of 93.

Regarding the UI element detection step, we investigated avenues for evaluating the accuracy of different approaches for computing the semantic similarities of Google Universal Sentence Encoder Cer et al. (2018) and BERT Devlin et al. (2018). We also performed tests of different flavours of Google Universal Sentence encoder to see the effect of *Deep Averaging Networks* (DAN) and transformers. The results for element detection yielded a promising 0.84 N3 element detection score using the

trained BERT model, 0.85 N3 element detection score with *Google Sentence Encoder* with transformers and over 0.91 with deep averaging networks. Interestingly, the most straightforward variant DAN performs best in calculating semantic similarity on UI elements, which are basic noun phrases.

In this project, to create test cases that include the user action and the interacted entities, we used the issues of the mobile applications. Although there were around 10000 cases, we had to filter through them. At the end of this process, we extracted around 4500 sentences to train our NER model. Then we used these cases as training and validation data for the BERT training. However, the training samples need to include more sentences from different applications to effectively classify named entities. This issue is one internal threat to the project. While we cannot claim that our NER model can generalize for all types of applications, from our evaluation, we can conclude that we extract useful information with a certain degree of confidence. We tried to achieve this by preparing different data sets from 3 different applications to test the NER model. This mitigated some of the effects of having a limited number of samples.

Another internal threat the model faces is the biased distribution of the labels. Since most of the user's actions with an interface are single click or tap actions, the training data contained many more representations of this action. This distribution is, unfortunately, present in the test data as well. However, the high score of action detection for all BERT models shows that even though the action data has a biased distribution, the model detects various types of actions from the semantic information of the sentences.

Despite the internal threats mentioned above, we can confidently conclude that both the element extraction and element detection yielded promising results which helped in the *text2test* automation step. Our main goal is to implement an automation process where even non-technical stakeholders can easily contribute to UI testing. Toward this goal, a test suite is created with the user assertions. This also opens a path for test-driven development.

We presented the *text2test* system, which aims to automate the execution of test cases for AUT based on information extracted from natural language test sentences. The system employs an approach that combines element extraction and element detection techniques to identify the relevant GUI elements and perform the specified actions.

Through experimentation, we evaluated the effectiveness of the *text2test* system on a set of test cases derived from bug reports. The outcomes demonstrated a high

rate of test case execution success, with the majority of applications obtaining a success rate of 100%. However, we reported challenges in accurately executing test cases for specific applications, primarily due to issues such as missing descriptive information, and application crashes These findings offer valuable insights for improving the *text2test* system and overcoming these challenges. Future development of *text2test* can focus on the following: first, it can concentrate on fixing the shortcomings discovered during the experimental phase in order to further improve the *text2test* system, secondly, a test oracle can be developed to automatically detect the success of the actions performed by the test framework. It is possible to make improvements to how cases with lacking descriptive information are handled, how accurately items are detected, and how to handle application-specific problems like crashes or broken elements.

Overall, the *text2test* system demonstrates promising capabilities in automating the execution of test cases using natural language test sentences. By leveraging the strengths of element extraction and element detection techniques, coupled with an efficient DFS algorithm, the system has the potential to streamline the testing process and improve efficiency in software development.

BIBLIOGRAPHY

Agrawal, A., Tripathi, S., Vardhan, M., Sihag, V., Choudhary, G., & Dragoni, N. (2022). Bert-based transfer-learning approach for nested named-entity recognition using joint labeling. *Applied Sciences*, *12*(3).

Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., & Imparato, G. (2012). A toolset for gui testing of android applications. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, (pp. 650–653).

Anand, S., Naik, M., Harrold, M. J., & Yang, H. (2012). Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, (pp. 1–11).

Bernal-Cárdenas, C., Cooper, N., Moran, K., Chaparro, O., Marcus, A., & Poshyvanyk, D. (2020). Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, (pp. 309–321).

Brin, S. (1998). Extracting patterns and relations from the world wide web. In *WebDB*, (pp. 172–183).

Cer, D., Yang, Y., yi Kong, S., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C., Sung, Y.-H., Strope, B., & Kurzweil, R. (2018). Universal sentence encoder.

Coppola, R., Raffero, E., & Torchiano, M. (2016). Automated mobile ui test fragility: An exploratory assessment study on android. In *Proceedings of the 2nd International Workshop on User Interface Test Automation*, INTUITEST 2016, (pp. 11–20)., New York, NY, USA. Association for Computing Machinery.

Costa, P., Paiva, A. C., & Nabuco, M. (2014). Pattern based gui testing for mobile applications. In *2014 9th International Conference on the Quality of Information and Communications Technology*, (pp. 66–74). IEEE.

Dai, Z., Wang, X., Ni, P., Li, Y., Li, G., & Bai, X. (2019). Named entity recognition using bert bilstm crf for chinese electronic health records. In *2019 12th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, (pp. 1–5).

Developers, G. (2021). Android debug bridge.

Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, *abs/1810.04805*.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Ernst, M. D. (2017). Natural Language is a Programming Language: Applying Natural Language Processing to Software Development. In Lerner, B. S., Bodík, R., & Krishnamurthi, S. (Eds.), *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (pp. 4:1–4:14)., Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Etzioni, O., Cafarella, M., Downey, D., Popescu, A.-M., Shaked, T., Soderland, S., Weld, D. S., & Yates, A. (2005). Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intelligence*, *165*(1), 91–134.

Fazzini, M., Prammer, M., d'Amorim, M., & Orso, A. (2018). Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, (pp. 141–152)., New York, NY, USA. Association for Computing Machinery.

Fischbach, J., Vogelsang, A., Spies, D., Wehrle, A., Junker, M., & Freudenstein, D. (2020). Specmate: Automated creation of test cases from acceptance criteria. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, (pp. 321–331). IEEE.

Gao, J., Bai, X., Tsai, W.-T., & Uehara, T. (2014). Mobile application testing: A tutorial. *Computer*, *47*(2), 46–55.

Granda, M. F., Parra, O., & Alba-Sarango, B. (2021). Towards a model-driven testing framework for gui test cases generation from user stories. In *ENASE*, (pp. 453–460).

Halpern, M., Zhu, Y., Peri, R., & Reddi, V. J. (2015). Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, (pp. 215–224). IEEE.

Honnibal, M. & Montani, I. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear.

Huggingface (2019). Bert model finetuning using masked language modeling objective.

Karpathy, A., Johnson, J., & Fei-Fei, L. (2015). Visualizing and understanding recurrent networks.

Kirubakaran, B. & Karthikeyani, V. (2013). Mobile application testing — challenges and solution approach through automation. In *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, (pp. 79–84).

Lafferty, J. D., McCallum, A., & Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *International Conference on Machine Learning*.

Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., & Dyer, C. (2016). Neural architectures for named entity recognition. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (pp. 260–270)., San Diego, California. Association for Computational Linguistics.

Li, N., Escalona, A., & Kamal, T. (2016). Skyfire: Model-based testing with cucumber. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, (pp. 393–400). IEEE.

Limsopatham, N. & Collier, N. (2016). Bidirectional lstm for named entity recognition in twitter messages. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, (pp. 879–888). COLING 2016.

Lin, B. Y. & Lu, W. (2018). Neural adaptation layers for cross-domain named entity recognition. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, (pp. 2012–2022).

Lin, J.-W., Wang, F., & Chu, P. (2017a). Using semantic similarity in crawling-based web application testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, (pp. 138–148). IEEE.

Lin, J.-W., Wang, F., & Chu, P. (2017b). Using semantic similarity in crawling-based web application testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, (pp. 138–148).

Ling, W., Luís, T., Marujo, L., Astudillo, R. F., Amir, S., Dyer, C., Black, A. W., & Trancoso, I. (2015). Finding function in form: Compositional character models for open vocabulary word representation. *CoRR*, *abs/1508.02096*.

Liu, C. H., Lu, C. Y., Cheng, S. J., Chang, K. Y., Hsiao, Y. C., & Chu, W. M. (2014). Capture-replay testing for android applications. In *2014 International Symposium on Computer, Consumer and Control*, (pp. 1129–1132). IEEE.

Ma, X. & Hovy, E. (2016). End-to-end sequence labeling via bi-directional lstm-cnns-crf.

Mahalakshmi, G., Vijayan, V., & Antony, B. (2018). Named entity recognition for automated test case generation. *Int. Arab J. Inf. Technol.*, *15*(1), 112–120.

McCallum, A., Freitag, D., & Pereira, F. C. N. (2000). Maximum entropy markov models for information extraction and segmentation. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, (pp. 591–598)., San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013a). Efficient estimation of word representations in vector space.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013b). Efficient estimation of word representations in vector space. *CoRR*, *abs/1301.3781*.

Moreira, R. M. & Paiva, A. C. (2014). Pbgt tool: an integrated modeling and testing environment for pattern-based gui testing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, (pp. 863–866).

Muccini, H., Di Francesco, A., & Esposito, P. (2012). Software testing of mobile applications: Challenges and future research directions. In *2012 7th International Workshop on Automation of Software Test (AST)*, (pp. 29–35).

North, D. et al. (2006). Introducing bdd. *Better Software*, *12*.

Palmer, D. D. & Day, D. S. (1997). A statistical profile of the named entity task. In *Fifth Conference on Applied Natural Language Processing*, (pp. 190–193)., Washington, DC, USA. Association for Computational Linguistics.

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, (pp. 1532–1543).

Ramshaw, L. & Marcus, M. (1995). Text chunking using transformation-based learning. In *Third Workshop on Very Large Corpora*.

Ritter, A., Clark, S., Mausam, & Etzioni, O. (2011). Named entity recognition in tweets: An experimental study. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, (pp. 1524–1534)., USA. Association for Computational Linguistics.

Rocktäschel, T., Weidlich, M., & Leser, U. (2012). Chemspot: a hybrid system for chemical named entity recognition. *Bioinformatics*, *28*(12), 1633–1640.

SmartBear (2019). What is cucumber?

Soeken, M., Wille, R., & Drechsler, R. (2012). Assisted behavior driven development using natural language processing. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, (pp. 269–287). Springer.

Solis, C. & Wang, X. (2011). A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, (pp. 383–387). IEEE.

Song, W., Qian, X., & Huang, J. (2017). Ehbdroid: beyond gui testing for android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (pp. 27–37). IEEE.

Souza, F., Nogueira, R. F., & de Alencar Lotufo, R. (2019). Portuguese named entity recognition using BERT-CRF. *CoRR*, *abs/1909.10649*.

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., & Su, Z. (2017). Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, (pp. 245–256).

Syed, M. H. & Chung, S.-T. (2021). Menuner: Domain-adapted bert based ner approach for a domain with limited dataset and its application to food menu domain. *Applied Sciences*, *11*(13).

Tao, C., Gao, J., & Wang, T. (2017). An approach to mobile application testing based on natural language scripting. In *SEKE*, (pp. 260–265).

Tjong Kim Sang, E. F. & De Meulder, F. (2003). Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, (pp. 142–147).

Viterbi, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, *13*(2), 260–269.

Weber, L., Sänger, M., Münchmeyer, J., Habibi, M., Leser, U., & Akbik, A. (2021). HunFlair: an easy-to-use tool for state-of-the-art biomedical named entity recognition. *Bioinformatics*, *37*(17), 2792–2794.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., & Brew, J. (2019). Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, *abs/1910.03771*.

Zhang, X., Zhao, J. J., & LeCun, Y. (2015). Character-level convolutional networks for text classification. *CoRR*, *abs/1509.01626*.

Zhang, Y. & Zhang, H. (2022). Finbert-mrc: financial named entity recognition using bert under the machine reading comprehension paradigm.

Zhao, Y., Su, T., Liu, Y., Zheng, W., Wu, X., Kavuluru, R., Halfond, W. G. J., & Yu, T. (2022). Recdroid+: Automated end-to-end crash reproduction from bug reports for android apps. *ACM Trans. Softw. Eng. Methodol.*, *31*(3).

Zhao, Y., Yu, T., Su, T., Liu, Y., Zheng, W., Zhang, J., & G.J. Halfond, W. (2019). Recdroid: Automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, (pp. 128–139).

Zhou, C., Li, B., & Sun, X. (2020). Improving software bug-specific named entity recognition with deep neural network. *Journal of Systems and Software*, *165*, 110572.