Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automated Web application testing driven by pre-recorded test cases[☆]

Nezih Sunman[a,b], Yiğit Soydan[b], Hasan Sözer[b,*]

[a] Corparate Technology, Siemens A.S., Istanbul, Turkey
[b] Ozyegin University, Istanbul, Turkey

A R T I C L E I N F O

Article history:
Received 30 March 2021
Received in revised form 28 April 2022
Accepted 14 July 2022
Available online 22 July 2022

Keywords:
Web application testing
Model-based testing
Test case generation
Test automation
Exploratory testing

A B S T R A C T

There are fully automated approaches proposed for Web application testing. These approaches mainly rely on tools that explore an application by crawling it. The crawling process results in a state transition model, which is used for generating test cases. Although these approaches are fully automated, they consume too much time and they usually require manual configuration. This is due to the lack of insight and domain knowledge of crawling tools regarding the application under test. We propose a semi-automated approach instead. We introduce a tool that takes a set of recorded event sequences as input. These sequences can be captured during exploratory tests. They are replayed as pre-recorded test cases. They are also exploited for steering the crawling and test case generation process. We performed a case study with 5 Web applications. These applications were randomly tested with state-of-the-art tools. Our approach can reduce the crawling time by hours, while compromising the coverage achieved by 0.2% to 7.43%. In addition, our tool does not require manual configuration before crawling. The input for the tool was created within 15 min of exploratory testing.

1. Introduction

Testing is essential for ensuring the reliability of software systems. However, it might lead to significant costs depending on the required reliability level as well as the size and complexity of the system under test. It is known that the cost of testing activities may account for at least half of the development costs (Beizer, 1990; Myers et al., 2012). A typical approach followed to reduce this cost is the adoption of test automation (Berner et al., 2005; Rafi et al., 2012). This trend is also well founded for testing Web applications. Indeed, there exist state-of-the-art approaches like Crawljax (Mesbah et al., 2012) and DANTE (Biagiola et al., 2020) that enable fully automated Web application testing. These approaches mainly rely on tools that explore an application by crawling it. The crawling process results in a state transition model, which defines a set of possible interactions of the application with the user. This model is used as a test model (Malik et al., 2010; Weißleder and Lackner, 2010) for generating test cases (Mesbah et al., 2012).

Fully automated approaches consume too much time as a major drawback. The overall crawling and testing process can take multiple days to complete (Biagiola et al., 2020). One might find this acceptable since no manual effort is required. The process can be completed by a set of dedicated machines without consuming any human resources. However, the duration of the process becomes too long to be effective and it is not acceptable especially in a continuous development environment. The reason for long duration is the lack of insight and domain knowledge of crawling tools regarding the application under test. For the same reason, tools might require manual configuration for each application under test (Biagiola et al., 2020) although the crawling and test case generation process is automated. These configurations are performed to provide hints to the crawling process for exercising certain user interactions and/or avoiding some others.

At the other extreme, one might consider applying a fully manual approach, namely Exploratory Testing (ET) (Myers and Sandler, 2004; Hetzel and Hetzel, 1991). This approach is not based on a model or a set of previously defined test cases. Learning, test design and test execution activities are performed concurrently (Bach, 2003). These activities are solely based on human effort to utilize human intuition and domain knowledge. Hence, their success depends on the experience and skills of testers (Gebizli and Sozer, 2017). Nevertheless, ET is one of the mostly applied and one of the most successful approaches in the industry (Itkonen et al., 2007; Gebizli and Sozer, 2017). It was also shown to be effective in combination with model-based testing to improve the effectiveness of the testing process (Gebizli and Sözer, 2017).

We propose a semi-automated approach for Web application testing that combines ET with crawling based automated

[☆] Editor: Prof. Raffaela Mirandola.
[*] Correspondence to: Ozyegin University, Nişantepe Mah. Orman Sk., No: 34-36, Alemdağ – Çekmeköy 34794, Istanbul, Turkey.
E-mail addresses: nezih.sunman@ozu.edu.tr, nezih.sunman@siemens.com (N. Sunman), yigit.soydan@ozu.edu.tr (Y. Soydan), hasan.sozer@ozyegin.edu.tr (H. Sözer).

https://doi.org/10.1016/j.jss.2022.111441
0164-1212/© 2022 Elsevier Inc. All rights reserved.

testing. We introduce a tool named AWET (Automated Web application testing based on ET). It is developed as a plugin of Crawljax (Mesbah et al., 2012), which has been recognized as the state-of-the-art crawler for Web applications. AWET exploits a set of pre-recorded test cases. In principle, any pre-recorded test case can be used as input. These test cases can be scripted by developers or test engineers as well. However, this would require expertise or technical/programming knowledge at least. Our main goal in this work is to exploit the domain/application knowledge, experience, and intuition of people in improving the automated test generation process. These people can just explore a Web application, while their actions are being recorded and exported as test cases. ET (Itkonen et al., 2016) is characterized by a continuous learning and adaptation process, where an application is explored based on creativity, intuition, and experience, without relying on a set of pre-designed test cases. This type of testing can be used for obtaining effective test cases without requiring formal specifications or programming.

AWET analyzes the recorded test cases to infer rules and expected input values, while using the Web application. These rules include ordering constraints among the user events. For instance, a valid username and password must be entered before clicking a button for signing in. Such rules and the corresponding input values are stored in a database. Then, AWET employs Crawljax to crawl the Web application and generate test cases automatically during the crawling process. A plugin attached to the crawler ensures that the rules and input values recorded in the database are applied, whenever the corresponding Web elements are encountered. Hence, AWET exploits a set of recorded tests to steer the crawling process rather than following manually encoded rules or just randomly crawling the application. As a result, the process achieves high coverage rates very quickly.

We performed a case study with 5 Web applications. These applications were previously tested with state-of-the-art tools, Crawljax (Mesbah et al., 2012) and DANTE (Biagiola et al., 2020). The overall testing process takes from a few hours up to more than 2 days with these tools. AWET achieved higher coverage rates with respect to those of Crawljax. The coverage rates of DANTE was still higher but very close to that of AWET, where the difference ranges from 0.2% to 7.43%. On the other hand, AWET completed the process within minutes. Pre-recorded test cases that are used by AWET as input were created during manual testing activities that took 15 min. The coverage level achieved by these test cases was much lower. Hence, AWET utilized pre-recorded test cases and automated crawling together, while improving the effectiveness of both.

The remainder of this paper is organized as follows. In the following section, we summarize the related studies published in the literature and position our work with respect to these. In Section 3, we provide background information on tools developed for testing Web applications and for exploring their state space automatically. In Section 4, we explain our overall approach. In Section 5, we present an empirical evaluation of the approach and discuss the obtained results. Finally, in Section 6, we conclude the paper.

2. Related work

Crawljax (Mesbah et al., 2012) has been recognized as the state-of-the-art crawler for Web applications. It was shown to be superior with respect to other alternatives by an empirical study (Breton et al., 2014) and employed in recent studies (Biagiola et al., 2020) as well. We also employed an extended version of this tool as part of AWET. Crawljax automatically explores the state space of AJAX (Asynchronous Javascript and XML) Web applications. The Document Object Model (DOM) of a Web page is used for representing a particular state of the application. Crawljax roams on the application to discover new DOM states by providing random inputs to any input area and clicking randomly on any clickable element in any combination. Details of this process is explained in the following section.

A drawback of Crawljax is that the crawling process can keep visiting the same set of states without either terminating or attaining any progress in exploration if the tool is not configured or interrupted after a time threshold. Moreover, naive exploration strategies such as depth-first, breadth-first and random exploration can concentrate on a limited set of features and lead to low test coverage. As such, there have been several improvements considered for its exploration strategy. For instance, FEEDEx (Fard and Mesbah, 2013) aims at maximizing diversity of states during the crawling process. There are many other strategies proposed (Benjamin et al., 2011; Dincturk et al., 2012; Choudhary et al., 2012, 2013) but neither of these exploits test cases recorded during ET activities. Testilizer (Fard et al., 2014) aims at exploiting the human knowledge existing in manually-written test cases. However, it does not infer rules that guide the crawling process as AWET does. Instead, it infers a state transition model from these test cases as a starting point. Then it uses a crawler to expand this model further by exploring uncovered paths and states. AWET can also take manually-written test cases as input. However, our goal is to exploit test cases that can be recorded without requiring any expertise or technical/programming knowledge.

The state transition model that is obtained during the crawling process can be used as a test model for generating test cases. There are also techniques and tools proposed for improving the effectiveness of this process. Recently, DANTE (Dependency-Aware Crawling-Based Web Test Generator) (Biagiola et al., 2020) was proposed for eliminating redundant test cases and avoiding infeasible test paths to prevent test breakages (Stocco et al., 2018) by analyzing dependencies among the generated test cases. However, it has to be configured for every Web application. So, although the process is fully automated, a successful execution actually requires manual effort as well as expertise. We discuss this issue in detail in Section 5. AWET also requires manual effort but it does not require expertise. Test cases and inputs that are recorded during ET activities are used as input by AWET to configure Crawljax automatically. ET can be performed by testers, who do not have any programming knowledge or a degree in computer science. DANTE performs an offline analysis after test cases are generated. Therefore, it can be used together with AWET as a complementary tool, in principle. AWET refines the set of generated test cases. DANTE tries to improve the quality of test cases after they are generated. We argue that it is more effective to improve the quality of the crawling process rather than improving the generated test cases after crawling. We show that we can converge to the same result by just exploiting a set of previously recorded test cases that are created in less amount of time and without relying on any technical/programming knowledge.

Test input generation is another problem to be addressed in automated Web application testing. There have been recent attempts (Biagiola et al., 2017) to attack this problem by employing search based optimization techniques like genetic algorithms. However, the success of such fully automated tools will always be limited without any knowledge about the application under test. Consider an authentication page, where the user has to provide a correct username and a password. These inputs must be manually provided in any case. There are also prerequisite relations among the user events. For instance, username and password fields should be populated before clicking a login button. A crawler may eventually figure out how to proceed further by trying various event permutations. However, the process would be much faster with an insight of a human being who knows what these fields and buttons are about and how they are related to each other. Rather than configuring a crawler for each application by encoding a set of rules, we propose a generically applicable approach, where the crawler is automatically configured based on the analysis of a set of recorded test cases.

ET was previously shown to be effective in combination with model-based testing to improve the effectiveness of the testing process (Gebizli and Sözer, 2017). That study was performed in the consumer electronics domain and the goal of the study was to extend and improve a manually created test model by comparing possible paths on the model with respect to test paths recorded during ET activities. In this work, we focus on Web applications. Our test models are automatically created. Recorded test sequences are used during the model generation process to steer the process dynamically rather than improving the model after it is generated. We employed a set of existing tools and extended them to implement our approach. We provide background information on these tools in the following section.

3. Tools for automating web application testing

Web applications have become increasingly more powerful and sophisticated. In particular, the use of client-side scripting has become predominant by the adoption of new technologies like AJAX. These trends posed a challenge on testing these applications and several tools have been proposed to automate the testing process.

One of the most popular tools currently being used for Web application testing is Selenium IDE (Integrated Development Environment),[1] which is primarily a capture-and-replay tool. It has two basic types of components to facilitate the capture-and-replay functionality. First, it provides *plugins* for various types of browsers such as Chromium and Firefox. These plugins have a GUI (Graphical User Interface) for easily recording the interactions of the user with a Web application. These recorded interactions can be exported in the form of reusable, executable test cases in various programming languages such as Java and Python. Listing 1 shows an example code snippet regarding an automatically generated test case in Java. Second, it provides *drivers* for automatically executing test cases that are either manually developed or automatically generated based on the records obtained from plugins. The test case in Listing 1, for instance, imports the Chrome driver (Line 2), which is initialized (Line 7) before the test case execution. Driver can be used for reaching to a Web page (Line 15), locating elements on the retrieved page (a text box is found in Lines 16–17 with its *name* attribute) and simulating user interaction (key press events are triggered in Lines 16–17).

Tools like Selenium IDE are capable of automating test execution. However they require effort for either developing the tests or performing manual tests so that they can be recorded to be replayed later on. Additional effort might be necessary in case of GUI modifications as well. Therefore, tools like Crawljax (Mesbah et al., 2012) have been introduced to facilitate fully automated Web application testing.

Crawljax automatically crawls the GUI of an AJAX-based Web application. It incrementally creates a state flow graph (SFG) of the application during the crawling process. It can test for a set of invariants and generate test cases in the meantime. Basic steps followed by this tool are outlined in Algorithm 1. The set of states and the SFG are initialized with the root state (Lines 1–2). New states are created and added as the application is crawled and state changes are detected. Hereby, each unique DOM instance (Yandrapally et al., 2020) represents a state and

Algorithm 1 Basic steps followed by *Crawljax*.

```
1:  $s_i \leftarrow$ initial state; $S \leftarrow \{s_i\}$;
2:  $SFG \leftarrow$ initialize state flow graph with $S$;
3:  while ($\neg$ timeout) and (null $\neq$ ($s \leftarrow$ a state yet to be explored in $S$)) do
4:    $E \leftarrow$ all clickable elements in $s$
5:    for each $e \in E$ do
6:      enter form values in $s$;
7:      click element $e$;
8:      $s_c \leftarrow$ current state;
9:      if $s_c \neq s$ then
10:       $S \leftarrow S \cup \{s_c\}$; update $SFG$ with $s_c$ and additional edges;
11:       run plugins;
12:     end if
13:   end for
14: end while
```

events that lead to these states represent transitions among the states. The crawling process continues as long as a predefined timeout duration is not exceeded and there exist states yet to be explored (Lines 3–14). A state yet to be explored is selected at a time (Line 3). Crawljax collects all the clickable elements residing in the corresponding DOM structure (Line 4). It automatically enters values for the forms in this structure and triggers a click event for each of the clickable elements (Lines 6–7). Then it retrieves the current state and checks if it is different than the explored state (Lines 8–9). State changes are detected based on the Levenshtein edit distance between the DOM structures. The new state is compared with all the existing states and it is added to the SFG if it is not among the already visited states. In addition, a new edge is created on SFG between the previous state and the current state (Line 10). Crawljax can be extended with plugins (Line 11). It embodies a plugin architecture[2] that facilitates such extensions. This architecture allows the incorporation of plugins that act as hooks or event listeners attached to the automated crawling process for monitoring and/or steering the process. It provides the corresponding interfaces as extension points for plugins that can be triggered not only after state changes but at various phases of the crawling process such as before or after state changes, user events and invariant violations. AWET exploits this extension mechanism as well.

ATUSA (Automatically Testing UI States) (Mesbah et al., 2012) was previously proposed as a set of plugin hooks attached to Crawljax for testing AJAX-based Web applications. These plugins check for a set of invariants during the crawling process. The set of invariants include generic ones such as the absence of error messages (e.g., 500 Internal Server Error) and dead links. They can be extended with application-specific invariants. ATUSA can also generate test cases offline based on the constructed SFG. It collects all the sink states in the SFG and computes the shortest path from the initial state to each of these states. Test cases can be generated online, during the crawling process as well. Crawljax can output test paths in the form of test cases during the crawling process. It segments these paths whenever a sink state is reached (i.e., no more clickable elements exist in the explored state or no new states are discovered). The crawling process continues from an unvisited state or the initial state. AWET follows this online approach for test case generation, which is adopted by DANTE (Biagiola et al., 2020) as well. A plugin attached to Crawljax outputs test paths in the form of Java test cases (using Selenium Web driver) during the crawling process.

[1] https://www.selenium.dev/selenium-ide/

[2] https://github.com/crawljax/crawljax/wiki/Writing-a-plugin

Listing 1: A code snippet regarding a sample test case that is automatically generated by Selenium IDE (Some of the source code lines are removed for brevity).

```java
import org.junit.Test;
import org.openqa.selenium.chrome.ChromeDriver;
public class SampleTest {
 private WebDriver driver;
 @Before
 public void setUp() {
  driver = new ChromeDriver();
 }
 @After
 public void tearDown() {
  driver.quit();
 }
 @Test
 public void sampletest() {
  driver.get("https://www.google.com/");
  driver.findElement(By.name("q")).sendKeys(Keys.DOWN);
  driver.findElement(By.name("q")).sendKeys(Keys.ENTER);
 }
}
```

Listing 2: A sample code snippet regarding a custom configuration in Crawljax

```java
1 CrawljaxConfigurationBuilder builder = CrawljaxConfiguration.builderFor(...);
2 InputSpecification input = new InputSpecification();
3 Form aForm = new Form();
4 aForm.field("name").setValues("Alice", "Bob", "Carol");
5 input.setValuesInForm(aForm).beforeClickElement("button").withText("Save");
6 builder.crawlRules().setInputSpec(input);
```

Crawljax and ATUSA are subject to a number of issues. First, some of the generated test cases can be infeasible due to application-specific constraints and input values (e.g., a particular input must be entered in a form before clicking on a button). Second, test cases may be dependent on each other due to their impact on the application state. These issues lead to test breakages (Stocco et al., 2018). Third, some of the test cases may cover the same paths and as such turn out to be redundant. DANTE (Biagiola et al., 2020) aims at addressing the second and third issues. It analyzes the generated test cases and their inter-dependencies to eliminate redundancies and to avoid infeasible test paths. In this work, we aim at addressing the first issue, which is a generic problem that is not addressed by Crawljax, ATUSA or DANTE. All these tools have to be manually configured for every Web application. Listing 2 shows a sample code snippet for adding custom crawling rules regarding specific elements.

Classes named *CrawljaxConfigurationBuilder*, *InputSpecification* and *Form* in Listing 2 (Lines 1–3) are all provided as part of Crawljax for specifying customized crawling rules. In this example, the set of inputs to be provided for a form are specified (Line 4). It is also specified that these inputs have to be provided before a particular button is clicked (Line 5). Such rules for locating elements and interacting with them as well as the relevant input values actually steer the crawler towards the interesting regions of the Web application. However, they are hard-coded. They are also application-specific and as such, they should be manually implemented for each application. In the following, we describe our approach that automatically extracts these rules and configures the crawler by analyzing a set of pre-recorded test cases.

4. AWET tool and the overall approach

The overall process is depicted in Fig. 1. Hereby, ET is performed first. A set of test cases are recorded during ET. These test cases are automatically analyzed to save the performed events and utilized inputs in a database. Crawling rules and strategies are also inferred and stored in this database during *Test Case Analysis*. Then, crawling of the Web application is performed. Test cases are generated during crawling, which is guided by the rules and strategies stored in the database.

The process is supported by a set of integrated tools as part of AWET, which is available at a public repository.[3] One of these tools is Selenium IDE, which is used as a black-box tool for two purposes. First, the browser plugin of Selenium is used at the beginning of the process for recording the sequences of events and inputs during ET, while the tester interacts with the application. Second, the *Selenium Driver* is used for automatically executing the generated test cases. Test cases are recorded and generated in the form of executable JUnit (Massol, 2003) tests. Another external tool employed as part of AWET is Crawljax. However, this tool is not used as a black-box tool. It is extended and tailored for our approach with a plugin, which can apply a set of rules and actions during the crawling process such as filling a form before a click event. These rules and extensions can also be provided at the beginning of the crawling process as custom configurations (See Listing 2). *Guided Crawling and Test Case Generation* depicted in Fig. 1 is performed with our extended version of the Crawljax tool (*Extended Crawljax*), where the necessary plugins and configurations are incorporated to be able to execute the rules and strategies automatically generated during *Test Case Analysis*.

Test Case Analysis is performed automatically by our tool, *Test Case Analyzer*, which uses a relational database to store information related to the recorded test cases. All the parameters recorded via the Selenium browser plugin are saved in this database. A snippet from the records extracted from the recorded

[3] https://github.com/nezihsunman/AWET

test cases are presented in Listing 3. The snippet corresponds to 3 commands performed, for each of which the set of properties are listed starting with an *id* at lines 3, 6 and 15 (The first *id* listed at the first line corresponds to the overall test) . The last two commands are associated with a *click* event (Lines 7 and 16). There might be multiple ways to locate a Web element in a Web page. The list of possible locators are provided as *targets*, e.g., Lines 9–13 and 18–22. AWET mainly utilizes *xPath* attributes of type *idRelative* (Lines 11 and 20) as explained later in this section. We employed a database to store these records mainly for efficient storage and retrieval of data. This solution is easier to scale for larger systems under test. In addition, the database is loosely coupled with the tool. It might even be deployed in the cloud, if necessary.

Listing 3: A snippet from the records extracted during a test session.

```
 1  "id": "06e61294-c46d-468f-b2d3-0e9c7da4643b",
 2  "name": "splittype_test_1", "commands": [{
 3  "id": "348c2558-41f3-4c88-95f3-8be2e63c584c",
 4  "comment": "", "command": "open", "target": "/",
        "targets": [],
 5  "value": ""}, {
 6  "id": "883f02ad-5303-44a7-9ffd-d4f1ea11168e",
 7  "comment": "", "command": "click",
 8  "target": "css=.popover-navigation > .btn",
 9  "targets": [
10  ["css=.popover-navigation > .btn", "css:finder"],
11  ["xpath=//div[@id='step-0']/div[3]/button",
        "xpath:idRelative"],
12  ["xpath=//div[3]/button", "xpath:position"],
13  ["xpath=//button[contains(.,'End tour')]",
        "xpath:innerText"]
14  ], "value": "" }, {
15  "id": "9a433ba9-6aab-49e1-887a-63f472b8d476",
16  "comment": "", "command": "click",
17  "target": "css=#add_wallet_button > span",
18  "targets": [
19  ["css=#add_wallet_button > span", "css:finder"],
20  ["xpath=//a[@id='add_wallet_button']/span",
        "xpath:idRelative"],
21  ["xpath=//div/a/span", "xpath:position"],
22  ["xpath=//span[contains(.,'Add')]",
        "xpath:innerText"]
23  ], "value": "" }
```

We used a simple database that comprises two tables. The first table stores click events, whereas the second one stores events for providing inputs to input areas like text fields. Table 1 shows a snippet from the database table that stores events for providing inputs to input areas like text fields. The first column represents the columns/fields of the table, whereas the second column shows an example record saved in the table. The first two fields are used for locating the form that is associated with a button. The remaining fields keep data provided by the user to fill the form. This example record partially represents the configuration rule coded in Listing 2, where the *name* form should be filled with value "Alice" before clicking the *save* button. There exists a separate record for each possible input value (e.g., "Bob", "Carol").

These tables also include information regarding the stored events such as the corresponding test case number and the identification of user interface elements. If the type of event is to provide input in a form, for instance, the record includes the HTML ID of the form as well as the input text entered by the user.

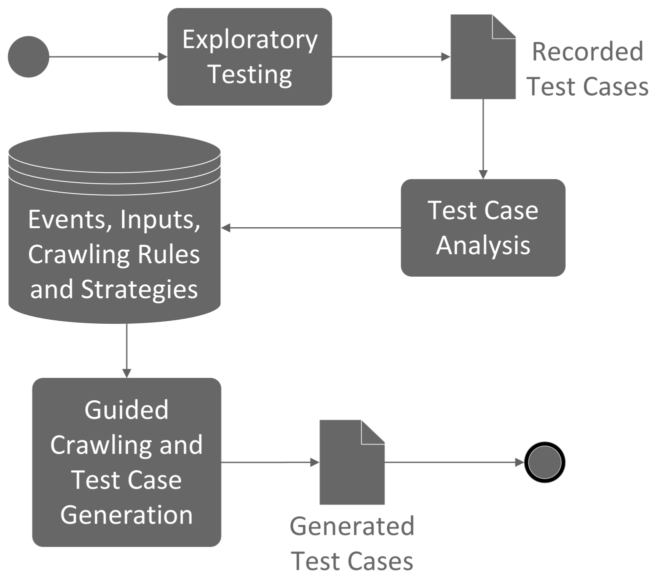
Exploratory Testing
Recorded Test Cases
Test Case Analysis
Events, Inputs, Crawling Rules and Strategies
Guided Crawling and Test Case Generation
Generated Test Cases

Fig. 1. The overall process followed with AWET depicted in the form of Business Process Model and Notation (BPMN).

Table 1
A snippet from the database table that stores events for providing inputs to input areas like text fields.

FormLocation	//option[. = 'SaveButtonPath?']
FormClickType	underxpath
TestName	Petclinictest_Test_1.java
Tagname	button
LocationID	name
UserValue	Alice
UserClickType	By.id

Other types of events like clicking a button or submitting a form are kept as *idRelativeXPath*, which includes the id attribute in the XPath (Leotta et al., 2014). For example, the following shows the content of a record regarding a single event.

```
xpath=//form[@id='add-owner-form']/div[6]/button
(tagname = "body", id = "input_password")
```

Hereby, the XPath query locates a *form* element that has "add-owner-form" as the *id*, the 6th *div* element within this form and the *button* in this *div* element. The *tagname* and the *id* of the button element are also recorded as "body" and "input_password" respectively. There are also alternative *locating mechanisms* other than XPath, which can be used for locating a particular element on a Web page. AWET currently supports the basic set of mechanisms that are supported by Selenium. These mechanisms enable locating an element by *id*, *tagname*, or even by *text*, where the element is selected if its content matches a specified regular expression (specified with *Selectors.WithText*).

The populated database does not only store the set of events and the necessary information for triggering these events, but also ordering dependencies among these events such as the set of fields that has to be populated before a click event. Algorithm 2 outlines the basic steps followed by *Test Case Analyzer*, while analyzing the test cases and populating the database.

Test Case Analyzer directly records click events on the database together with all the necessary information to replay the event. It stores all the other types of events (e.g., *sendKey*) in a queue, whenever they are encountered (Line 11). All the events in this queue are dequeued and stored in the database whenever a *click*

Listing 4: A code snippet from the plugin used for extending Crawljax.

```
1  public CrawljaxConfiguration.CrawljaxConfigurationBuilder getCrawljaxConfig() {
2   InputSpecification inputSpecification = new InputSpecification();
3   Map<String, Collection<List<ID>>> uniqueList = idService.getIdWithHaspMap(1, ApplicationNames.getName());
4   for(Map.Entry<String,Collection<List<ID>>> entry:uniqueList.entrySet()){
5    Form bForm = new Form();
6    inputSpecification.setValuesInForm(bForm). beforeClickElement("button").underXPath(xpath);
7   }
8   builder.crawlRules().setInputSpec(inputSpecification);
9   builder.crawlRules().setDisableIdAndNameIdentification(true);
10  builder.setHandleSameFormInputsOncePerState(true);
11  return builder;
12 }
```

Algorithm 2 Basic steps followed by *Test Case Analyzer*.

```
1:  $DB$ ← database connected; $q$ ← an event Queue is created;
2:  for each $t \in$ Test cases do
3:    for each event $e \in$ t do
4:      if $e$ is of type Click then
5:        while $q$ is not empty do
6:          $e_p$ ← dequeue an event from $q$;
7:          add $e_p$ to $DB$ as a prerequisite event for $e$;
8:        end while
9:        add $e$ to $DB$ as a click event;
10:     else
11:       enqueue $e$ to $q$;
12:     end if
13:   end for
14: end for
```

event is encountered (Lines 4-9). They are stored as the list of prerequisite events for that click event (Line 7). So a list of events are associated with the first click event ahead in the event sequence.

Extended Crawljax uses the database that is populated by *Test Case Analyzer*. It connects to this database at the beginning of the crawling process and automatically creates a configuration. It takes and applies the set of rules and input values from the database. They are not hard-coded. A code snippet from the plugin used for extending Crawljax is presented in Listing 4. Hereby a method called *getCrawljaxConfig* is overwritten to incorporate a custom configuration for crawling. The object *idService* is used for connecting to the database and retrieving the list of Web elements (Line 3). Each type of element is handled separately. The code snippet in Listing 4 shows the handling of a button element (Line 6), for instance, where a set of form elements have to be provided with certain input values before a *click* action is performed.

Hence, AWET is a generic tool that does not require expertise for configuring the crawler. It just requires a set of recorded test cases regarding the application under test. The configuration is automatically performed based on locating mechanisms, input values and event ordering dependencies that are retrieved from the database. Note that the set of test cases used for configuration can be extended in time with newly recorded test cases, generated test cases or even manually scripted test cases, if exist. By this way, *Test Analysis* can be performed on the extended set to learn new crawling strategies and rules, including those discovered during the previous crawling session. However, reusing existing test cases in the context of web application evolution is not trivial and further research is necessary to resolve issues that may arise due to conflicts and inconsistencies. We present an empirical evaluation of our approach in the following section.

5. Evaluation

In this section, we present an empirical evaluation of our approach performed with 5 experimental objects and 5 subjects. Our goal is to investigate the effectiveness of AWET in terms of the coverage rate achieved and the overall duration of the process. We are also interested in the impact of the ET phase on results. We define the following research questions based on these concerns:

- *RQ1*: What is the coverage rate of tests that are generated by AWET and how does it compare with respect to those generated by fully automated approaches?
- *RQ2*: What is the overhead of AWET both in terms of manual effort and computation time and how does it compare with the fully automated approaches?
- *RQ3*: How the coverage rate of tests that are generated by AWET is influenced by the duration of ET and the tester profile?

In the following, we first introduce our experimental setup. Next, we present and discuss the obtained results. We conclude the section with a discussion of limitations regarding our approach and AWET as well as threats to the validity of our evaluation.

5.1. Experimental setup

We used 5 Web applications as experimental objects, namely *dimeshift* (Dimeshift, 2021), *petclinic* (PetClinic, 2021), *phoenix* (Phoenix, 2021), *retroboard* (RetroBoard, 2021), *splittypie* (Splittypie, 2021). These applications are open source single-page Web applications. Each of them includes more than a thousand lines of client side (JavaScript) code. They were previously used as a benchmark set of applications in previous studies (Biagiola et al., 2019, 2020).

ET is performed with 5 participants, who volunteered to take part in our study. The profile of these participants are listed in Table 2. We can see that all the participants are either undergraduate or graduate students with 5 years of programming experience at most. They do not have any software development or testing experience in the industry except the last one (*P5*) listed in Table 2. Two of them do not even have a computer science background. The first 4 participants did not also have any prior knowledge regarding the applications that are to be tested. They were also not involved in any training. Participants were just provided with the Web application, and they were asked to explore the page within the limited time. We first performed a case study with *P5*, who has (one year) experience in testing Web applications. Then, we performed another study with the remaining 4 participants for a shorter duration of ET.

Table 2
The profile of subjects who performed ET.

#	Background	Education level	Programming experience
P1	Industrial Engineering	Graduate	5 years
P2	Mechanical Engineering	Graduate	4 years
P3	Computer Science	Undergraduate	2 years
P4	Computer Science	Undergraduate	2 years
P5	Computer Science	Graduate	4 years

Table 3
Comparison of AWET coverage rates (%) with respect to ET and the state-of-the-art tools for the experimental objects (EO).

Tool/EO	*dimeshift*	*petclinic*	*phoenix*	*retroboard*	*splittypie*	Mean
ET	15.40	10.50	12.40	17.22	6.33	12.37
AWET	41.87	27.32	46.50	39.51	21.08	35.26
Crawljax	26.53	26.04	36.21	37.93	15.18	28.38
DANTE	**42.07**	**28.22**	**53.93**	**40.50**	**24.32**	**37.81**

We compared the effectiveness of AWET with respect to those of Crawljax and DANTE. We used the Crawljax configuration and DANTE implementation that are available online.[4] We used the same Crawljax configuration that was previously used for the evaluation of DANTE (Biagiola et al., 2020). The default exploration strategy and state abstraction function were employed in this configuration. In addition, we used the same set of experimental objects that are used for the evaluation of DANTE. We reproduced the experimental setup used in that evaluation. The comparison among the tools is made with respect to test coverage and test duration for answering our research questions. We defined a timeout value (30 min) as the stopping criteria for the crawling process of AWET. This is the same timeout value that is used by DANTE as the stopping criterion. We did not use this stopping criterion for the original Crawljax. It was applied with the default configuration. The ratio of executed lines of the JavaScript code to the total number of lines is calculated for measuring test coverage. DANTE includes a module that can measure coverage, while test cases are executed on the Web application. We adopted the same module to measure coverage achieved by AWET. The recorded test cases as well as the docker images are available at the AWET repository[5] for the reproducibility of the results. 22 tests and 589 click operations are recorded in total during a 15 min of ET activity performed by the 5th participant. These are used by AWET as input for generating test cases that are executed with the *Selenium Driver*. Results are shared and discussed in the following section, in alignment with the research questions.

5.2. Results and discussion

5.2.1. Coverage rate of the generated tests (RQ1)

Table 3 lists results regarding test coverage results used for answering *RQ1*. The first row (ET) of Table 3 lists the coverage achieved as a result of the execution of these test cases for the 5 tested Web applications. The second row lists the coverage achieved by AWET. The third row lists the coverage achieved by Crawljax. The last row lists the coverage achieved by DANTE.

We can see from Table 3 that the coverage rates achieved by AWET are much higher than those achieved with Crawljax. This improvement was possible by only providing a set of test cases recorded during 15 min of ET. It is also interesting to observe that the coverage of these test cases are low (See the first row of Table 3). AWET uses these test cases to guide Crawljax and

[4] https://github.com/matteobiagiola/ICST20-submission-material-DANTE
[5] https://github.com/nezihsunman/AWET

Table 4
Comparison of AWET crawling and test execution time (in minutes) with respect to those of Crawljax and DANTE for the experimental objects (EO).

Tool/EO	*dimeshift*	*petclinic*	*phoenix*	*retroboard*	*splittypie*
AWET	7.6	2.7	4.3	1.2	0.9
ET + AWET	22.6	17.7	19.3	16.2	15.9
Crawljax	> 2880	> 2880	> 2880	> 2880	> 2880
DANTE	34.8	35.9	39.0	34.4	33.3

it turns out to be more effective with respect to both using Crawljax without guidance and using only the recorded test cases without crawling. On the other hand, we observe that coverage rates achieved by DANTE are still higher than those of AWET although they are very close. The largest difference is observed for *phoenix* with 7.43%. The smallest difference is observed as 0.2% for *dimeshift*, which is the largest application among the 5 experimental objects.

5.2.2. Required manual effort and computation time (RQ2)

Although DANTE performs better than AWET in terms of coverage rates, it has two drawbacks in other aspects. First, it requires the manual implementation of hard-coded set of rules like those in Listing 2. Therefore, it requires effort, programming experience and expertise regarding the modular extension mechanisms and the Application Programming Interface (API) of the crawler. The second drawback is its runtime performance.

Table 4 lists the runtime measurements for 5 Web applications. The first row lists the crawling and test execution time for AWET. The second row includes the time for ET (15 min) as well. The third and the fourth rows list the crawling and test execution times for Crawljax and DANTE, respectively.

We can see in Table 4 that AWET completed the testing process within 10 min. The duration for even the largest application, *dimeshift* is 7.6 min. Crawljax was forced to stop after 2 days of execution for all the applications. The second row lists the total time spent for AWET, including ET that took 15 min. We observe that investing 15 min of manual effort at the beginning leads to hours of reduction in the subsequent phases. The other tools do not require such an initial investment except DANTE, which relies on an initial manual effort required for application-specific configurations. This effort would depend on the amount of configuration. At one extreme, one can just use Crawljax with random exploration without any configuration. In this case, the exploration might get stuck at many pages and the coverage of the generated test cases would be limited. At the other extreme, one can implement a detailed configuration by covering all the Web elements in all the pages, their inputs and interactions to create an extensive amount of test cases. Hence, there is a trade-off between the amount of configuration and the coverage of the crawling process. Moreover, the effort would also change depending on the expertise and knowledge of the developer/tester, who implements the configuration. The knowledge and expertise regarding the application under test, Web applications in general and the Crawljax API would directly impact the effort. Therefore, it is not possible to quantify the effort in general and unfortunately, the effort spent in configuring DANTE for the 5 experimental objects is not recorded/reported. However, there are some indirect measures and factors that impact the effort. One of the most important factors is the number of Web elements in the application that can be subject to user interaction. One must traverse all the pages in the application to determine relevant inputs for each action as well as interactions among these actions such as ordering constraints. On the other hand, lines of code written for configuration can be considered as an indirect measure although it is not a reliable measure by itself to measure

Table 5
The number of Web elements (WE) in each experimental object (EO) and the number of lines of code (LOC) written for configuring the crawler for DANTE Biagiola et al. (2020).

Count/EO	*dimeshift*	*petclinic*	*phoenix*	*retroboard*	*splittypie*
WE	37	51	31	14	21
LOC	253	231	201	150	139

Table 6
Coverage rates (%) achieved when AWET only used test cases recorded during the first 5 min of ET performed by the participants for the experimental objects (EO).

Participant/EO	*dimeshift*	*petclinic*	*phoenix*	*retroboard*	*splittypie*
P1	34.15	11.23	38.11	28.75	9.20
P2	35.69	15.10	41.67	35.53	9.17
P3	17.38	11.24	23.20	20.13	18.15
P4	16.48	11.65	14.41	20.18	11.01
P5	**38.15**	**26.54**	**43.17**	**38.25**	**19.25**

effort (Fenton and Bieman, 2015). We calculated and reported these measurements in Table 5.

The first and the second rows of Table 5 lists the number of Web elements and the number of lines of configuration code written for employing DANTE for the 5 experimental objects, respectively. We believe that it is not possible to estimate effort in a generic and absolute sense. It can highly vary based on the types of elements as well as the amount of dependencies among these elements. For instance, we do not observe a correlation between the number of Web elements and the number of lines of configuration code written for the 5 experimental objects (See Table 5). We expect that the effort of an extensive configuration that covers input values and possible interactions for all the Web elements would be comparable to that of manual testing.

In principle, AWET and DANTE are complementary as mentioned in Section 2. AWET improves the crawling process, whereas DANTE filters test cases after crawling. The overall cost of AWET involves *(i)* ET, *(ii)* automated analysis of recorded test cases (negligible), and *(iii)* crawling and test generation. The overall cost of DANTE involves *(i)* manual configuration, *(ii)* crawling and test generation, and *(iii)* post-processing the generated tests. Unfortunately, we do not know the exact cost of manual configuration. However, indirect measurements suggest that its cost is higher than ET. Second, it requires skills. Third, AWET requires less time overall even if we ignore that step.

5.2.3. Impact of ET duration and the tester profile (RQ3)

Effectiveness of AWET is dependent on the set of test cases provided as input and as such, the effectiveness of ET, which is the motivation for *RQ3*. The effectiveness of ET is, in turn, dependent on the tester and the duration of the activity. To be able to analyze the impact of these factors further, we repeated the process for a shorter duration of ET.

Table 6 lists the coverage results achieved by AWET based on test cases recorded during 5 min of ET instead of 15 min. These results facilitate a better understanding of the trade-off between the manual effort and the test effectiveness and as such, partially answering *RQ2* to evaluate overhead of AWET in terms of manual effort. The automated testing process also took less than 5 min. So, the overall process including ET and automated testing took less than 10 min. Yet, we can see that coverage becomes better than Crawljax especially for large applications. Another interesting observation is that ET performed by *P1* and *P2* provided better input than *P3* and *P4* for the process. These are the participants, who did not have Computer Science background as opposed to the other two.

Overall, we observe regarding *RQ3* that the impact of ET on the effectiveness of AWET is not significantly influenced by the technical background of participants. We see that the duration of ET has some impact on the coverage achieved by AWET, when we compare the values listed in the second row of Table 3 (obtained with 15 min of ET by *P5*) with the last row of Table 6 (obtained with 5 min of ET by *P5*). However, this impact is also not very significant considering the fact the duration is reduced from 15 min down to 5 min. Testing experience turns out to be most important factor that influences the effectiveness of ET and also the results obtained by AWET. The number of participants is not enough for generalizing this outcome but previous empirical findings (Gebizli and Sozer, 2017) support this observation. We observed that the first two participants were motivated more for the study, contended to cover the tested applications as much as they could. We attribute the difference in their performance to this behavior. The best results are obtained with the input provided by the ET activities of *P5*, who was the only participant with prior knowledge and experience regarding the Web applications used as experimental objects. We discuss limitations of AWET and validity threats for our evaluation in the following section.

5.3. Limitations and threats to validity

AWET currently supports only *clicking* and *form filling* as events that can be performed by the user. *Hover* and *focus in/out* events, for instance, are not supported. However, the tool can be extended to incorporate these as well.

We assume that the DOM structure conforms to HTML5 or above. We also assume that this structure will remain the same for the application under test. Otherwise, the recorded test cases can turn out to be obsolete and crawling process should be repeated.

Our evaluation is subject to an external validity threat since it is based on 5 experimental objects. These objects were selected for a fair comparison with the state-of-the-art as they were previously used as a benchmark. The number of participants in our study is also limited. However, our results were promising for at most 15 min of ET. Moreover, all the participants were either undergraduate or recently graduated students. This is actually a drawback for our approach. AWET might perform even better if its input is generated as a result of ET activities performed by experienced testers, who also have deep knowledge regarding the systems under test. We expect that these testers would explore the most interesting (hidden, hard-to-reach and error-prone) parts of the application. Test cases recorded during this exploration would also steer the crawling more effectively towards better coverage and error detection. Our results support this expectation; however, the number of participants in our study is not high enough for proving the validity of this expectation in general. Moreover, all the participants except one of them (P5) have used the 5 Web applications as experimental objects for the first time for testing them.

6. Conclusion

We introduced an approach and a tool, AWET for Web application testing, which combines ET with crawling based automated testing. AWET exploits a set of test cases that are recorded during ET activities to automatically configure a crawler. Our case study with 5 Web applications showed that data recorded during an ET activity of 15 min can reduce the duration of the automated testing process by hours for achieving similar coverage rates. Moreover, there is no need for configuring a tool for every application under test. The insight and experience of human testers can be transferred to an automated testing tool without requiring any expertise that is necessary for performing manual configurations. Test activities in our case study were carried out by undergraduate or recently graduated students. The performance of AWET might have been much better if ET activities were conducted by experienced test engineers or those who had knowledge on the domain and the application under test.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Bach, J., 2003. Exploratory Testing Explained. Satisfice, Inc., URL http://www.satisfice.com/articles/et-article.pdf.

Beizer, B., 1990. Software Testing Techniques, second ed Van Nostrand Reinhold Co., New York, NY, USA.

Benjamin, K., von Bochmann, G., Dincturk, M.E., Jourdan, G.V., Onut, I.V., 2011. A strategy for efficient crawling of rich internet applications. In: Proceedings of the International Conference on Web Engineering. pp. 74–89.

Berner, S., Weber, R., Keller, R.K., 2005. Observations and lessons learned from automated testing. In: Proceedings of the 27th International Conference on Software Engineering. pp. 571–579.

Biagiola, M., Ricca, F., Tonella, P., 2017. Search based path and input data generation for Web application testing. In: Proceedings of the Search Based Software Engineering Symposium. pp. 18–32.

Biagiola, M., Stocco, A., Ricca, F., Tonella, P., 2019. Diversity-based Web test generation. In: Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 142–153.

Biagiola, M., Stocco, A., Ricca, F., Tonella, P., 2020. Dependency-aware web test generation. In: Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification. pp. 175–185.

Breton, G.L., Bergeron, N., Halle, S., 2014. A reference framework for the automated exploration of Web applications. In: Proceedings of the 19th International Conference on Engineering of Complex Computer Systems. pp. 81–90.

Choudhary, S., Dincturk, M.E., Mirtaheri, S.M., Jourdan, G.V., von Bochmann, G., Onut, I.V., 2013. Building rich internet applications models: Example of a better strategy. In: Proceedings of the International Conference on Web Engineering. pp. 291–305.

Choudhary, S.R., Prasad, M., Orso, A., 2012. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in Web applications. In: Proceedings of the International Conference on Software Testing, Verification and Validation. pp. 171–180.

Dimeshift, 2021. Dimeshift: Easiest way to track your expenses. https://github.com/jeka-kiselyov/dimeshift.

Dincturk, M.E., Choudhary, S., von Bochmann, G., Jourdan, G.-V., Onut, I.V., 2012. A statistical approach for efficient crawling of rich Internet applications. In: Proceedings of the International Conference on Web Engineering. pp. 362–369.

Fard, A.M., Mesbah, A., 2013. Feedback-directed exploration of Web applications to derive test models. In: Proceedings of the 24th International Symposium on Software Reliability Engineering, ISSRE. pp. 278–287.

Fard, A.M., Mirzaaghaei, M., Mesbah, A., 2014. Leveraging existing tests in automated test generation for Web applications. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 67–78.

Fenton, N., Bieman, J., 2015. Software Metrics: A Rigorous and Practical Approach, third ed. Taylor and Francis Group.

Gebizli, C.S., Sozer, H., 2017. Impact of education and experience level on the effectiveness of exploratory testing: An industrial case study. In: Proceedings of the 12th Workshop on Testing: Academic and Industrial Conference – Practice and Research Techniques. pp. 23–28. Tokyo, Japan.

Gebizli, C.S., Sözer, H., 2017. Automated refinement of models for model-based testing using exploratory testing. Softw. Qual. J. 25 (3), 979–1005.

Hetzel, W.C., Hetzel, B., 1991. The Complete Guide to Software Testing, second ed. John Wiley & Sons, Inc., New York, NY, USA.

Itkonen, J., Mantyla, M.V., Lassenius, C., 2007. Defect detection efficiency: Test case based vs. Exploratory testing. In: First International Symposium on Empirical Software Engineering and Measurement. IEEE Computer Society, pp. 61–70.

Itkonen, J., Mäntylä, M.V., Lassenius, C., 2016. Test better by exploring: Harnessing human skills and knowledge. IEEE Softw. 33 (4), 90–96.

Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2014. Reducing Web test cases aging by means of robust XPath locators. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops. pp. 449–454.

Malik, Q.A., Jääskeläinen, A., Virtanen, H., Katara, M., Abbors, F., Truscan, D., Lilius, J., 2010. Model-based testing using system vs. test models — what is the difference?. In: Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems. pp. 291–299.

Massol, V., 2003. JUnit in Action, first ed. Pearson.

Mesbah, A., van Deursen, A., Roest, D., 2012. Invariant-based automatic testing of modern Web applications. IEEE Trans. Softw. Eng. 38 (1), 35–53.

Myers, G.J, Badgett, T., Sandler, C., 2012. The Art of Software Testing, third ed. John Wiley and Sons Inc., Hoboken, NJ, USA.

Myers, G.J., Sandler, C., 2004. The Art of Software Testing. John Wiley & Sons.

PetClinic, 2021. Angular version of the spring PetClinic Web application. https://github.com/spring-petclinic/spring-petclinic-angular.

Phoenix, 2021. Phoenix: Trello tribute done in elixir, Phoenix framework. https://github.com/bigardone/phoenix-trello.

Rafi, D.M., Moses, K.R.K., Petersen, K., Mäntylä, M.V., 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: Proceedings of the 7th International Workshop on Automation of Software Test. pp. 36–42.

RetroBoard, 2021. Retrospective board. https://github.com/antoinejaussoin/retro-board.

Splittypie, 2021. Splittypie: Easy expense splitting. https://github.com/cowbell/splittypie.

Stocco, A., Yandrapally, R., Mesbah, A., 2018. Visual Web test repair. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 503–514.

Weißleder, S., Lackner, H., 2010. System models vs. Test models -distinguishing the undistinguishable? In: Informatik 2010: Service Science — Neue Perspektiven für Die Informatik, Beiträge der 40. Jahrestagung der Gesellschaft für Informatik E.V. (GI), Band 2, 27.09. - 1.10.2010, Leipzig. pp. 321–326.

Yandrapally, R., Stocco, A., Mesbah, A., 2020. Near-duplicate detection in Web app model inference. In: Proceedings of the 42nd International Conference on Software Engineering. pp. 186–197.

Nezih Sunman graduated from mechanical engineering in 2020 and is also a double-major student in the department of computer science at Ozyegin University. He is currently working as a software developer at Siemens Advanta in Istanbul, Turkey. Previously, he worked as a software developer intern at General Electric, Istanbul, Turkey for 1 year. He is the winner of the Automated Negotiating Agents Competition (ANAC), which was held at IJCAI 2020. He is interested in artificial intelligence, automated testing and Web-based application development.

Yiğit Soydan received his B.Sc. degree in computer science from Ozyegin University, Turkey, in 2021 and he joined the software development team at Yapı Kredi Technology where he is currently working as an assistant software developer.

Hasan Sözer received his B.Sc. and M.S. degrees in computer engineering from Bilkent University, Turkey, in 2002 and 2004, respectively. He received his Ph.D. degree in 2009 from the University of Twente, The Netherlands. From 2002 until 2005, he worked as a software engineer at Aselsan Inc. in Turkey. From 2009 until 2011, he worked as a postdoctoral researcher at the University of Twente. In 2011, he joined the department of computer science at Ozyegin University, where he is currently working as an associate professor.