Journal of Computer and Knowledge Engineering, Vol. 4, No. 1, 2021.(11-24)
DOI: 10.22067/cke.2021.72575.1025

A Semantic Web Enabled Approach to Automate Test Script Generation for Web Applications*

Research Article

Mahbobe Dadkhah[1] Saeed Araban[2] Samad Paydar[3]

Abstract: Software testing is one of the most important activities for ensuring quality of software products. It is a complex and knowledge-intensive activity which can be improved by reusing tester knowledge. Generally, testing web applications involve writing manual test scripts, which is a tedious and labor-intensive process. Manually written test scripts are valuable assets encapsulating the knowledge of the testers. Reusing these scripts to automatically generate new test scripts can improve the effectiveness of software testing and reduce the cost of required manual interventions. In this paper, a semantic web enabled approach is proposed for automatically adapting and generating test scripts; it reduces the cost of human intervention across multiple scripts by accumulating the human knowledge as semantic annotations on test scripts. This is supported by designing an ontology which defines the concepts and relationships required for test script annotation. The proposed approach is based on novel algorithms for adapting and generating new test scripts. The initial experiments show that the proposed approach is promising as it successfully increases the level of test automation.

Keywords: Automated testing, semantic web, Test adaptation, Test generation, Test ontology.

1. Introduction

Web applications are one of the mostly used software systems in our everyday life which require repetitive testing of their existing and new features due to their inherently evolving nature. Modern web applications within a domain usually implement a set of common features to be performed on a wide range of entities. For example, in the context of educational systems, features such as sorting or filtering are implemented for multiple entities such as presenting courses, and taking courses or classes. Rigorous testing of such systems requires creating a large number of scripts for testing each feature on every entity of the system. Testers usually tend to write as few test scripts as possible for a small number of entities due to the high cost of testing (i.e. time and budget). This leads to a limited test coverage and undetected errors, which will be mostly discovered by the end users. Moreover, there are common features such as pagination or login among many web applications which are implemented similarly. These similarities in implementing features can be translated into similarities of their test scrips structures. Reusing such scripts and adapting them to automatically generate multiple new test scripts can reduce the cost of testing.

Manually, writing test scripts is a complex, costly, and labor-intensive activity, especially if the number of needed test scripts is large. However, manual testing benefits from the domain knowledge of the tester who is writing the test scripts. Testers rely on their domain knowledge to recognize entities relevant to each feature. They also use their knowledge of testing to design a script consisting of a sequence of required steps to cover the business logic of a feature. In some test steps, testers should specify elements of the GUI as entities and describes their attributes to interact with, and they can also identify test data for each entity. The time and effort that testers put into writing manual test scripts makes them valuable assets of the system. Reusing these scripts requires explicitly specifying knowledge of the tester and separating it from the test data, test elements, and the structure of the Application Under Test (AUT).

In this paper, a semantic web enabled approach is proposed for reusing test scripts and adapting them to test the same feature on another entity of the same application, or to test a similar feature on another application. This process is based on proposing a three-level test script abstraction hierarchy that is conceptualized by semantic annotations representing the tester's knowledge. An ontology is defined to represent the concepts and relationships associated with test scripts. In addition, novel algorithms are proposed based on the ontological annotations to adapt and generate new test scripts.

The contributions of the present study is to propose a semi-automatic process for reusing test scripts. This process employs the following:

- Three levels of test script abstraction
- New algorithms for system-level and entity-level test script adaptation
- New test script generation algorithm

The initial experiments demonstrate that the proposed approach is appropriate and promising although more works are still needed to fully achieve the potentials.

The paper is organized as follows: a brief literature review is described in section two. In section three, the proposed approach is introduced and its underlying concepts and algorithms are described in details. The evaluation of the proposed approach is presented in section four, and finally, section five concludes the paper.

2. Related works

In this section, we briefly review works related to the proposed approach which fall into two categories: automated web application testing and semantic web enabled testing.

* Manuscript received August; 18, 2021, accepted. October, 16, 2021.

[1] PhD Candidate, Ferdowsi University of Mashhad, Iran.

[2] Corresponding author. Assistant Professor, Ferdowsi University of Mashhad, Iran. Email: araban@um.ac.ir.

[3] Assistant Professor, Ferdowsi University of Mashhad, Iran.

2.1 Automated web application testing

Web applications have been increasingly growing during the past two decades, and today they play an important role in our everyday life. The demand for quality web applications has resulted in proposing various automated techniques by researchers.

Crawling-based technique is one of the most studied approaches. In this technique, crawlers explore the state spaces and mine the behavior models of the applications. However, they are limited by the required manual configurations for input value selection [1]. Moreover, they are often application-specific which result in redundant test cases. The test dependency associated with the test cases obtained from a crawling session are used to eliminate redundant tests and produce minimized test suites [2]. A rule-based approach using input topic identification and GUI state comparison is proposed to represent DOM elements as vectors in a vector space formed by the words used in the elements [3].

The feedback-directed automated test generation is another technique proposed to use previously generated test cases for leveraging dynamic data [4]. A feedback-directed automated test generation framework is proposed by Artzi et al. [5], which collects execution information to generate test inputs leading to increased coverage. Yuan and Memon [1] proposed an approach to iteratively run generated test cases and use run-time information to enhance those test cases. Elbaum et al. [6] leverage user-session data gathered in the operation of web applications to assists test data generation. In another study [7], the captured user-session data are used for the automation of the replay and the oracle components of web applications.

Search-based test data regeneration technique introduced by Yoo and Harman [8] uses a meta-heuristic search algorithm for generating additional test data from existing test data. Mirzaaghaei et al. [9] define a set of heuristics to repair test cases and generate new test cases for evolved software. Rule-based approaches [3] use input topic identification and GUI state comparison. Gao et al. [10] use human knowledge in the form of tester annotations to automatically repair unusable test scripts. Testers annotate the automatically generated Event Flow Diagram (EFG), and the repairing transformations are then used to synthesize a new test script. The results showed that the proposed technique is effective and the annotations would reduce human costs. Milani et al. [11] proposed leveraging the existing crawling-based generated tests with human knowledge to extend the test suite for increasing code coverage.

2.2 Semantic web enabled testing

In our previous work [12], we conducted a systematic literature review to identify the state of the art and the benefits of semantic web enabled software testing in both industry and academia. The results showed that semantic web technologies would improve various activities in software testing process. Among these activities, test generation and test data generation gained more attention which would mostly rely on two significant test methods, including model-based and rule-based. It was found that model-based approaches would mostly use different UML diagrams, while rule-based approaches would utilize ontologies to model interactions, behaviors, EFGs, or GUI elements relations for test generation. In one study [13], for example, an ontology-based Behavior-Driven Development (BDD) approach is proposed for automated assessment of web GUIs. In this approach, a set of interactive behaviors on GUIs are predefined which can be implemented once and then are automatically reused to generate tests by building different scenarios in different business domains.

Most of the proposed test data generation approaches use ontology mapping. Hajiabadi and Kahani [14] proposed a test data generation technique in which test input for filling forms are automatically generated to model and evaluate dynamic features of the web application. In another research [15], semantic annotations are used for enriching Event Flow Graph (EFG) based on an ontology of GUI events. Semantic annotations have been used to automatically generate test data and test oracle [16]. In another study [17], the web of data is utilized to map GUI model to the classes and predicate them in the semantic knowledge-bases to generate realistic test data matching the semantics of the correlated test input fields.

Test reuse is another activity benefiting from semantic web technologies which is mostly based on semantic similarity metrics. For example, the semantic similarity between the existing test cases and test requirements of the application is tested as a basis for test reuse [18]. In another study, ontology matching technique is used for matching ontology of the AUT with the ontology of applications in which test cases are going to be reused.

3. The proposed approach

In this study, three levels of test script abstractions shown in Figure 1 are proposed based on the following observations:

1. Web applications within a specific domain usually provide some common features implemented in a similar way. For example, sorting feature provided in most web applications in the e-commerce domain. While the core logic of these features is similar (i.e., the main interactions between the user and the application), detail implementation of them may have some differences. This leads to some similarities in scripts considered for testing these features. For example, testing the sorting feature in every system requires first choosing the sorting criteria, and then verifying whether all the items are ordered based on the value of a specific attribute corresponding to the chosen sort criteria (e.g. price). These main interactions form the logic of a feature that are similar in most web applications and can be considered as logical test steps required for testing this feature. However, implementation of each interaction may be different in various systems. For example, one application includes only clicking on one of the presented links (each link representing one sorting criteria), while another application includes first opening a drop-down list of options and then clicking a link for choosing the sorting criteria. Based on this observation, a system-level abstraction is proposed to generalize a test script in a way that can be reused and adapted for testing a similar feature on other web applications. Such scripts are called *test interfaces*, which are independent of the AUT, entity under test, and test data.
2. Modern web applications perform their functionality through features that are usually implemented for

multiple entities. For example, filtering feature in most web applications in the e-commerce domain is provided for various types of entities. Users can filter the given products by choosing them from a list of data options for each of their attributes. For example, one can filter products representing Laptop entity based on Operating system attribute and choose Windows from the list of presented operating systems. Investigating scripts for testing such feature on various entities of an application shows that the structures of these scripts, including number and order of test steps along with some system-dependent elements and variables, are similar. Based on this observation, an entity-level abstraction is proposed to generalize a test script in a way that can be reused and adapted for testing the same feature on multiple entities of the same web application. Such scripts are called *abstract test scripts*, which are independent from test data but are written for a specific web application.

The lowest level of abstraction includes *concrete test scripts*, which are dependent on a specific data for the attribute of a specific entity in an application. This test script abstraction hierarchy can support automatic generation of concrete test scripts for testing a feature with various test data.

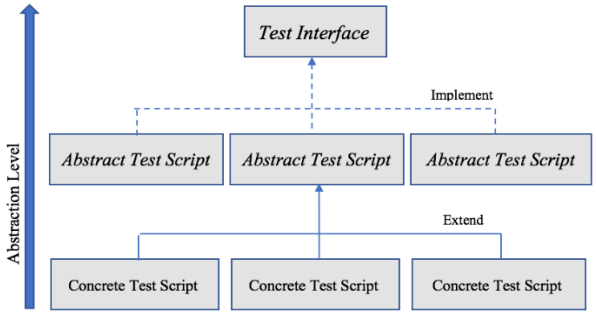

Figure 1. The proposed three levels of test script adaptation

The proposed approach utilizes test script annotation as a mechanism to realize these levels of abstractions and makes a test script independent of a specific application, entity, or test data. In the following, we will propose test script annotations in details along with the adaptation and generation algorithms designed based on these annotations.

3.1 Test Script Annotation

The proposed approach utilizes semantic web technologies, including semantic annotations and ontologies, to represent testers knowledge. Required concepts for annotating test scripts along with their properties and relations are formally defined by Test Script Ontology (TSO) which is shown in Figure 2. It is an application ontology [20] and hence does not cover all the concepts and relations in the software testing domain, except the concepts required for annotating test scripts used in the proposed approach. TSO is developed based on the ROoST ontology [21], which is a reference ontology in the software testing domain. The basic concepts of software testing domain, especially those that define a test script and different parts of it (e.g. test script, Test result, test input), are defined in the ROoSTs. Meanwhile, testing artifacts sub-ontology is reused in the TSO ontology.

The concepts defined in this ontology are used by the tester to increase the abstraction level of test scripts to be automatically reused by test script adaptation and generation algorithms. The TSO ontology defines five categories of concepts for test script annotation:

- 1) Concepts that define a test element as a test data provider. The scripts written for testing web applications determine the elements in the GUI of the AUT to interact with. These elements are determined by locators in the test steps of the scripts. Some of these elements can be used to locate and extract test data (i.e., test input and expected result) from the GUI that they belong to.
- 2) Concepts that define parts of a test script or a test step (e.g. test input, expected result). The main usage of these concepts is to determine the placeholder for test data that are provided by the annotations from previous category. When tester define an element as a provider, she/he must define where that provided data should be placed in the new generated test script.
- 3) Concepts that determine the dependency level of the parts of a test script or a test step (i.e., system dependent and entity dependent). System dependent annotation indicates that a part of script remains similar in testing all entities of the AUT. In contrast, entity dependent shows that a part of the test script needs to be adapted for different entities of an AUT.
- 4) Concepts that determine logical steps. The testers use these annotations to specify logical test steps based on their expertise in the testing domain and their knowledge of test scenario in order to test a common feature. Logical test steps are a bundle of multiple test steps which can be seen as a one logically meaningful step.
- 5) Concepts that determine a test step to be optional or mandatory. The mandatory test steps are the basic building blocks of the test script and should be present in all derived test scripts (i.e., adapted or generated). The optional test steps, in contrast, are dependent on the implementation of the AUT, and in some cases, they may not be present.

For better understanding of the proposed annotations, two annotated test script are described based on the level of independency they provide (see T and
T). These scripts are written using Selenium[4], which is a popular test tool in academy and industry. Each test step in selenium scripts has three parts: *command*, *target*, and *value*. Command specifies the action to be applied on a web element which is identified in the target. Some test steps require a data or a variable called value. If a test step has annotation, annotations and the step are presented in continuous line numbers of the script. For example, in
T, line two includes annotations for the test step in line three and line four includes annotations for the test step in line five. It is possible for a test step not to have any annotation (see line 8, 11, and 12 in Table 2). That is, these steps can be copied without any changes to the end scripts in the generation and adaptation algorithms. Some test steps might have multiple annotations. It is worth noting that if a test step has multiple annotations, the order of the annotations is not

[4] https://www.selenium.dev/

important for the adaptation and generation algorithms.

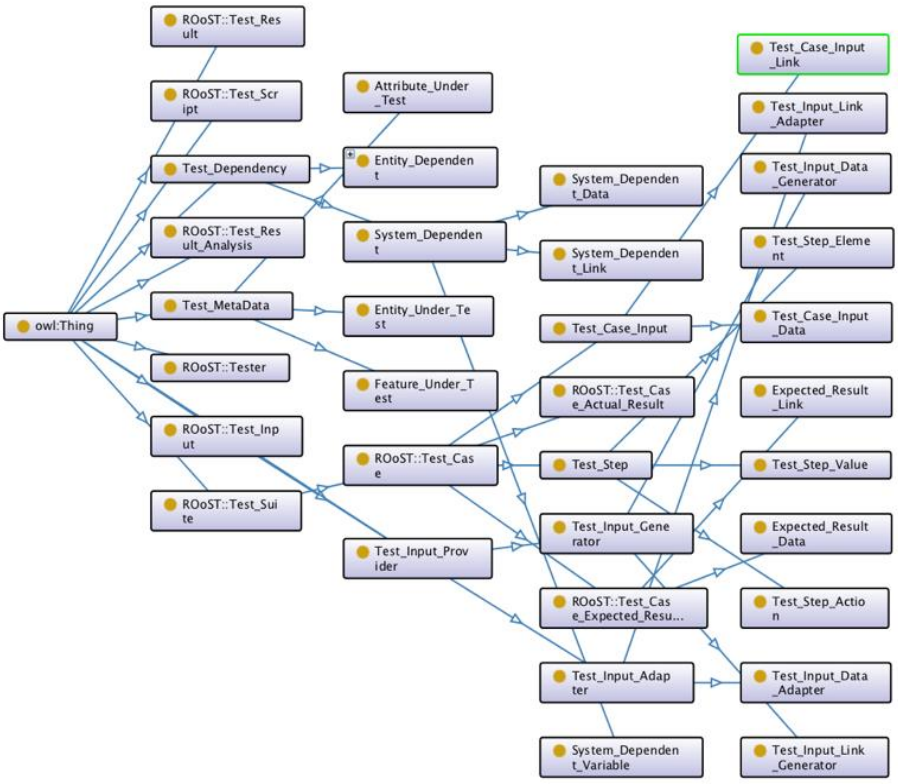

Figure 2. Test Script Ontology (TSO)

The annotated script in T 1 represents a test interface. This script is written to test login feature in Yahoo[5] web application and contains two logical steps. The first logical step (line 2-6) is for inserting the username and the second (line 7-11) is for inserting the password. In both logical steps, there is an optional step for clicking a button which may not be present in all applications. This script contains annotations from category four and five in addition to annotations from the first three categories. Annotations of category four and five give information about the whole test step, and they are thus used to annotate the command part of the test step.

The annotated script in
T 2 represent an abstract test script. This script is written for testing filter feature in Banimode[6] web application and contains annotations from the first three categories of annotations. In this application products are presented in web pages along with a set of filter sections. In each section, a list of possible data options is presented based on an attribute of the products (i.e., entities). For example, a filter section is based on brand attribute of the products including a list of checkboxes representing names of all brands that the products belong to. In this script, after opening the web page (line 1), one of the checkboxes representing a filter option is clicked (Reebok in this example). Then, all of the presented products are verified to have the same brand (Reebok in this script). The element of test step in line three is annotated as a provider (for both adaptation and generation processes) using annotations from category one. The English name tag of this element is the expected result of the script which is the value part of test step in line 10. The expected result is specified by the tester using annotations from category two in value part of the line nine (@Expected_Result). The automatically provided data for the expected result in the new adapted or generated test scripts should be placed in value part of the line 10. The value for variable in line 7 and the element of the verification step in line 10 are the same for all scripts testing filter feature in this application. Therefore, they are identified by the tester as system-dependent using annotations from category three.

As described in the previous test scripts, annotations from the first categories provide entity independency and can be used to create an abstract test script. Annotations from the last two categories provide system independency and can be used to create test interface.

Annotated test scripts are input to the proposed algorithms. The tester annotates test scripts manually; thus, it is possible that the tester forgets some annotations or that he/she has some inconsistencies in his/her annotations. Therefore, it is reasonable to perform some validations on the given annotations. A simple preprocessing is used in the experiments of this paper, including two steps:

1. Any inconsistencies in annotations are checked. For example, a test step cannot be both system-dependent and entity-dependent.
2. Any missing annotations are checked. For example, if there is an @Expected_Result_Provider annotation in the script, there must also be another annotation which defines the placeholder (an @Expected_Result annotation).

[5] https://www.yahoo.com/

[6] https://www.banimode.com/

Table 1. A case of test interface: An ATS for the feature of 'login' at Yahoo

TS1: Testing the feature of 'Login' at Yahoo with username and password data			
#	Command	Target	Value
1	open	https://login.yahoo.com	
2	@Start_ Logical_Step	@Test_Input_Data	
3	Type	id=login-username	username
4	@Optional	@Test_Input_Link	
5	Click	id=login-signin	
6	@End_Logical_Step		
7	@Start_Logical_Step	@Test_Input_Data	
8	Type	id=login-passwd	password
9	@Optional	@Test_Input_Link	
10	Click	id=login-signin	
11	@End_Logical_Step		
12	@Optional	@System_Dependent_Link	
13	click	linkText=Mail	
14	@	@System_Dependent_Link	
15	Assert-element -present	linkText=Compose	

Table 2. An abstract test script: An ATS for the feature of 'Filter' in Banimode web application

TS1: Testing the feature of 'Filter' for entity 'Shoes' based on the attribute of 'Brand' with data 'Reebok'			
#	Command	Target	Value
1	open	https://www.banimode.com	
2	@	@Expected_Result_Data_Adapter(xpath_suffix=/span/span[2]), @Expected_Result_Data_Generator(xpath_suffix=//span[@class='ename ename']),	
3	click	xpath=//div[@id='filter-manufacturers']/div/label	
4	@	@System_Dependent_Link	
5	store xpath count	xpath=//div[@id='product_list']/article	n
6	@	@System_Dependent_Variable	
7	execute script	return 1	i
8	while	${i} <= ${n}	
9	@	@System_Dependent_Link	@Expected_Result_ Data
10	verify text	css=.col-4:nth-child(${i}) .product-card-brand	Reebok
11	execute script	return ${i}+1	i
12	end		

The annotation preprocessing step aims at checking an annotated script to detect problematic issues to which the proposed approach is sensitive. If such an issue exists, the tester is asked to verify the script.

As annotated test scripts have an important role in the proposed approach, it is necessary to describe how they are defined in this approach.

Definition 1 (Annotated test script): An annotated test script (ATS) is formally defined as a tuple of the form $ATS = \{TC, SGUI, F, E, A, AUT\}$

where:

- TC is a sequence of $(s_1, \dots, s_k)$ where each s_i $(1 \leq i \leq K)$ is an annotated test step
- s_i is a tuple of the form $\{C_i, AC_i,\ T_i, AT_i,\ V_i, AV_i\}$ where:
  - C_i is the command of test step
  - AC_i is a set of annotations for C_i
  - T_i is a set of locators to target the element
  - AT_i is a set of annotations for T_i
  - V_i is the test data (if there is any)
  - AV_i is a set of annotations for V_i
- $SGUI$ is the root web page to run the test script
- F is the feature of the AUT to be tested
- E is an entity in the domain of AUT
- A is an attribute of E

Source GUI (SGUI) is part of the GUI of the AUT through which the test is done. The tester creates the test script for this GUI either manually or through testing tools such as Selenium. Therefore, the tester is expected to know this GUI and its elements, and he can annotate it based on the concepts defined in the TSO ontology to create an ATS.

Destination GUI (DGUI) is a GUI which we want to test it by adapting the given ATS. In entity-level adaptation, DGUI and SGUI are parts of the GUI of the same AUT; however, in the system-level adaptation they are part of the GUI of different applications.

3.2 Entity-Level Test Script Adaptation

The process of entity-level test script adaptation is defined as below:

Definition 2 (Entity-level test script adaptation). It is a process which takes an annotated test script $ATS = \{TC, SGUI, F, E, A, AUT\}$ with TC as a sequence of annotated test steps $(s_1, \dots, s_k)$ which is created to test feature F on attribute A of entity E in $SGUI$ of AUT, and then adapt it to test feature F on attribute A of entity E' in interface $DGUI$ of AUT and produce an Adapted Test Script $AdaptedTS = \{TC', DGUI, F, E', A, AUT\}$ with TC' as a sequence of $(s'_1, \dots, s'_k)$.

In this level of adaptation, the number (k) and order of test steps will not be affected. This is based on the idea that developers try to preserve consistency in the implementation of a feature and the structure of GUIs presented to the users throughout the whole system. The entity-level test script adaptation algorithm is described by the entity-level adaptation procedure shown in Figure 3. This algorithm involves a loop at a high level which iterates through each test step s_i and tries to adapt it to be executed successfully on $DGUI$. If the test step is not annotated, it can be copied to the $AdaptedTS$ (lines 4,5). If it is annotated as a system dependent step and contains an element, the element is checked to be present in DGUI and then is copied to the $AdaptedTS$ (lines 6-9). In other cases, element step (e_i) of a test specified by a set of locators T_i in SGUI needs to be adapted. The goal is to find a corresponding element e'_i in $DGUI$ as the target element of test step s'_i in such a way that s'_i can be executed successfully on DGUI.

```
Algorithm: Entity-level test script adaptation
input:
    ATS: An Annotated Source Test Script
    DGUI: Destination GUI
output:
    AdaptedTS: Adapted Test Script
1:  Procedure entityLevelAdaptation ()
2:      TC' = (s'1, ..., s'k) ∈ AdaptedTS where s'i =∅
3:      For all test steps si ∈ TC
4:          If si.annotation = NULL
5:              AdaptedTS. add (si); DGUI ← execute (si, DGUI); continue;
6:          If annotation ANNOT ∈ ATi where ANNOT = system-dependent
7:              ei ← findElement (Ti, SGUI); e'i ← findElement (Ti, DGUI)
8:              If e'i != NULL and Type(e'i) = Type(ei)
9:                  AdaptedTS. add (si); DGUI ← execute (si, DGUI); continue;
10:         suggestedSteps ← GUIBasedElementAdaptation (si, SGUI, DGUI)
11:         If tester confirms s'i from list of suggestedSteps
12:             AdaptedTS. add (s'i); DGUI ← execute (s'i, DGUI); continue;
13:         suggestedSteps ← SemanticEnabledElementAdaptation (si, SGUI, DGUI)
14:         If tester confirms s'i from list of suggestedSteps
15:             AdaptedTS. add (s'i); DGUI ← execute (s'i, DGUI); continue;
16:     tester may make manual modifications to the AdaptedTS
17:     For all test steps s'i ∈ TC'
18:         If annotation ANNOT ∈ AT'i where ANNOT. type = Adapter
19:             testDataAdaptation (s'i, ANNOT, DGUI)
20: Return AdaptedTS

1:  procedure GUIBasedElementAdaptation (si, SGUI, DGUI)
2:      ei ← findElement (Ti, SGUI)
3:      For all element e'i ∈ DGUI
4:          If e'i.Id = ei.Id and e'i.type = ei.type
5:              si. setTarget(e'i); suggestedSteps.append(si); Return
6:          If e'i.location = ei.location and textualSimilarity (e'i.associatedText, ei. associatedText) ≥ θ
7:              si. setTarget(e'i); suggestedSteps.append(si); Return

1:  procedure SemanticEnabledElementAdaptation (si, SGUI, DGUI):
2:      Get list of candidateElements ∈ DGUI
3:      For each element e'i in candidateElements
4:          DGUIEs ← findElement (e'i. getTarget, ANNOT. parameter, DGUI);
5:          If DGUIEs != Null
6:              SGUIEs ← findElement (t, ANNOT. parameter, SGUI);
7:              SGUIR ← findSemanticRelation (ei, SGUIEs)
8:              DGUIR ← findSemanticRelation (e'i, DGUIEs)
9:              If equivalent (SGUIR, DGUIR)
10:                 si.setTarget (e'i); suggestedSteps.append(si); Return
```

Figure 3. Entity-level test script adaptation algorithm

The entity-level adaptation algorithm contains two major functions for element adaptation. The first one is *GUI-based*

element adaptation, which aims to adapt elements only based on the information presented in the SGUI and DGUI in two phases. The first phase looks for exactly the same element in DGUI (lines 24, 25). Exact elements are defined by their Id attribute, because Ids should be unique in a web page as stated by W3C standards. Therefore, if the element has an Id attribute for each element of test steps in ATS, the DGUI is searched for an element with the exact same Id. If such an element found in DGUI, it will be considered as the adapted element of that test step.

This phase of element adaptation is also applicable for locators that specify test data. For example, consider the test script for examining the feature of 'Filter' in the attribute Brand of different entities in Banimode website. In this application, each one of the data options for selecting a brand in the 'Filter' section has a unique Id which is identic for all the entities and web pages in the whole system. Therefore, if such test script is adapted with test data 'jeanswest' for another entity in this application, then the exact test data 'jeanswest'' will be a valid test data for the adapted test script. If an adapted element could not be find in this phase, then the second phase will be performed.

The second phase includes searching DGUI for an element which is located in the same place of that element in SGUI (lines 26, 27). This is based on the idea that the structure of GUIs is normally organized in order to assure consistency in the web page appearance and to increase the usability of the application. But, if there is such an element in DGUI, the similarity of its associated text will be checked against the associated text of e_i. The associated text of an element is defined with the nearest text to that element in the DOM structure. If the element itself does not contain a text or label, then its inner or outer text of will be considered as its associated text. Since in the entity-level adaptation both GUIs belong to the same AUT, it is expected that developers use similar phrases for representing a concept through the system.

If the element cannot be adapted only based on GUI information, then a semantic web enabled approach is used to find semantically similar elements. It is based on the idea that in a script for testing a feature there may be a meaningful relation between test elements and test data of the script. This relation in scripts for testing similar attributes on different entities can be similar. For example, consider a script for testing the feature of 'Filter' over 'Laptop' entity based on their operating system in Digikala[7] website. The script is similar to the one in
T 2. In this script, the entity name and the data options for the attribute under test (a list of existing operating systems for laptops) can be extracted from the elements in the script using testers annotations. Therefore, we have a set of semantically related data in the test script extracted from SGUI to which the ATS belongs using annotations. For example, this set of data can be {laptop, {Microsoft Windows 10, Apple Mac OS, Google Chrome}}. When reusing this script for testing the same feature on another entity (e.g. Smart Phone) due to the differences in number and names of attributes in two entities, GUI-based adaptation is not successful. However, the same semantic relation may exist between a corresponding set of data in DGUI. For example, the corresponding set of data in Digikala is {smart phone, {Android 10, iOS 10, Windows Phone 8}. Therefore, in this phase the GUI is searched for finding a set of elements that their associated texts have the same semantic relation to the set of data in SGUI extracted from ATS. The semantic web data sources are searched in this phase to find the semantic relation between two sets of data.

This process is described and is performed by semantic-enabled element adaptation in four phases.

Phase one: It includes searching for candidate elements in the DGUI (line 4). The first group of candidate elements includes elements of the DGUI with associated text similar to the e_i when it is expected that developers use similar phrases for representing a concept in the application. The second group of candidates includes elements with the same type of the e_i when it is expected that developers use the same type of elements for implementing a feature for similar attributes of various entities.

Phase two: It includes identifying the candidate set of data in DGUI using provider annotations. For this purpose, annotations from category two are used to find elements that have structural relations with each one of candidate elements. If e_i has provider annotations (adapter or generator), the provided data options in the SGUI are identified (line 6). The provider annotations in ATS specify a structural relation between e_i and the provided set of data options in SGUI using relative XPath or XPath Axes. This set of data is $\{e_i, \{\text{SGUIEs}\}\}$. As it is expected that the adapted element e_i' has similar structural relation with its own provided data options, this structural relation is checked for each candidate elements of DGUI too (lines 3-5). If such structural relation exists for any of the candidate elements, then that set of data would be considered for finding semantic relations. A set of data for each e_i' in candidate elements is $\{e_i', \{\text{DGUIEs}\}\}$.

Phase three: This phase involves finding semantic relations between data in each set of data using the semantic web. In this phase, the existence of any semantic relation between the two sets of data is checked. First, the semantic relation between e_i and provided data options by e_i (SGUIEs) is searched (line 7) as the first set of data. Then, the semantic relation between each candidate element e_i' and provided data options by e_i' (DGUIEs) is searched (line 8) as the second set of data. For this purpose, a SPARQL query is created using sets of data. The associated texts of e_i, e_i', and their provided data are mapped to the data of the semantic web knowledge-bases using a SPARQL query that looks for predicates. Since predicates are used more often than classes to represent attributes, and attribute under test is supposed to be similar for various entities, we only try to map associated texts to a predicate. If no predicate is found, this process can be extended to search alternative namespaces in a knowledge base or other knowledge bases. In the experiments of this paper, DBpedia is searched as one of the largest knowledge bases available on the web. DBpedia knowledge base is accessed online through its SPARQL endpoint, which is an interface that supports information retrieval from DBpedia through SPARQL queries. Therefore, the proposed approach can work with other knowledge bases that implement a SPARQL endpoint. If a semantic relation is found in

[7] https://www.digikala.com

DBpedia between data in any candidate sets of data, then the similarity of this relation to the relation between e_i and its provided data is checked.

Phase four: This phase involves checking the similarity of semantic relations between sets of data in SGUI and DGUI. For this purpose, the equivalent procedure is used. The simplest situation is when the semantic relation between two sets of elements is exactly the same, i.e., exactly the same predicate. In this situation e_i' is considered as the adapted element of e_i. If the two semantic relations are not exactly the same, the semantic similarity of these relations is checked. In the semantic web, the similarity of the relations can be represented by defined properties. In this work, the existence of owl:equivalent property between two relations is considered as their similarity. If the two relations were similar based on this property, then e_i' is considered as the adapted element of e_i. Therefore, in this stage, an element in the DGUI will be considered as an adapted element when both the structural and semantic relation exist.

After adapting test elements, the tester can modify the $AdaptedTS$ manually if it is needed. The resulted TC' includes a set of adapted test steps with adapted elements. However, in case the script has provider annotations of type adapted, its test data should be updated. Consider the example for feature 'Filter' in Digikala, where the ATS is written for 'Laptop' entity with data Microsoft Windows 10 as its expected result. Now that this script is adapted to test 'Smart phone' entity, the expected result should be updated to a valid data for this entity. The *test data adaptation* process is performed to adapt the test data on DGUI based on the provider annotations specified by the tester (line 19).

3.3 System-Level Test Script Adaptation

This level of adaptation is performed on two different systems that implement a similar feature. Therefore, the SGUI and DGUI have different structures and belong to different AUTs. At this level, if the entity under test in DGUI is different from the entity in SGUI, then adaptation is not effective due to the differences in both structure and semantic of the DGUI to the SGUI. Adapting a script for testing a similar feature on a different entity of a different application requires changing many parts of the script. The automated adaptation is not logical in this case due to the minimum automation and maximum manual intervention. Therefore, in this experiment, entity-less test scripts are considered for system-level adaptation.These scripts test features are independent from a specific entity or its attributes (e.g. sorting).

The proposed system-level adaptation process takes as input a *test interface*, which represents a general and comprehensive scenario for testing a given feature. In test interfaces, tester can specify the most general condition with all required test steps and then annotate the optional test steps that may not be present in all applications. In this case, the number of required test steps in a logical test step of the is fewer than the number of test steps in that logical step of the ATS. In contrast, if the tester does not create a general and comprehensive test interface, the input ATS may lack some required test steps in logical steps for examining specified feature on DGUI. In the proposed semi-automated approach, it is assumed that the tester creates a comprehensive enough ATS and otherwise he/she can manually modify the produced $AdaptedTS$. Therefore, the number of test steps in the $AdaptedTS$ (k') may be different from ATS (k), but the order of existent steps will not be affected. Based on these assumptions, the process of system-level test script adaptation is defined as below:

Definition 3 (System-level test script adaptation). It is a process which takes an annotated test script $ATS = \{TC, SGUI, F, AUT1\}$ with TC as a sequence of annotated test steps $(s_1, \dots, s_k)$, which is created to test feature F in interface $SGUI$ of $AUT1$ and then adapt it to test feature F in $DGUI$ of $AUT2$ and produce an adapted test script $AdaptedTS = \{TC', DGUI, F, AUT2\}$ with TC' as a sequence of $(s_1', \dots, s_{k'}')$ where $k' \leq k$.

Algorithm: System-level test script adaptation
input:
  ATS: An Annotated Source Test Script for $AUT1$
  $DGUI$: Destination GUI of $AUT2$
output:
  $AdaptedTS$: Adapted Test Script for examining $AUT2$

1: **procedure** systemLevelAdaptation ()
2: $TC' = (s_1', \dots, s_k') \in AdaptedTS$ *where* $s_i' = \emptyset$
3: $LTS[] \leftarrow$ extractAllLogicalTestSteps(TC)
4: **for all** logical test steps $LTS_j \in LTS$ **do**
5: **for all** test steps $s_i \in LTS_j$ **do**
6: $e_i \leftarrow$ findElement (T_i, $SGUI$)
7: **for all element** $e_i' \in DGUI$ **do**
8: Find e_i' with maximum *semanticSimilarity* (e_i, e_i')
9: s_i.setTarget (e_i'); *suggestedSteps.append*(s_i)
10: **if** *suggestedSteps* = Null
11: **if** annotation $ANNOT \in AT_i$ where $ANNOT$= Optional
12: Notify tester of an optional test step
13: **If** tester confirms s_i' from list of *suggestedSteps*
14: $AdaptedTS$. *add* (s_i'); *DGUI* $\leftarrow$ execute (s_i', *DGUI*); continue;
15: tester may make manual modifications to the $AdaptedTS$
16: **For all** test steps $s_i' \in TC'$
17: **If** annotation $ANNOT \in AT_i'$ where $ANNOT$. *type = Adapter*
18: *testDataAdaptation* (s_i', *ANNOT, DGUI*)
19: Return $AdaptedTS$

Figure 4. System-level test script adaptation algorithm

The system-level test script adaptation algorithm is described by the *system level adaptation* procedure shown in

Figure 4. The whole adaptation process involves two nested loops at a high level: 1) The outer loop iterates through each logical test step LTS_j and processes each logical test step as a whole (lines 4-15). 2) The inner loop iterates through each test step s_i from a particular logical test step $s_i \in LTS_j$ and aims to adapt it to be successfully executed on DGUI. The system-level adaptation algorithm uses semantic similarity for element adaptation. The goal is to adapt element (e_i) of a test step specified by a set of locators T_i in SGUI of AUT1 and find a corresponding element e_i' in $DGUI$ of AUT2 as the target element of test step s_i' in such a way that s_i' can be executed successfully on DGUI.

In entity-level adaptation, structural information from GUI and semantic relations between elements of GUI are used for element adaptation. This is based on the idea that developers maintain consistency in implementing a feature for various entities of an application. In system-level adaptation, such similarity does not exist between structures of GUIs in two different web applications. Therefore, at this level of adaptation, semantic similarity of web elements is used. This is based on WordNet, which is a lexical database of semantic relations between different words in a network of words.

In the proposed approach, semantic similarity between the two web elements is computed as a weighted sum of the similarity of their Ids, names, and associated texts. This is based on the idea that the developers intentionally use meaningful id and name attributes that probably represent the semantic of that element. The associated texts of an element include its text or label. If the element itself does not contain a text or label, then the inner or outer text of that element will be considered as its associated text. Therefore, the associated texts of an element represent the function of that element to the end users and should be a meaningful phrase which indicates its usage. In the proposed approach, the semantic similarity between web elements is computed by the following formula:

Semantic similarity ($Element_1$, $Element_2$) =
W_{id} * WNSimilarity (id_1, id_2) +
W_{name} * WNSimilarity ($name_1$, $name_2$) +
W_{text} * WNSimilarity ($text_1$, $text_2$)

where W_{id}, W_{name}, and W_{text} are the weights which determine importance of similarity of the id, name, and associated texts of the two elements. Having two lists of terms $T = \{t_1, t_2, \dots, t_n\}$ and $T' = \{t_1', t_2', \dots, t_n'\}$ so that $|T| \leq |T'|$, *WNSimilarity* (T, T') is equal to the value of the best correspondence between T and T'. Each correspondence is a set of assignments of t_j' *(1≤j≤n)* elements to t_i *(1≤i≤m)* elements where no t_j' is assigned to more than one. The best correspondence is the one which has the largest value among all possible correspondences. Finally, the value of a correspondence R is computed based on the proposed formula by Paydar [PaydarInfsoft].

3.4 Test Script Generation

The proposed three levels of test script abstraction along with the entity-level and system-level adaptation algorithms provide the foundation for automatically generating test scripts.

Automatic test script generation involves two activities. First, generating a correct sequence of test steps to test the intended feature of the AUT. Second, it generates a set of test data in accordance to that sequence of test steps. The proposed adaptation algorithms perform the first activity and provide a sequence of test steps. This sequence can be used to generate multiple test scripts with different test data.

Test script generation process takes as input an abstract test script in the form of an ATS; therefore, this script is adapted to implementation details of the application under test and the underlying entity. The abstract test script can be created in two ways. First, tester can write a test script for the intended entity of the AUT and annotate it with concepts from annotation categories one, two, and three to create an abstract test script. Second, a test script that is written for another entity or another application is reused to produce an abstract test script through entity-level or system-level adaptation algorithms. In both cases, the test script includes required annotations to provide test data for automatically generating multiple concrete test scripts. The process of test script generation is defined as below:

Definition 4 (Test script generation). It is a process which takes as input an annotated test script $ATS = \{TC, SGUI, F, E, A, AUT\}$ with TC as a sequence of annotated test steps $(s_1, \dots, s_k)$ which is created to test feature F on attribute A of entity E in $SGUI$ of AUT, and then generate a set of test scripts $\{TS_1, \dots, TS_n\}$ with a set of test data $\{TD_1, \dots, TD_n\}$ in which $TS_x = \{TC_x, SGUI, F, E, A, AUT\}$ with TC_x $(s_1, \dots, s_k)$ to test feature F on attribute A of entity E in $SGUI$ of AUT.

There are different approaches proposed in the literature for generating test data such as ontology mapping [22], rule-based approaches [23], using the web of data as a source of test data generation [24], or simply specifying a repository for importing required data. In the proposed approach, the required test data is extracted from the GUI of AUT based on the annotations of the tester. This is based on the idea that in some features such as 'Filter' the required test data (e.g. test input and expected result) are presented as options in the GUI that can be extracted. The tester can enrich the test scripts with his/her knowledge of the AUT using annotations. Then, these annotations can be used to automatically generate test data.

4. Evaluation

For the purpose of evaluation, a prototype of the proposed approach is implemented in Java, and then it is evaluated. Two popular web applications from the e-commerce domain are selected, i.e., Digikala and Banimode, respectively referred to as APP1 and APP2. Four features of these applications with different levels of complexity are selected, which are 'Filter, 'Sort', 'Pagination', and ' Login'. For brevity, we refer to these features as F1, F2, F3, F4. In this

experiment, Selenium is used for creating ATSs and also for executing produced test scripts. The experiment is performed on a Macbook Pro laptop running Mac OS X10 with Intel Core i5 processor (2.4 GHz) and 8 GB memory.

The proposed approach includes three main processes: entity-level test script adaptation, system-level test script adaptation and test script generation. In this section, evaluation of these processes is separately discussed. Then, the efficiency of the proposed approach and the effectivity of semantic web for the proposed approach are evaluated.

4.1 Entity-level test script adaptation

In order to evaluate the proposed entity-level adaptation algorithm, an experiment for testing three common features of four applications is conducted (i.e., F1, F2, F3). For this experiment, first, a test script for each feature on every application is manually created. Then the test scripts are annotated to create abstract Test scripts (6 test scripts in total). Each abstract test script is adapted to a set of 20 randomly selected DGUIs of the same application with the same feature (120 in total).

For measuring the effectiveness of this algorithm, the percentage of executable, modified, preserved, and adapted test steps in the result test scripts produced by entity-level adaptation algorithm are reported. The concept of executable indicates the percentage of successfully executable test steps to the all test steps of produced $AdaptedTS$s and shows how successful the proposed algorithm is in adapting test scripts. The concept of modified indicates the percentage of test steps that the proposed algorithm failed to adapt and hence needed manual modification by the tester to the all test steps of produced $AdaptedTS$s. The goal is to increase the level of automation by decreasing these manual interventions by the tester. The concept of adapted indicates the percentage of successfully adapted test steps to the executable test steps of produced $AdaptedTS$s. The concept of preserved indicates the percentage of test steps that are copied from ATS to the $AdaptedTS$s without any change. These are test steps without any annotation or test steps annotated as system-dependent that can be reused without any change. The results are shown in T 3 for each feature of every application.

Table 3. Results of the entity-level test script adaptation

Application	APP1			APP2		
Feature	F1	F2	F3	F1	F2	F3
Executable (%)	89.1	97.3	93.4	97.1	98.4	99
Modified (%)	10.9	2.7	6.6	2.9	1.6	1
Adapted (%)	18.7	16.2	13.2	12.5	3.5	28.6
Preserved (%)	81.3	83.8	86.8	87.5	96.5	71.4

Analysis of the results shows that the average percentage of executable test steps for the proposed entity-level adaptation is about 95.7%. Among these executable test steps, about 84.5 are preserved from the given ATS. This shows that when adapting a test script for the same feature of a system, more than 80% of test steps can be reused without any changes. About 15.5% of executable test steps are adapted successfully by the proposed approach. Test elements in the resulted test steps are adapted using the two proposed procedures. T 4. shows percentage of test elements adapted using each one. About 19% of the test elements in a script are adapted using the proposed semantic web enabled procedure. Automatically adapting these elements is a challenging task and is one of the strengths of the proposed approach which is provided by the Semantic Web.

Table 4. Percentage of elements adapted based on each proposed procedure

Entity-level element adaptation procedures	GUI-based	Semantic enabled	Total
Percent (%)	80.6%	19.4%	100%

4.2 System-level test script adaptation

Evaluating the proposed system-level adaptation algorithm, is done by another experiment for testing three features of the two applications (i.e., F2, F3, F4). The system-level adaptation algorithm needs a Test Interface as its input. Therefore, first test scripts for each feature on every application is manually created. Then, test scripts are annotated to create a test interface which is comprehensive enough to be able to test a similar feature on other applications. The results are shown in T 5. for each feature in every application. The effectiveness of this algorithm is measured in a similar way to the entity-level adaptation algorithm.

Table 5. Results of the system-level test script adaptation

Feature	F2		F3		F4	
Application	AP P1	AP P2	AP P1	AP P2	AP P1	AP P2
Executable (%)	85.7	80.9	84.2	94.7	80	73.3
Modified (%)	14.3	19.1	15.8	5.3	20	26.7
Adapted (%)	25	20	0	13.3	85.7	87.5
Preserved (%)	75	80	100	86.7	14.3	12.5

The percentage of executable test steps for features F2 and F3 is more than 80% in two applications, but the percentage of adapted test steps is less than about 25%, which indicates that most of the executable test steps were preserved. This shows that entity-less scripts for testing some similar features in different systems have more than 80% similar test steps. In contrast, the percentage of adapted test steps for feature F4 is more than about 85%. That is, most of the steps in scripts for testing this feature needs to be adapted and the proposed approach successfully adapts them. The low percentage of the preserved test steps in this feature is due to the fact that most of the steps in the script are actions to be applied on an element of the GUI and thus these test elements should be adapted to the application under test.

The percentage of modified test steps in features F2 and F4 are more than F3. The proposed approach fails at adapting these test steps and tester manually modified them. One of the main reasons is that scripts for testing these features include test steps that execute JavaScript functions on a variable. The proposed approach fails at adapting targets

including variables in such test steps. Another reason is related to the difference in verification steps of testing similar features on different applications. Verification steps are usually very dependent to the application under test, which makes adapting the elements of these steps to be a challenging task. The average percentage of executable test steps for the proposed system-level adaptation is 83.1%. The percentage of adapted test steps is different and dependent on the feature under test.

4.3 Test Script Generation

The proposed test generation algorithm is evaluated by conducting an experiment to generate concrete test scripts for feature F1 in the two applications. The proposed generation algorithm uses tester's knowledge of the AUT in the form of semantic annotations. The proposed test script generation algorithm is evaluated by the percentage of automatically generated test scripts with valid test data that can be executed successfully. The results for APP2 is 100%, which means that all the possible test scripts for APP2 are automatically generated and a maximum test coverage is provided. Filter F1 in APP2 is implemented with constant and similar attributes for all products of the system which facilitates test script generation. The results of APP1 is about 87% because feature F1 in this application is implemented for a wide range of products with different test data options for each attribute. Based on these results, the automation level provided by the proposed approach is promising. The most test steps that the proposed approach fails to successfully generate are verification steps. When using different data for testing this feature on various products, the element to be verified is located in different places of the DOM structure which makes generating test data harder.

4.4 Efficiency

The proposed approach is a semi-automated approach which requires tester's intervention in some cases. In this approach, tester can perform two types of manual operations: 1) modification and confirmation. Modification requires modification of a GUI element, test data, or a test step, while Confirmation includes selecting, deleting or confirming the suggested test data or GUI elements, and deleting an optional test step. For evaluating efficiency of the proposed approach, the cost of human intervention is measured in terms of time spent on each operation; therefore, the number of operations is counted. The average number of modified, confirmed, and automatic operations for all the three experiments described above are shown in Figure 5.

The results show that for all algorithms and features in these experiments the percentage of manual operations is quite small compared to the automatic operations. The proportion of manual operations in system-level adaptation algorithm is more than other two algorithms and in generation algorithm is less than others. In entity-level adaptation algorithm and generation algorithm, the proportion of manual confirm operations is more than manual modify operations while in system-level adaptation algorithm, this is the opposite. And for all features, the automatic operations performed by the proposed algorithms are much larger than those performed manually.

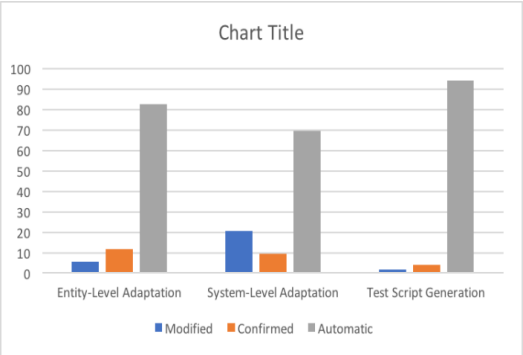

Figure 5. The operation cost of the proposed algorithms

4.5 Semantic Web Readiness

The proposed entity-level test script adaptation algorithm utilizes the Web of Data for finding web elements that have semantically meaningful relations. Therefore, we conducted an experiment to evaluate the possibility of finding such relations between web elements. This experiment seeks to find out for what percent of the web elements in a GUI it is possible to find a semantic relation on the semantic web. However, since the semantic web contains various sources, it is not possible to search all of them. In this experiment, DBpedia is used as representative of the semantic web sources. For this experiment a set of 10 GUIs are selected from the APP1. The results show that for 81% of the attributes, there is a subject with that attribute linking with at least one of its data options. Amon these attributes, there are similar attributes for different entities. For example, different digital devices such as tablets, laptops, computers, and mobile phones have similar attributes (e.g. memory, processor type, and display size). Results show that for 86% of these attributes are represented with similar predicates on the semantic web. These results provide good potential for the proposed entity-level element adaptation.

4.6 Fault detection

In this section, an experiment for measuring the effect of the proposed approach on test coverage and fault detection is described. At the end of the year 2020, the 'Sort' feature in Digikala website was extended to sort the products based on their discount in a way to show the most discounted products first. This feature was released without proper testing and undetected errors were discovered by the end users in many pages of the application. This was probably the result of a limited test coverage due to the required time and cost of testing. In November and December 2020, we conducted an experiment to evaluate the ability of the proposed approach in improving test coverage and fault detection. For this purpose, the required scripts for testing this feature on different pages of this application were automatically generated. The input ATS for proposed test script generation was created in two ways (see Figure 6).

The first one is to directly create a script for this feature in Digikala website and annotate it to be an abstract test script ready for test generation process. The second one is to reuse an existing test interface in another application (shown in grey). The 'Sort' feature based on most discount was supported in Banimode application and its test interface was previously created for the previous experiment. Then, it is required to adapt this test interface for the Digikala application using the proposed system-level adaptation algorithm. This adaptation required only three manual

operations: two modification operations and one confirmation operation. Both ATSs were used by the test script generation process to create multiple test scripts for a set of 20 DGUIs. The produced test scripts were run automatically by Selenium tool. These DGUIs were also investigated manually to discover existing faults. The faults discovered by running scripts produced by the proposed approach were compared to the manually discovered faults. The results show that about 99.3% of faults are discovered.

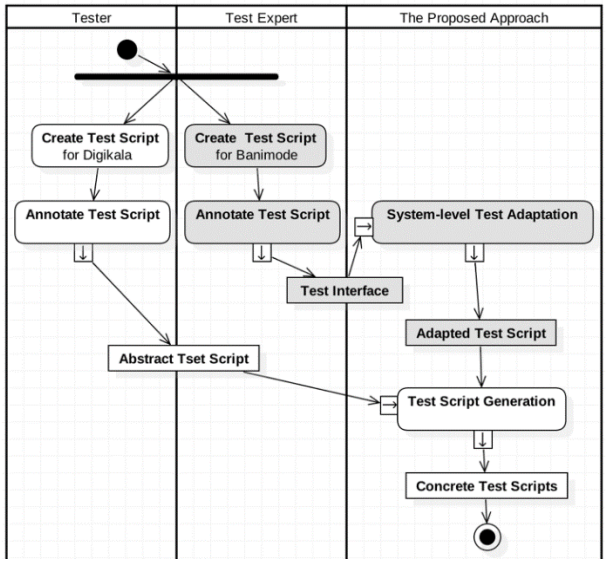

Figure 6. Different ways of creating an entity-less script for test generation process

5. Conclusion

This work was motivated by two observations: 1) web applications in a domain providing common features and implementing these features in a similar way, 2) several web applications providing features that have been implemented for multiple entities and attributes. Usually, test scripts have to be re-written for each attribute of every entity in the system domain. Human-written test scripts are valuable sources of knowledge that can be reused. Reusing test scripts is a knowledge-intensive activity and can be improved by effective utilization of semantic web technologies. The goals of the proposed approach are to increase the level of test automation and reduce testing cost by introducing a three-level test abstraction hierarchy to separate test logic and structure from underlying application, entity and test data. These levels of test script abstraction are realized by annotating test scripts based on the concepts defined by the TSO ontology.

Our approach consists of algorithms for test script reuse that are designed based on these abstraction level: 1) an entity-level test script adaptation algorithm which adapt annotated scripts for testing the same feature on other entities of an application, 2) a system-level test script adaptation algorithm which adapt annotated scripts for testing a similar feature on other applications, and 3) a test script generation algorithm to automatically generate concrete test scripts. Our evaluation results of the two real-world applications show that the proposed approach is promising in terms of effectivity and efficiency.

The results also demonstrate that the automatic operations performed by the proposed approach are much larger than the required manual operations. The approach is related to the semantic web in two ways. First, it exploits ontologies for semantic annotation and provides testers with mechanisms to annotate test scripts with their knowledge. Second, it uses the semantic web sources to automatically obtain its required information. In a nutshell, the empirical results suggest that this idea is both feasible and promising.

References

[1] X. Yuan and A. M. Memon, "Iterative execution-feedback model-directed GUI testing," *Information and Software Technology*, Vol. 52, No. 5, pp. 559–575, (2010).

[2] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Dependency-aware web test generation," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, pp. 175–185 (2020).

[3] J.-W. Lin, F. Wang, and P. Chu, "Using Semantic Similarity in Crawling-Based Web Application Testing," in *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation*, pp. 138–148 (2017).

[4] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 75–84 (2007).

[5] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of JavaScript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 571–580 (2011).

[6] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II, "Leveraging user-session data to support web application testing," *IEEE Tranaction on Software Engineering*, vol. 31, no. 3, pp. 187–202 (2005).

[7] S. E. Sprenkle, L. L. Pollock, and L. M. Simko, "Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour," *Software Testing, Verification, and Reliability*, vol. 23, no. 6, pp. 439–464 (2013).

[8] S. Yoo and M. Harman, "Test data regeneration: generating new test data from existing test data," *Software Testing, Verification, and Reliability*, vol. 22, no. 3, pp. 171–201 (2012).

[9] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting test suite evolution through test case adaptation," *IEEE International Conference on Software Testing, Veriication and. Validation, ICST 2012*, vol. 2, pp. 231–240 (2012).

[10] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "SITAR: GUI Test Script Repair," *IEEE Transaction on Software Engineering*, vol. 42, no. 2, pp. 170–186 (2016).

[11] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah, "Leveraging existing tests in automated test generation for web applications," in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14, pp. 67–78 (2014).

[12] M. Dadkhah, S. Araban, and S. Paydar, "A systematic literature review on semantic web enabled software

testing," *Journal of Systems and Software*, vol. 162, p. 110485 (2020).

[13] T. R. Silva, M. Winckler, and H. Trætteberg, "Ensuring the Consistency Between User Requirements and Graphical User Interfaces: A Behavior-Based Automated Approach," in *International Conference on Computational Science and Its Applications*, pp. 616–632 (2019).

[14] H. Hajiabadi and M. Kahani, "An automated model based approach to test web application using ontology," in *IEEE Conference on Open Systems*, pp. 354–359 (2011).

[15] A. Rauf, S. Anwar, M. Ramzan, S. ur Rehman, and A. A. Shahid, "Ontology driven semantic annotation based GUI testing," in *International Conference on Emerging Technologies (ICET)*, pp. 261–264 (2010).

[16] R. Tönjes, E. S. Reetz, M. Fischer, and D. Kuemper, "Automated testing of context-aware applications," (2015).

[17] L. Mariani and M. Pezze, "Link : Exploiting the Web of Data to Generate Test Inputs," (2014).

[18] R. Li and S. Ma, "The Use of Ontology in Case Based Reasoning for Reusable Test Case Generation," in *International Conference on Artificial Intelligence and Industrial Engineering*, pp. 369–374 (2015).

[19] S. Dalal, S. Kumar, and N. Baliyan, "An Ontology-Based Approach for Test Case Reuse," in *Intelligent Computing, Communication and Devices*, pp. 361–366 (2015).

[20] C. Menzel, "Reference Ontologies — Application Ontologies : Either / Or or Both / And ?," (2003).

[21] É. F. de Souza, R. de A. Falbo, and N. L. Vijaykumar, "ROoST: Reference Ontology on Software Testing," *Appl. Ontol.*, vol. 12, no. 1, pp. 59–90 (2017).

[22] Z. Szatmári, J. Oláh, and I. Majzik, "Ontology-based test data generation using metaheuristics," in *Proceedings of the 8th International Conference on Informatics in Control, Automation and Robotics*, vol. 2, pp. 217–222 (2011).

[23] C. D. Nguyen, A. Perini, and P. Tonella, "Ontology-based Test Generation for Multiagent Systems," in *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, pp. 1315–1320 (2008).

[24] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Automatic Testing of GUI-based Applications," *Software Testing, Verification, and Reliability*, vol. 24, no. 5, pp. 341–366 (2014).