



# Transformers for GUI Testing: A Plausible Solution to Automated Test Case Generation and Flaky Tests

**Zubair Khaliq, Sheikh Umar Farooq, and Dawood Ashraf Khan**, University of Kashmir

*Artificial intelligence has made significant progress in software testing as it is being incorporated into all stages of the software test lifecycle (STLC). However, formidable challenges remain within software testing activities in the STLC.*

**A**rtificial intelligence (AI) is gradually changing the landscape of software engineering in general<sup>1</sup> and software testing in particular,<sup>2</sup> both in research and industry. In the last two decades, the rapid growth of interest in topics where AI has been applied to software testing is a testimony to the appetite the software testing community has for AI. This mercurial temperament is a consequence of AI providing efficient solutions to the longing problems faced by the testing community. This is due to the superior abilities of AI in automated testing, which provide insightful data on

application risks, create test cases, and eradicate repetitive work from testers.

Various AI-related works have been carried out for the task of automated test suite generation. Khari et al.<sup>3</sup> compare the efficiency of most well-known AI techniques for the task of test suite generation. Other AI-based techniques have been used for test suite optimization.<sup>4</sup> Although AI is continuously being integrated at all levels of the test pyramid (unit, integration, and GUI), its current impact on GUI-level testing is still in its infancy. Testing at the GUI level requires prerequisite technical concepts, thus functional tests on a GUI are usually conducted by testers with good experience of using automation tools like GUITAR,<sup>5</sup> Selenium,<sup>6</sup> and Appium.<sup>7</sup> These

Digital Object Identifier 10.1109/MC.2021.3136791  
Date of current version: 11 March 2022

tools provide a mechanism for interacting with the GUI elements of an executing application as a part of automated testing.

The use of automation tools for testing at the GUI level is supported because these tools enable regression testing (a type of testing designed to confirm that existing functionality is still working when changes are made to an application or environment), enhance the speed of test execution, and help save time for testers. However, the use of such automation tools is also heavily censured because of its failure to automatically generate test cases, which, in the current scenario, are mostly compiled manually from a test specification completely relying on human intelligence and creativity. Many studies bring out other opprobrium, such as flaky test cases.<sup>8,9</sup> One of the many factors that may lead to flakiness is the inability to repeat test execution in a reliable manner, primarily caused by a modification to the system under test (SUT). Such problems force testers to carry out manual testing at the GUI level, resulting in more effort and elevated cost. Realizing the impact of AI in the domain of software testing,<sup>2</sup> we ascertain the potential of AI in addressing these issues.

In this article, we propose a learning-based approach capable of producing near-human level test cases automatically. We postulate generating test cases for the SUT from the structure of the GUI using a language-based transformer rather than relying on the test specification document for generating test cases. Because the GUI test case is a sequence of events, we map the sequence of events to test cases using a sequence-to-sequence transformer model. Further, the experiments prove that the proposed approach helps to repair flaky tests when the SUT

goes through any modification without manual intervention.

We summarize our contributions as follows:

- › We propose an automated test case generation approach for GUI testing using a sequence-to-sequence transformer model. This approach is novel in the sense that it uses the element classification system to perceive the application state and generates testflows in the English language according to the perceived state.
- › We evaluate our model for repairing flaky tests in case the GUI goes through some modification. The model should behave in a way such that a modification to the GUI should generate new testflows for regression automatically without requiring testers to change the executing script.
- › We demonstrate that the generated testflows, which are abstract in the sense that they are generated in the English language, can be translated to concrete testflows (executable test scripts) using a simple parser.

## METHODOLOGY

The first part, namely, the “Data Set Collection and Fine-Tuning” section, discusses the training process from data collection to fine-tuning the model. The process includes fine-tuning a pre-trained sequence-to-sequence transformer to act as a translator able to transform the description of the GUI to testflows (in the English language). In the “Execution Process” section, we discuss the environment wherein actual test execution is carried out using the proposed approach.

## Data set collection and fine-tuning

To collect the data set needed to fine-tune our model, we selected e-commerce mobile applications on the Android platform for our case study. We chose these applications because they are modified frequently, so we expect more flakiness in testflows. As our case study involves Android applications, we use Appium to extract the XML hierarchy and execute test scripts. Figure 1 depicts the training process.

**Data collection.** The e-commerce applications were selected from both the App Store and the Play Store. The inclusion criteria (reviews) included applications with the number of aggregate reviews greater than “600.” After applying the inclusion criteria, in total 86 e-commerce applications were selected for the study.<sup>10</sup>

We can extract the elements of applications by means of the following two approaches:

1. Using the XmlPullParser interface<sup>11</sup> in the org.xmlpull.v1 package of the Android platform. Every individual element on the GUI is identified by a tag name in the XML. We call such elements *naive*, as they are just classified based on their structure (for example, Button, Text-Box, Combo, and so on). To generate correct testflows, we need additional information regarding the components (for example, to distinguish between a login button and a sign-up button we need a textual label on the buttons). Such information is associated as a property of the element and included inside each tag. We call an element with

associated property values a *complete* element. We also require coordinates of each identified element with respect to the GUI to interact with the elements using Appium.

2. The applications that render GUIs directly to an image buffer (for example, web canvas applications) generally cannot have their GUI structure extracted automatically. The pixels are drawn directly to the screen, and there is no underlying XML or HTML structure to extract widget locations. The prior research to address this issue is carried out by White et al.,<sup>12</sup> proposing a technique to identify GUI elements solely through visual information. The authors use object-detection models based on convolutional neural networks to identify the elements of a UI.

Description of the GUI structure is represented using a text string, wherein each element is separated by a “,” (for example, “Text Box Email, Text Box Password, Button Sign Up”), and each individual element is represented by a combination of tag name and textual information (for example, “Text Box” and “Password”). This GUI description will be the input variable of our data set. The coordinates and their corresponding elements are stored in a list to be used

later during execution. Now we require corresponding labels for input variables in our data set, which we compile in the “Handcrafting Testflows” section.

**Handcrafting testflows.** A *testflow* is a series of steps representing a test case, and the individual steps in such a flow are called *test steps*. We record a testflow (in English) for every possible description of the GUI structure (extracted in the “Data Collection” section) representing a correct test case according to the operations that can be performed on the elements. The testflows are the labels of our data set. We call a description associated with its corresponding testflow a *test mapping*. The testflows for training were selected from the following common test cases:

- › login functionality
- › registration functionality
- › product addition to cart
- › checkout from cart
- › search based on product name.

The following are examples hand-written testflows:

- › Enter User Name in Text Box User Name, Enter password in Text Box Password, Click on Login button
- › Enter product name in Text Box Search, Click on Search Button.

Most of the test steps in a testflow need to follow a particular order allowing

for dependency in test steps, which is called *intratest-flow dependency (ITD)*. For example, a testflow on a login page with the intent to login has dependency among the test steps, in that the user name and password text fields need to be populated first, only then can the login button be clicked. As neither of the approaches for extracting the element descriptions (presented in the “Data Collection” section) ensure descriptions to be in ITD-preserving order, while populating the data set, we swapped the position of the element descriptions (the input variable of our data set) according to ITD ordering (we call such a GUI description an *ITD-preserving GUI* one), and while creating testflows for the descriptions we ensured that the ITD is preserved for each testflow (we call such a testflow an *ITD-preserving* one).

**The transformer.** We used GPT-2 as the transformer for the translation task. GPT-2 is a transformer model pretrained on a very large corpus of English data in a self-supervised fashion. In architecture, GPT-2 is built using transformer decoder blocks only. GPT-2 is an auto-regressive model because after each token is produced, that token is added to the sequence of inputs. Primarily created for generating text on a level that people found hard to distinguish from prose written by humans, it generalizes well to other tasks like translation of text, question answering,



FIGURE 1. Fine-tuning of the GPT-2 on the mapped testflows.

summarizing text, and even code generation. It is a general-purpose learner and was not specifically trained to do any of these tasks, but its ability to perform them is an extension of its general ability to accurately synthesize the next item in an arbitrary sequence.<sup>13</sup> GPT-2 was released under four model dimensions, each differing in the number of parameters trained with and in the number of decoder layers. The largest variant has 48 decoder blocks and 1.5 billion parameters, while the smallest one has 12 decoder blocks and 117 million parameters.

**Fine-tuning the model.** GPT-2 comes with different pretrained model sizes. We take the advantage of transfer learning (a technique in which already-trained weights of a model pretrained on a high volume of data are used) and just fine-tune GPT-2 for our translation task without training it from scratch. Fine-tuning is the process of tweaking a model trained on a large data set to a particular (likely much smaller) data set. We want GPT-2 to learn a mapping from the GUI description to testflows.

We perform fine-tuning on all the variants of the GPT-2 model using the

mapped test cases where each test mapping is represented using a hypothesis  $f: X \rightarrow Y$  which consists of pairs  $(x, f(x))$ . The  $x$  in the pair is the GUI description, and  $f(x)$  is the mapped testflow.

Language models like GPT-2 are called *few-shot learners* because they showed good results for tasks on which they are not trained.<sup>13</sup> Few-shot learners need only be fine-tuned for the tasks on which they are not trained.

### Execution process

Figure 2 illustrates the execution process. We start with an application and run it for testing on the Appium server using the pytest framework. A selector is used to select a test based on the element description with affixed test data. The affixed element description is fed to the fine-tuned GPT-2 model to generate a testflow. The affixed element description is input as a final prompt of an English sentence, and we sample from the model with greedy decoding and use the first generated sentence as the testflow. The testflow is then fed to a parser to translate the testflow into a test script, and the generated executable test script is then executed on the same application using Appium.

## EXPERIMENT DESIGN

The goal of our study is to determine whether our proposed approach can generate testflows similar to designer-written testflows given a GUI description. Our experiments seek to answer the research questions described in the following paragraphs.

Our study starts with identifying whether it is necessary to fine-tune the model for a testflow generation task (RQ<sub>1</sub>). Next we try to compare the performance of each model variant of GPT-2 to the testflow generation task (RQ<sub>2</sub>). We also try to study the impact of adding supplementary information to the GUI description on the testflow generation process (RQ<sub>3</sub>). We evaluate our proposed approach to automatically generate correct testflows if the SUT undergoes modifications (RQ<sub>4</sub>). Finally, we study the impact of ITD on the process of testflow generation using the trained model (RQ<sub>5</sub>).

We use the Bilingual Evaluation Understudy (BLEU) score metric for evaluating the generated testflows. The BLEU score is a widely adopted metric developed for evaluating the predictions made by automatic machine-translation systems.<sup>14</sup> A score

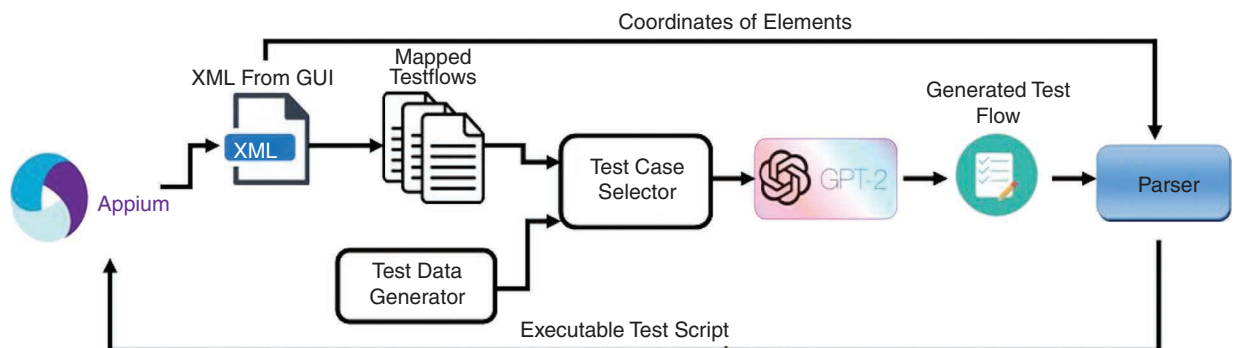


FIGURE 2. The overall setup for the actual test case execution.



of “1” represents fully matched sentence pairs, while a score of “0” is a representation of completely unmatched sentences. A value between one and zero gives a similarity score of the contending sentences.

#### **RQ<sub>1</sub>: Is it necessary to fine-tune the model?**

First, we try to study whether we need to fine-tune the model for mapping testflows from element descriptions. If our testflow mapping process is also translated well by GPT-2 without

an indication of the correctness of the testflows generated by a particular model.

#### **RQ<sub>3</sub>: How does adding supplementary information impact the generation of testflows?**

The coverage of a test case is highly dependent on the test data, and it is for this purpose that we want to attach test data to the input testflows and study the effect of adding this data to the testflow. The examples

both of these testflows will receive the same GUI description from the XML. To differentiate between the functionality to check, we add functionality context to the GUI description and study the resultant behavior of the model.

#### **RQ<sub>4</sub>: What is the percentage of modifications that our fine-tuned model catches?**

Apart from trying to generate test cases automatically from the GUI description, the purpose of our study is to find a mechanism that would automatically regenerate a new testflow when a change to a GUI is performed. The change may include the inclusion of an element or deletion of an element, or a modification to the GUI, such as changing a text property or location of an element, or adding an item to a combo box, and so on. When such modifications are applied to the GUI of an application, it renders the previously created testflows fragile. Such fragile and brittle testflows are called flaky tests in the literature.

**WE OBSERVE THAT HIGHER  
DIMENSIONAL MODELS SHOW IMPROVED  
VALIDATION LOSS OVER LOWER  
DIMENSIONAL ONES.**

fine-tuning, then we can skip altogether the process of fine-tuning the model.

#### **RQ<sub>2</sub>: How do the different model variants of GPT-2 compare to one another when fine-tuned for testflow generation?**

This research question aims at identifying the relationship of each model variant to the testflow generation task. For this, we fine-tuned each variant with 2,100 prewritten test mappings. We calculate the cross-entropy loss on the test set (validation loss), and we also calculate the average of the BLEU score of all testflows from the test set. The cross-entropy loss will give us a measurement of the performance of model variants with each other, while the BLEU score of all the test samples provides us with

include prefixing strings like “valid,” “invalid,” and so forth to the Text-Box element description and verifying whether correct testflows are generated (for instance, for the description “Invalid Text Box Email,” the test step “Enter Invalid Email in Text-Box Email” should be generated). For such a testflow, a successful test is one that gives a correct error message, and to observe the error messages, we append element descriptions with “Observe Message” strings. A verification of this behavior is called a *test oracle*.

Additionally, for some testflows, we analyzed that the GUI description is the same but that a different testflow needs to be generated. For example, on a login page, we might want to test the login functionality, and also, we might want to test whether the “Forgot Password” link is working correctly; however,

#### **RQ<sub>5</sub>: What is the overall impact of an ITD-preserving property in testflow generation?**

How much of the ITD is preserved in the generated test cases? An answer to this question is certain from the results of experiments carried out in RQ<sub>2</sub>, where the higher BLEU scores of model variants meant that ITD is preserved in the generated testflows. The experiment results of RQ<sub>2</sub> were calculated on a data set in which both the GUI description and the testflows were order preserving, but

- ▶ what if the GUI descriptions (input variables in our data set) are not ITD-preserving descriptions?

- › what if the testflows in our data set are not ITD-preserving ones?
- › what if both the GUI descriptions and testflows are not ITD-preserving ones?

To answer these questions, we carried out a cross-sectional study over half of the original data set (the instances were selected randomly to include testflow mappings from all five selected test cases). This version preserves ITD ordering in both the GUI description and the corresponding testflows (V1). We created three additional versions of the data set: V2, V3, and V4. We created V2 by randomly swapping the element descriptions without swapping the test steps in the testflow. This ensures an ITD-preserving order in testflows but not in the GUI description. Similarly, V3 ensures an ITD-preserving order in the GUI description but not in the testflows, and V3 does not ensure an ITD-preserving order in either the GUI description or the testflows.

## EXPERIMENT RESULTS

### RQ<sub>1</sub>: Is it necessary to fine-tune the model?

The next sections detail the results of our study. We input three types of prompts to each model size of GPT-2. Each prompt contains pairs of the GUI description and its associated testflows. In prompt1( $p_1$ ), we add one training example and evaluate the output for the prompt. We measure the similarity between the output of the model and the actual output using the BLEU score. We added five training examples in prompt2 ( $p_2$ ) and 10 in prompt3 ( $p_3$ ) and measured the BLEU score. The

subscript on each value denotes the particular prompt. The following are the recorded values:

- › gpt  $\rightarrow 0.32_{p_1} \rightarrow 0.43_{p_2} \rightarrow 0.49_{p_3}$
- › gpt-medium  $\rightarrow 0.42_{p_1} \rightarrow 0.48_{p_2} \rightarrow 0.52_{p_3}$
- › gpt-large  $\rightarrow 0.42_{p_1} \rightarrow 0.49_{p_2} \rightarrow 0.54_{p_3}$
- › gpt-xl  $\rightarrow 0.45_{p_1} \rightarrow 0.52_{p_2} \rightarrow 0.55_{p_3}$ .

From the values, we infer three important results. First, all the model variants are bad at predicting testflows from element descriptions without fine-tuning. The highest BLEU score we achieve is for the gpt-xl variant (0.55), where we feed it with 10 training examples. It turns out that using even the largest model variant still generates unacceptable testflows. However, as is evitable from the recorded values, adding more training examples in the form of prompts to the model variants will increase their effectiveness in the generation of testflows. Upon this result, we

build our foundation to carry out the research further and hypothesize that fine-tuning all the model variants will likely increase their effectiveness in the generation of testflows from GUI descriptions. From the recorded values, we also infer that the effectiveness of model variants increases in line with the size of the model for all the prompts, and this persuades us to study RQ<sub>2</sub> to achieve a good understanding of individual model performances.

### RQ<sub>2</sub>: How do the different model variants of GPT-2 compare to one another when fine-tuned for testflow generation?

Figure 3 shows the cross-entropy loss on the test set recorded while evaluating the model variants. We plot the loss on a log scale for the purpose of better visualization. The higher dimensional models start with a low loss of  $-3.765_{\text{LogScale}}$  for gpt-xl, as compared to  $-3.351_{\text{LogScale}}$  for gpt, highlighting the fact that higher dimensional models

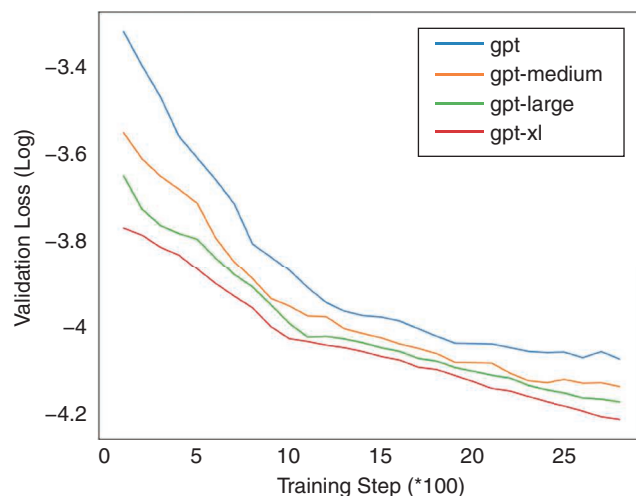


FIGURE 3. The validation loss while fine-tuning model variants.

tend to learn the testflow generation process fast. We observe that higher dimensional models show improved validation loss over lower dimensional ones. We also observe that the validation loss between gpt and gpt-medium ( $4.605 \times 10^{-3}$ ) is significant when compared to the validation loss between gpt-medium and gpt-large ( $2.529 \times 10^{-3}$ ) and between gpt-large and gpt-xl ( $1.963 \times 10^{-3}$ ). This observation gives us an indication of which model variant to use when we are to decide on a tradeoff between performance and resources (for example, when we are low on resources, gpt-medium is a good choice as it is better than gpt and much closer to gpt-large and gpt-xl on validation loss). The following list shows the average BLEU scores of the model variants from the test set:

- › gpt  $\rightarrow 98.171 \times 10^{-2}$
- › gpt-medium  $\rightarrow 99.235 \times 10^{-2}$

- › gpt-large  $\rightarrow 99.587 \times 10^{-2}$
- › gpt-xl  $\rightarrow 99.998 \times 10^{-2}$ .

From the results, we infer that the higher the model is in dimensionality, the higher the BLEU score it produces. The BLEU score is calculated on a sample of 210 testflows as we split the training set in a 90:10 ratio. We deduce that after fine-tuning, all variants of GPT-2 generate more correct testflows.

**RQ<sub>3</sub>: How does adding supplementary information impact the generation of testflows?**

Some of the testflows generated using the mutated training examples are listed in Table 1. The gpt-medium was again used for this experiment. We calculated the BLEU score for the 210 test examples of this mutated training data set ( $99.946 \times 10^{-2}$ ) against the unmutated data set ( $99.235 \times 10^2$ ) and found a score difference of  $7.106 \times 10^{-3}$ ,

which is significantly low. We deduce that adding the aforementioned supplementary information slightly increases the performance of testflow generation by learning each of the following behaviors:

- › **Add test data:** The results showed significant improvement when test data were added to the GUI elements. The model learned the behavior of adding the same test data to the respective testflows.
- › **Verify behavior:** The model also learned the exact mappings when fed with this information. The model generated “Observe Message” suffixed to the testflow for which it sees the same suffix in the GUI description.
- › **Add functionality context:** We learned that this information helped the model predict the intent of the testflow.

TABLE 1. The generated testflows by mutating the GUI description with supplementary information.		
Supplementary information added	Mutated GUI description	Generated testflows
Without added information	Text box email, text box password, button back, button login, link forgot password, and button create an account	Enter email in text box email, enter password in text box password, and click on “login” button.
Test data	Valid text box email, invalid text box password, button back, button login, and link forgot password	Enter valid email in text box email, enter invalid password in text box password, and click on “login” button.
Verify behavior	Text box email, text box password, button back, button login, link forgot password, and not observe message	Enter email in text box email, enter password in text box password, click on “login” button, and not observe message.
Add functionality context	Check login, text box email, text box password, button back, button login, link forgot password, and button create an account	Enter email in text box email, enter password in text box password, and click on “login” button.
Add functionality context	Check link forgot password, text box email, text box password, button back, button login, link forgot password, and button create an account	Click on “Forgot Password Link.”

#### RQ<sub>4</sub>: What is the percentage of modifications that our fine-tuned model catches?

Because modifying an application is equivalent to modifying the XML of said application, which consequently means modifying the element description of the GUI, we modified the element description of the 210 test examples from the test set for testing purposes in the experiment for RQ<sub>3</sub>. We applied a total of 725 modifications to the test samples manually and changed the testflows from the same test mapping accordingly. The changed testflows now represent the actual testflows. The modified GUI descriptions from the test set were executed on the model trained in our experiment for RQ<sub>3</sub> to produce predicted testflows. We later calculated the average BLEU score for all the predicted testflows against the original testflows. We observed a BLEU score of “1,” apparently highlighting the performance of the model in catching all the modifications to the GUI structure of the application.

#### RQ<sub>5</sub>: What is the overall impact of an ITD-preserving property in testflow generation?

We fine-tuned gpt-medium on all these versions. We recorded a BLEU score of  $99.0685 \times 10^{-2}$  for V1,  $99.0689 \times 10^{-2}$  for V2,  $99.0679 \times 10^{-2}$  for V3, and  $99.0682 \times 10^{-2}$  for V4. From these results, we infer that if we want an ITD-preserving order in the generated testflows, we just need to make sure the ITD property is preserved in the testflows of the data set with which the model is trained. This consequently means that in the process of handcrafting testflows (as discussed in the “Handcrafting Testflows” section), we do not need to reorder GUI descriptions as per the ITD ordering, but we need to make sure that the testflows are

written to preserve the ITD property. This result is important for people who want to compile a different data set for the same approach as ours.

### GENERATING TEST SCRIPTS FROM THE TESTFLOWS

In this section, we discuss a parsing mechanism that generates test scripts meant to be executed in the test environment. For this, we created a simple English-language parser that analyzes the testflows and generates test scripts. The parser heavily relies on the structure of the generated testflow. In designing the parser, we take the advantage of the limited number of elements present in the developer frameworks (fewer than 20 for Android, and examples include TextView, EditText, Button, ImageButton, ToggleButton, and so on) and a limited number of actions that can be performed on each element (for instance, “clicking” a button, “Send-Keys” on a text box, and so forth).

Each of the test steps inside the testflow generated by the model is separated by a “,”. We segregate the test steps on “,” which acts as a delimiter. Next we discuss the functioning of the parser on the following two simple example test steps:

1. Enter Valid First Name in Text Box First Name.
2. Click on Sign In Button.

The type of test step is identified from the first substring of the test step, which in the first example is “Enter,” and in the second example is “Click.” The type of step gives the parser additional information regarding the test step. For the first example, step type “Enter” tells the parser that what follows the step type (“Click”) is a description of a “text string to be input” and a description of


“text box in which the text string is to be input,” both separated from each other via the “in” string. Similarly, for the second example, the test step “Click” informs the parser that what follows the “on” string in the test step is the “button which is to be clicked.” The parser uses this information to generate the corresponding test script. The following are sample test scripts for the given examples:

---

```
1) self.driver.find_element_by_xpath('//android.widget.EditText[@text =“identifier-of-textbox”]).send_keys(“text-string-to-be-input”)
2) TouchAction(self.driver).tap(x=34,y=254).perform().
```

---

Code snippet 1 interacts with the elements using locator strategies, while code snippet 2 directly interacts with the elements by tapping on the location of the element. The latter method provides flexibility to interact with GUI elements of the web canvas applications without actually having a description of the XML. The parser is created to generate test scripts for the Python language. This approach provides flexibility as we can fine-tune the parser to generate scripts in any language.

 Our work and experiments evaluate the impact and potential of transformer models to generate GUI test cases automatically from the GUI of an application by learning from developer-written test cases. The evidence from our experiments strongly supports the use of such models for the task of test case generation and test case repairing. Highly encouraging results from our experiments indicate that upon



## ABOUT THE AUTHORS

**ZUBAIR KHALIQ** is a Ph.D. candidate in the Department of Computer Sciences, North Campus, University of Kashmir, Delina, Baramulla, Jammu And Kashmir, 193103, India. His research interests include automated software testing, GUI testing, machine learning, and deep learning. Khaliq received a master's in computer applications from the University of Kashmir with a distinction. Contact him at zikayem@gmail.com.


**SHEIKH UMAR FAROOQ** is an assistant professor in the Department of Computer Sciences, North Campus, University of Kashmir, Delina, Baramulla, Jammu And Kashmir, 193103, India. His research interests focus on empirical software

testing, defect prediction, and software engineering education. Farooq received a Ph.D. in computer sciences from the University of Kashmir. He is a member of ACM and Computer Society of India. Contact him at suf.cs@uok.edu.in

**DAWOOD ASHRAF KHAN** is an assistant professor in the Department of Computer Sciences, North Campus, University of Kashmir, Delina, Baramulla, Jammu And Kashmir, 193103, India. His research interests focus on safety-critical embedded systems, deep learning, and distributed systems. Khan received a Ph.D. in computer science and engineering from the University of Lorraine. Contact him at dawood.khan@uok.edu.in.

fine-tuning, general-purpose language models like GPT-2 can be leveraged for these tasks. From the results of our experiments, we also conclude that these models are able to preserve the ITD property of testflows. Further, we demonstrated how testflows generated by such models can be translated to executable test scripts.

The work was limited to studying a particular category of application domain; however, this study paves the way for researchers and practitioners to carry out likely studies for other domains as well. More test cases also need to be incorporated into the future data sets. As GPT-2 is a decoder-only transformer, a likely study on other types of models (encoder only and encoder decoders) will provide better insights into which model is performing better. Upon analyzing application structures and behaviors, we found a higher disparity in the test oracle, but with more data, correct test oracles will be generated by such models. Looking further ahead, we believe, following on this study in

the future will help the software testing industry keep pace with the overly changing dynamic of the “continuous integration/continuous delivery” (CI/CD) pipeline. 

## REFERENCES

1. M. Harman, “The role of artificial intelligence in software engineering,” in *Proc. 2012 1st Int. Workshop Realizing Artif. Intell. Synergies Softw. Eng. (RAISE)*, Zurich, Switzerland, pp. 1–6, doi: 10.1109/RAISE.2012.6227961.
2. H. Hourani, A. Hammad, and M. Lafi, “The impact of artificial intelligence on software testing,” in *Proc. 2019 IEEE Jordan Int. Joint Conf. Elect. Eng. Inf. Technol. (JEEIT)*, pp. 565–570, doi: 10.1109/JEEIT.2019.8717439.
3. M. Khari, A. Sinha, E. Herrerra-Viedma, and R. G. Crespo, “On the use of meta-heuristic algorithms for automated test suite generation in software testing,” in *Toward Humanoid Robots: The Role of Fuzzy*
4. M. Khari, P. Kumar, and G. Shrivastava, “Enhanced approach for test suite optimisation using genetic algorithm,” *Int. J. Comput. Aided Eng. Technol.*, vol. 11, no. 6, pp. 653–668, 2019, doi: 10.1504/IJCAET.2019.102496.
5. B.-N. Nguyen, B. J. Robbins, I. Banerjee, and A. Memon, “GUITAR: An innovative tool for automated testing of GUI-driven software,” *Automat. Softw. Eng.*, vol. 21, no. 1, pp. 65–105, 2013, doi: 10.1007/s10515-013-0128-9.
6. Selenium, “Selenium automates browsers.” <https://www.selenium.dev> (Accessed: Jun. 6, 2021).
7. “Introducing Appium,” Appium. <http://appium.io> (Accessed: Jun. 5, 2021).
8. A. Contan, C. Dehelean, and L. Miclea, “Test automation pyramid from theory to practice,” in *Proc. 2018*

- IEEE Int. Conf. Automat., Qual. Testing, Robot. (AQTR), pp. 1–5, doi: 10.1109/AQTR.2018.8402699.
9. R. Coppola, L. Ardito, M. Torchiano, and E. Alégroth, “Translation from layout-based to visual android test scripts: An empirical evaluation,” *J. Syst. Softw.*, vol. 171, p. 110,845, Jan. 2021, doi: 10.1016/j.jss.2020.110845.
  10. “DATA/top apps.csv,” GitHub, 2019. <https://github.com/Zubair2019/DATA/blob/main/top%20apps.csv>
  11. “XmlPullParser,” Android Developers. <https://developer.android.com/reference/org/xml-pull/v1/XmlPullParser> (Accessed: Jun. 10, 2021).
  12. T. D. White, G. Fraser, and G. J. Brown, “Improving random GUI testing with image-based widget detection,” in *Proc. 2019 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, pp. 307–317, doi: 10.1145/3293882.3330551.
  13. A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019. [Online]. Available: <https://github.com/openai/gpt-2>
  14. K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: A method for automatic evaluation of machine translation,” in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2002, pp. 311–318, doi: 10.3115/1073083.1073135.



# Call for Articles

## IEEE Pervasive Computing

seeks accessible, useful papers on the latest peer-reviewed developments in pervasive, mobile, and ubiquitous computing. Topics include hardware technology, software infrastructure, real-world sensing and interaction, human-computer interaction, and systems considerations, including deployment, scalability, security, and privacy.

**Author guidelines:**  
[www.computer.org/mc/pervasive/author.htm](http://www.computer.org/mc/pervasive/author.htm)

**Further details:**  
[pervasive@computer.org](mailto:pervasive@computer.org)  
[www.computer.org/pervasive](http://www.computer.org/pervasive)

**IEEE pervasive COMPUTING**  
 MOBILE AND UBIQUITOUS SYSTEMS