Journal of Computer and Knowledge Engineering, Vol. , No.. 2021.
DOI:

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

Mahboubeh Dadkhah[1], Saeed Araban[2], and 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 involves 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.**

***Index Terms*—Automated testing, semantic web, Test adaptation, Test generation, Test ontology**

I. INTRODUCTION

Web applications are one of the mostly used software systems in our everyday life which due to their inherently evolving nature require repetitive testing of their existing and new features. Modern web applications within a domain are usually implemented in a way to implement a set of common features to be performed on a wide range of entities. For example, in educational systems domain, features like sorting or filtering are implemented for multiple entities such as presented courses, taken 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 like 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 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 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 consist 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 their attributes to interact with; and, they can 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 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 realized by semantic annotations representing the testers 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 this work can be briefly mentioned as proposing a semi-automatic reuse process for reusing test scripts. This process employs:

- 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 sound 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.

II. RELATED WORKS

In this section, we briefly review works related to the

* Manuscript received , accepted .
[1] Ph.D. student, Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran, Email: mah.dadkhah@mail.um.ac.ir
[2] Assistant professor, Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran, Email: araban@um.ac.ir
[3] Assistant professor, Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran, Email: s-paydar@um.ac.ir

proposed approach which fall into two categories: automated web application testing, and semantic web enabled testing.

A. *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 resulted in proposing various automated techniques by researchers. To reduce the cost of software testing, automated testing is useful and important [1]. During the last decade, various approaches have been proposed for software test automation.

Crawling-based techniques is one of the most studied approaches. In these techniques, 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 [2]. Moreover, they are often application-specific and result in redundant test cases. A crawler-based approach is proposed in [3] to automatically generate the GUI state model for Android applications. The results show improved coverage in comparison to the manually created models. A rule-based approach using input topic identification and GUI state comparison is proposed in [4] which represents DOM elements as vectors in a vector space formed by the words used in the elements.

Software test reuse is one of the well-known approaches for reducing test costs and improving quality in software testing domain. Fischer et al. [5], conducted an experiment in highly configurable systems to automatically generate test suites for new configurations by reusing existing tests. They propose to automatically compose test variants for new combinations of configuration options by reusing parts of the source code of existing configuration options tests that are not tested together previously. Mirzaaghaei et al. [6] define a set of heuristics to use test adaptation patterns in existing test cases and generating new test cases for evolved software. Mussa and Khendek [7], proposed a model-based testing framework for connecting different levels of testing by enabling reusability and optimization across different levels of testing. They propose to reuse test cases at one level to generate test cases of subsequent levels of testing and optimize them by relating to test cases of preceding testing levels or removing them if they are found redundant.

The impacts or significance of testers knowledge in have been investigated in the literature. A systematic literature review has been conducted in [8] to identify issues and challenges regarding human knowledge in software testing. It is concluded that effectiveness of software testing depends on the availability of testers' knowledge and it is capable of minimizing the same defects or mistakes being repeated in the next software testing projects. Another survey on application of Knowledge Management (KM) principles is conducted in software development companies [9]. Conclusions shows that applying KM in software companies can bring several benefits in terms of increase in quality of results, and reduction in cost, time and effort. Milanifard et al. [2], leverage existing crawling-based generated tests with human knowledge to extend the test suite for increasing code coverage.

Gao et al. [10] use human knowledge in the form of tester annotations to automatically repair unusable test scripts. Tester annotate the automatically generated Event Flow Diagram (EFG) and repairing transformations are used to synthesize a new test script. The results show that the proposed technique is effective and annotations reduced human cost. Milani et al. proposed leveraging existing crawling-based generated tests with human knowledge to extend the test suite for increasing code coverage.

B. *semantic web enabled testing*

In our previous work [11], we have conducted a systematic literature review to identify state of the art and benefits of semantic web enabled software testing in both industry and academia. The results show that semantic web technologies improve various activities in software testing process. Among these activities, test generation and test data generation have gained more attention which mostly rely on two significant test methods i.e., model-based and rule-based. Model-based approaches mostly used different UML diagrams.

Tao et al. [12], proposed an ontology based method for testing automated and autonomous driving functions. Automatic test case generation is performed using Combinatorial Testing. In order to improve test case generation, constraints are added to the automatically generated input models. Mekruksavanich et al. [13], proposed an ontology-based design flaw detection for object oriented software. An ontology of flaw structures is proposed to describe the knowledge in the flaw domains. To perform the detection algorithm, the source code is transformed to first order logic facts and pattern matching mechanism is applied between facts and ontology rules. Haq and Qamar [14], proposed a test case generation framework by integrating learning based methods and ontology-based requirement specification for conducting black box testing. They used learning-based testing to execute existing test cases derived from formal requirements and infer the model of system under test. Moitra et al. [15], proposed a tool called ASSERT which has a formal requirements analysis engine and helps capturing requirements. ASSERT automatically generates a complete set of test cases based on captured requirements and thus provides clear and measurable productivity gains in system development. Rule-based approaches utilized ontologies to model interactions, behaviors, EFGs, or GUI elements relations for test generation. For example, in [16] 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 is predefining which could be implemented once and then automatically reused to generate tests by building different scenarios in different business domains. Another technique is to use semantic annotations for enriching test artifacts based on an ontology. Semantic annotations have been used in [17] to automatically generate test data and test oracle. A comparison of these semantic web enabled test generation approaches based on some of their characteristics are presented in TABLE I.

TABLE I Specifications of proposed semantic web enabled test generation approaches

Study	Requirement type	Test level	Test generation		Test evaluation	
			Method	Artifact/Model	Method	Criteria
[12] Tao 2019	Functional	Higher levels	Scenario-based	Ontology-based combinatorial testing input models	Case study results from simulation platform	-
[13] Mekruksavanich 2017	Functional	Higher levels	Rule-based	Ontology of design flaws	Case study results	Precision, false positive
[14] Haq 2019	Functional	Higher levels	Learning-based	Ontology-based requirement specification	-	-
[15] Moitra 2019	Functional	Higher levels	Requirements-based	requirements written in structured natural language	Case study results from simulation	Model coverage, Structural coverage
[16] Silva 2019	Functional	Higher levels	Behavior-driven	Ontology-based user stories	Case study results	Passed and failed tests
[17] Tonjes 2015	Functional	-	Behavior-driven	FSM-based Application Behaviour Model (ABM)	Comparison with Random Selection	Failure detection rate, Computation time
The proposed approach	Functional	Higher levels	Semantic annotations	Ontology-based annotated test scripts	Comparison with human-written test scripts	Automatically generated test steps, Fault detection, Efforts

-: not mentioned.

Test data generation is another activity that semantic web technologies have been used for improving it. Mariani et al. [18], utilized the Web of data to map GUI model to the classes and predicates in the semantic knowledge-bases to generate realistic test data that match the semantics of the correlated test input fields. Test reuse is another activity that benefits from semantic web technologies which is mostly based on semantic similarity metrics. For example, Li et al. [19], used the semantic similarity between existing test cases and test requirements of the application to be tested as a basis for test reuse. Another research [20] used ontology matching technique for matching ontology of the AUT with ontology of applications which it's test cases are going to be reused.

III. The Proposed Approach

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

1) Web applications within a specific domain usually provide some common features that are 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 (main interactions between user and application) detail implementation of them may have some differences. This similarity leads to similarities in scripts 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 which corresponds 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, required interactions in different applications may be different in various systems. For example, in Digikala[4] website the Sort feature is implemented as a list of links (with each link representing one sorting option e.g., sort based on price or discount). Sorting the presented products in this application includes only one step which is clicking on one of the presented sorting options. In Amazon[5] web application, though the Sort feature is implemented as a drop-down list of sorting options. Therefore, selecting a sort option in this application includes two steps: first, opening the drop-down list, and second, clicking one of the presented sorting options. Based on this observation, a system-level abstraction is proposed to generalize a test script in a way that it can be reused and adapted for testing a similar feature on other web applications. Such scripts are called Test Interfaces which are independent from 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 presented products by choosing from a list of data options for each attribute of those products. 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

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

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

shows that structure 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 to a specific data for an 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.

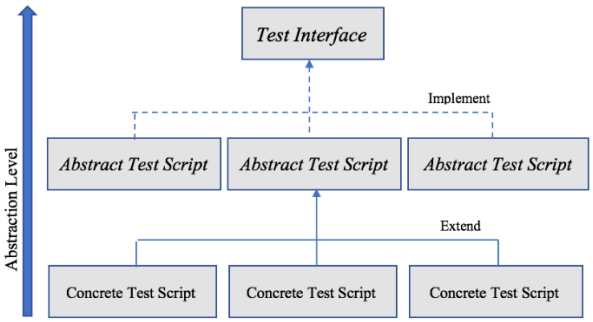

Fig. 1. The proposed three levels of test script adaptation

The proposed approach utilizes test script annotation as a mechanism to realize these levels of abstractions and make a test script independent from a specific application, entity, or test data we propose test script annotations. In the following, detailed description of test script annotations is described along with the adaptation and generation algorithms designed based on these annotations.

A. 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 a Test Script Ontology (TSO) which is shown in Fig. 2. It is an application ontology [21] and hence, does not cover all the concepts and relations in the software testing domain but only the concepts required for annotating test scripts that are used in the proposed approach. TSO is developed based on the ROoST ontology [22] 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) which were defined in the ROoSTs Testing Artifacts sub-ontology have been reused in the TSO ontology.

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:

- Category one: concepts that define a test element as a test data provider. Scripts that are written for testing web applications determine 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.
- Category two: 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.

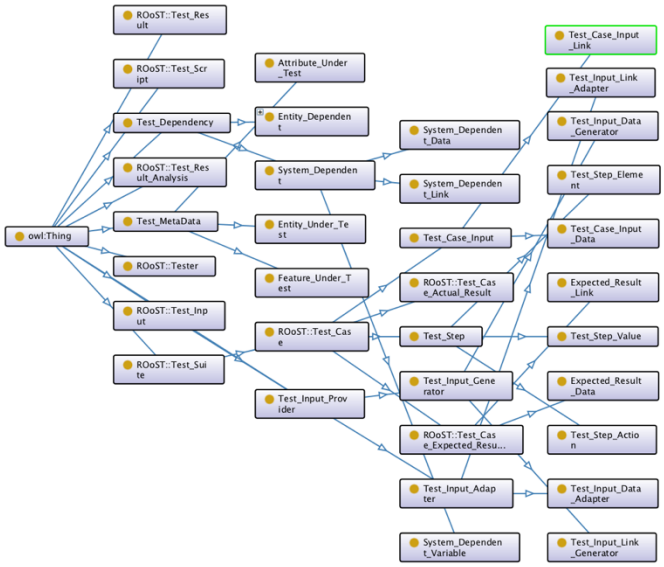

Fig. 2. The Test Script Ontology (TSO)

- Category three: concepts that determine dependency level of parts of a test script or a test step (i.e., system dependent and entity dependent). System dependent annotation indicates that part of script is similar for testing all entities of the AUT. Entity dependent in contrast, shows that part of the test script needs to be adapted for different entities of an AUT.
- Category four: concepts that determine logical steps. The tester uses these annotations to specify logical test steps based on her expertise in the testing domain and knowledge of the best test scenario 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.
- Category five: 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 to the implementation of the AUT and in some cases, may not be present.

For better understanding of the proposed annotations, two

annotated test scripts are described based on the level of independency they provide (see TABLE II and TABLE III). These scripts are written using Selenium[6] 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 as Value. In annotated test scripts, annotations of a test step always presented in the line immediately before the test step. For example, in TABLE III, line two includes annotations for the test step in line three and line four includes annotations for the test step in line five. Some annotations paired (e.g. line 2 and 6 in Table II). It is possible for a test step not have any annotation (e.g. line 8, 11, and 12 in TABLE III). Unannotated test steps can be copied unchanged by the generation and adaptation algorithms. Some test steps might have multiple annotations. If a test step has multiple annotations, the order of these annotations is not important.

The annotated script in TABLE II, represent a Test Interface. This script is written for testing login feature in Yahoo[7] web application and contains two logical steps. The first logical step (line 2-6) is for entering the username and the second (line 7-11) is for entering 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 thus, are used to annotate the Command part of a test step.

The annotated script in TABLE III, represent an Abstract Test Script. This script is written for testing Filter feature in Banimode[8] 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 that contains a list of checkbox representing names of all brands that the products are from. In this script after opening the web page (line 1), one of the checkboxes representing a filter option is clicked (line 13). Then, all of the presented products are verified to have the same brand as the selected one. 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. The provided data for expected result of new adapted or generated test scripts should be placed in this step these scripts. 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 a Test Interface.

Annotated test scripts are inputs to the proposed algorithms. The tester annotates test scripts manually; hence it is possible that tester to forget some annotations or to have some inconsistencies in annotations. Therefore, it is reasonable to perform some validations on the given annotations. A simple preprocessing is used in the experiments of this paper which includes two steps:

1. Inconsistencies in annotations is checked. For example, a test step cannot be both system-dependent and entity-dependent.
2. Missing annotations is checked. For example, if there is an *expected result provider* annotation in the script, there must also be another annotation which defines the placeholder for the expected result.

TABLE II An ATS of a Test Interface for the 'Login' feature in Yahoo web application

TS1: to test feature 'Login' 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	

The annotation preprocessing step is aimed at checking an annotated script to detect problematic issues to which the proposed approach is sensitive. If such issue existed, the tester is asked to verify the script. Since annotated test scripts have an important role in the proposed approach, it is required to describe how they are defined in this approach.

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

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

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

TABLE III An ATS of an Abstract Test Script for the 'Filter' feature in Banimode web application

TS1: to test feature 'Filter' for entity 'Shoes' based on attribute '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		

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 $(1 \le i \le K)$ is an annotated test step that 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 which the test is being done through. 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, but in the system-level adaptation, they are part of the GUI of different applications.

B. Entity-Level Test Script Adaptation

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

Definition 2 (Entity-level test script adaptation). 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 *entityLevelAdaptation* procedure and is shown in Fig. 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.

The entity-level adaptation algorithm contains two major functions for element adaptation. The first one is *GUIBasedElementAdaptation* which tries 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 since as stated by W3C standards, Ids should be unique in a web page. Therefore, for each element of test steps in ATS, if the element has an Id attribute, the DGUI is searched for an element with the exact same Id. If such an element found in DGUI, it is considered as the adapted element.

```
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 s_i ∈ ATS.TC
4:          if s_i.annotation = NULL
5:              AdaptedTS. add (s_i); DGUI ← execute (s_i, DGUI); continue;
6:          if annotation ANNOT ∈ s_i.AT_i where ANNOT = system-dependent
7:              e_i ← findElement (s_i.T_i, SGUI); e'_i ← findElement (s_i.T_i, DGUI)
8:              if e'_i != NULL and Type(e'_i) = Type(e_i)
9:                  AdaptedTS. add (s_i); DGUI ← execute (s_i, DGUI); continue;
10:         suggestedSteps ← GUIBasedElementAdaptation (s_i, 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 (s_i, 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 ∈ AdaptedTS.TC'
18:         if annotation ANNOT ∈ s'_i.AT'_i where ANNOT. type = Adapter
19:             testDataAdaptation (s'_i, ANNOT, DGUI)
20: Return AdaptedTS

1:  procedure GUIBasedElementAdaptation (s_i, SGUI, DGUI)
2:      e_i ← findElement (s_i.T_i, SGUI)
3:      for all element e'_i ∈ DGUI
4:          if e'_i.Id = e_i.Id and e'_i.type = e_i.type
5:              s_i. setTarget(e'_i); suggestedSteps.append(s_i); Return
6:          if e'_i.location = e_i.location and textualSimilarity (e'_i.associatedText, e_i. associatedText) ≥ θ
7:              s_i. setTarget(e'_i); suggestedSteps.append(s_i); Return

1:  procedure SemanticEnabledElementAdaptation (s_i, 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 (e_i, SGUIEs)
8:              DGUIR ← findSemanticRelation (e'_i, DGUIEs)
9:              if equivalent (SGUIR, DGUIR)
10:                 s_i.setTarget (e'_i); suggestedSteps.append(s_i); Return
```

Fig. 3. Entity-level test script adaptation algorithm

This phase of element adaptation is also applicable for locators that specify test data. For example, consider the test script for examining feature Filter on 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 exact test data 'jeanswest'' will be a valid test data in the adapted test script. If an adapted element couldn't be found in this phase, then the second phase will be performed.

The second phase includes searching the 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 providing a feature is normally organized in the same way to assure consistency in the web page appearance and to increase 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 feature Filter over Laptop entity based on their operating system Attribute in Digikala website. The script is similar to the one in TABLE III. 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 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 in and is performed by *SemanticEnabledElementAdaptation* function in four phases.

Phase one: 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 as 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 as it is expected that developers use the same type of elements for implementing a feature for similar attributes of various entities.

Phase two: Identifying candidates 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 $\{e_i, \{\text{SGUIEs}\}\}$. Since it is expected that the adapted element e_i' have 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 candidateElements is $\{e_i', \{\text{DGUIEs}\}\}$.

Phase three: Finding semantic relations between data in each set of data using the semantic web. In this phase, the existence of any semantic relation between sets of data is checked. First, the semantic relation between e_i and provided data options by e_i (SGUIEs) is searched (line 7) and then, the semantic relation between each candidate element e_i' and provided data options by it (DGUIEs). 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 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: Checking similarity of semantic relations between sets of data in SGUI and DGUI. For this purpose, the *equivalent()* procedure is defined. The simplest situation is when the semantic relation between two sets of elements is exactly the same i.e., exact 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 this work, the existence of owl:equivalentProperty 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 adapted element when both the structural and semantical 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 *testDataAdaptation* process is performed to adapt test data on DGUI based on the provider annotations specified by the tester (line 19).

C. 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 structure and belong to different AUTs. In this level, if the entity under test in DGUI is different from the entity in SGUI, then due to the differences in both structure and semantic of the DGUI to the SGUI adaptation is not effectual. 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 that are independent from a specific entity or its attributes (e.g., sorting).

System-level adaptation takes as input a Test Interface as a general and comprehensive scenario for testing a 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, number of required test steps in a logical test step of the $AdaptedTS$ is less than 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). 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 $k' \leq k$.

The system-level test script adaptation algorithm is described by the *systemLevelAdaptation* procedure and is shown in Fig. 4. The whole adaptation process involves three nested loops at a high level: The outer loop iterates through each logical test step LTS_j and process each logical test step as a whole (lines 4-15); The first inner loop iterates through each test step s_i from a particular logical test step $s_i \in LTS_j$ and tries to adapt it to be successfully executed on DGUI; The second inner loop iterates through each element $e'_i \in DGUI$ to find an element with maximum semantic similarity to e_i.The system-level adaptation algorithm use 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, in this level of adaptation, semantic similarity of web elements is used. This is based on WordNet [23] which is a lexical database of semantic relations between different words in a network of words.

In the proposed approach, semantic similarity of 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 is probably representing 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, semantic similarity of web elements is computed by the following formula:

$$\begin{aligned} &semanticSimilarity\ (Element_1, Element_2) = \\ &W_{id} * WNSimilarity(id_1, id_2) + \\ &W_{name} * WNSimilarity(name_1, name_2) + \\ &W_{text} * WNSimilarity(text_1, text_2) \end{aligned}$$

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 $T = \{t_1, t_2, \dots, t_m\}$ 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 $(1 \leq j \leq n)$ elements to t_i $(1 \leq i \leq m)$ elements where no t'_j is assigned to more than one t_i. The best correspondence is the one which has the largest value among all possible correspondences. Finally, value of a correspondence R is computed based on the proposed formula by Paydar and Kahani [24].

D. 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, generating 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.

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: $AdaptedTS.TC' = (s'_1, \ldots, s'_k) \in AdaptedTS$ *where* $s'_i = \emptyset$
3: $LTS[] \leftarrow$ extractAllLogicalTestSteps($ATS.TC$)
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 ($s_i.T_i$, $SGUI$)
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 $ANNOT \in s_i.AT_i$ 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 $s'_i \in AdaptedTS.TC'$
17: **if** annotation $ANNOT \in s'_i.AT'_i$ where $ANNOT.type$ = *Adapter*
18: *testDataAdaptation* (s'_i, $ANNOT$, $DGUI$)
19: Return $AdaptedTS$

Fig. 4. System-level test script adaptation algorithm

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). 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 $(s_1, \ldots, s_k)$ 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 $\{TS_1, \ldots, TS_n\}$ with a set of test data $\{TD_1, \ldots, TD_n\}$ in which $TS_x = \{TC_x, SGUI, F, E, A, AUT\}$ with TC_x $(s_1, \ldots, s_k)$ 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 [25], rule-based approaches [26], using the web of data as a source of test data generation [18], 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 like Filter the required test data (e.g., test input and expected result) are presented options in the GUI that can be extracted. Tester can enrich the test scripts with her knowledge of the AUT using annotations. Then, these annotations can be used to automatically generate test data.

IV. Evaluation

For the purpose of evaluation, a prototype of the proposed approach is implemented in Java. To assess the proposed approach, an experiment is designed and executed to address the following research questions:

RQ1: Is the proposed approach effective in test script adaptation and generation?

RQ2: Is the proposed approach efficient in test script adaptation and generation?

RQ3: Is the proposed approach successful in fault detection?

RQ4: Is the semantic web useful in providing the required data?

A. Measures

The following metrics are defined based on the provided definitions for measuring effectiveness of the proposed approach:

- PSE: The Percentage of test Steps Executable; The PSE can be calculated as NSE/TNS*100 where TNS is total number of test steps and NSE is the Number of test Steps Executable counts test steps in TC' that can execute successfully on DGUI;
- PSP: The Percentage of test Steps Preserved; The PSP can be calculated as NSP/NSE*100 where NSP is the Number of test Steps Preserved counts test steps from TC that are copied into TC' and execute successfully on DGUI; The NSP can be calculated as $|\{i : 1 \leq i \leq k \wedge s_i = [s'_i]\}|$.
- PSA: The Percentage of test Steps Adapted; The PSP can be calculated as NSA/NSE*100 where NSA is the Number of test Steps Adapted counts test steps from TC whose replacements were successfully adapted during adaptation process and can execute successfully on DGUI.

The NSA can be calculated as $|\{i : 1 \le i \le n \wedge s_i <> [\,] \wedge s_i <> [s_i']\}|$.

- PSM: The Percentage of test Steps that the proposed approach fails to automatically adapt and need Manual intervention of the tester; The PSM can be calculated as NSM/NSE*100 where NSM is the Number of test Steps that need Manual intervention.

B. Experimental Objects

Three popular Iranian web applications from the e-commerce domain are selected i.e., Digikala, Banimode and Tagmond[9], referred to as APP1, APP2 and APP3 respectively. These applications are appropriate for our study because they provide us a variety in term of complexity and also diversity of entities. Four features of these applications, with different levels of complexity are selected, which are Filter, Sort, Pagination, and Login, referred to as F1, F2, F3, and F4 respectively. Selenium[10] is used for creating ATSs and also for executing produced test scripts.

C. Experimental Subjects

The results of the proposed approach are compared with human-written test scripts using Selenium. Four undergraduate students in the software engineering major were employed to create the original test scripts. They have passed two courses (namely Software Engineering I and II) and are familiar with software testing concepts. They also have experiences in developing web based systems. They had experience with Selenium tool in their university projects. The students were given clear instructions on how to create the scripts. As an additional sanity check, all the scripts were executed on their respective applications to ensure that they were in fact executable.

D. Experimental Setup

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.

1) Independent and Dependent Variables

For this experiment, one independent variable is the experience of testers with the tool. Another independent variable of this experiment is the constraints in searching semantic web data sources. The semantic web contains various sources and it is not possible to search all of them. In this experiment, DBPedia is used as representative of the semantic web sources.

The dependent variables include the percentage of successfully executable test steps produced by the proposed approach. This includes adapted and generated test steps. The percentage of automatic operations are compared with the percentage of required manual operations by the tester.

E. Results

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

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 base test script for each feature on every application is manually created. Then the test scripts are annotated to create Abstract Test Scripts (9 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 (180 in total) with the proposed entity-level adaptation algorithm. The adapted test scripts produced by the proposed approach are compared to the test scripts written by the testers.

For measuring the effectiveness of this algorithm, the percentage of executable test steps (PSE) is reported to measure how successful the proposed algorithm is in adapting test scripts. The results of the proposed approach are shown in TABLE IV. These results also indicate the percentage of test steps that the proposed algorithm failed to adapt and hence needed manual modification by the tester. This measures the level of automation provided by the proposed approach and how much it reduced the required manual effort by the tester. Analysis of the results shows that the average percentage of executable test steps for the proposed entity-level adaptation is about 95.9%. While PSE for all features in APP2 and APP3 are high, the lowest PSE is for feature F1 in APP1. This is due to the diverse range of entities with different attributes in this application which makes automated adaptation a challenging task. However, the proposed approach successfully adapted nearly 90% of test steps.

TABLE IV Executable test steps produced by the entity-level adaptation

App	APP1			APP2			APP3		
Feature	F1	F2	F3	F1	F2	F3	F1	F2	F3
PSE (%)	89.1	97.3	93.4	97.1	98.4	99	92.5	97.8	98.9

Further, the results are compared to the human-written test scripts in terms of PSA and PSP (shown in TABLE V). PSA, indicates the percentage of successfully adapted test steps while PSP, indicates the percentage of test steps that are preserved from ATS to the $AdaptedTS$s without changing or adapting.

Since the proposed approach is a semi-automatic one, there are some test steps that the approach fails to automatically adapt and need manual intervention of the tester. The percentage of these steps for each feature of every application is also presented in TABLE V (PSM). The proposed entity-level adaptation algorithm was not able to successfully adapt a low percentage of about 4.1% of the test steps on average (PSM). Among all applications in this experiment, APP2 has the lowest PSM on average which means the automation level was the highest for this application. This is due to the consistent structure of the GUIs for different entities in this application.

[9] https://www.tagmond.com

[10] Selenium 3.17.0

TABLE V Comparison of the average PSA and PSP for the proposed entity-level adaptation algorithm

App	APP1						APP2						APP3					
Feature	F1		F2		F3		F1		F2		F3		F1		F2		F3	
Approach	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS
PSA (%)	16.6	23.4	15.7	17.7	12.3	14.1	12.1	14.2	3.4	4.5	28.3	29	12.8	17.7	6.9	8.5	24	24.8
PSP (%)	72.4	76.6	81.5	82.3	81	85.9	84.9	85.8	94.9	95.5	70.6	71	79.6	82.3	90.8	91.5	74.8	75.2
PSM (%)	11	-	2.8	-	6.7	-	3	-	1.7	-	1.1	-	7.6	-	2.3	-	1.2	-

HWS: Human-Written Scripts.
PA: Proposed Approach

Feature F1 needed the most percentage of manual intervention among all features in all applications of this experiment (7.3% on average). Thus, adapting this feature for different entities of the same application needs more manual intervention of the tester

The average percentage of preserved test steps in the two approaches are about 81.1% and 82.2% respectively which are very close. This shows that the proposed approach performs very well in recognizing the test steps that can be reused for testing the same feature on various entities in an application. This also 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. The average percentage of successfully adapted test steps in the proposed approach and human-written test scripts are about 14.6% and 17.1% respectively. The results discussed in this section provide a positive answer for the research question RQ1 for entity-level test script adaptation.

Test elements in resulted test steps are adapted using the two proposed procedures. TABLE VI 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 VI 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%

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 comprehensive enough to be able to test a similar feature on other applications. The results are shown in TABLE VII for each feature in every application. The effectiveness of this algorithm is measured in a similar way to the entity-level adaptation algorithm.

TABLE VII Executable test steps produced by the system-level adaptation

App	APP1			APP2			APP3		
Feature	F2	F3	F4	F2	F3	F4	F2	F3	F4
PSE (%)	85.7	84.2	80	80.9	94.7	73.3	86.9	94.7	86.6

The average percentage of executable test steps for features F2, F3, and F4 in all the three applications are about 84%, 91%, and 80% respectively. Percentage of executable test steps in feature F3 is more than F2 and F4. One of the main reasons is that scripts for testing these features include test steps that perform an action on a variable. The proposed approach fails at adapting test steps that include variables in their targets. Another reason is 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 resulted test scripts are compared to the human-written test scripts in terms of PSA and PSP (shown in TABLE VIII). The average percentage of test steps that need manual intervention in system-level testing (14.7%) is more than entity-level adaptation (4.1%). While the logic of test in in the two applications is similar, the structure of their GUIs is different.

The percentage of preserved test steps in the two approaches for almost all features and applications are very close which is due to that entity-less scripts for testing similar features in different systems, have more than 50% similar test steps in common. The proposed approach performs very well in recognizing the test steps that can be reused for testing a similar feature in different applications

The exception in this experiment is feature F4 which the average of its PSP for all applications in two approaches is roughly 11%. The low percentage of preserved test steps in this feature is due to the fact that most of the steps in the script were actions to be applied on an element of the GUI and these elements must be adapted to the application under test. The average percentage of adapted test steps for this feature using the proposed approach and human-written scripts are about 69.4% and 88.6% respectively. This means that about 19% of test steps needed manual intervention to be successfully adapted.

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 three 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 percentage of automatically generated test scripts with valid test data that can be executed successfully. The results for APP2 is 100% which means all the possible test scripts for APP2 are automatically generated and

a maximum test coverage is provided.

TABLE VIII Comparison of the average PSA and PSP for the proposed system-level adaptation algorithm

App	APP1						APP2						APP3					
Feature	F2		F3		F4		F2		F3		F4		F2		F3		F4	
Approach	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS	PA	HWS
PSA (%)	21.4	32.8	0	15.8	68.5	88.5	16.8	33.7	12.6	15.8	64.1	89.2	17.3	30.4	12.6	15.6	75.7	88.2
PSP (%)	64.2	67.2	84.2	84.2	11.4	11.5	64.7	66.3	82.1	84.2	9.16	10.8	69.5	69.6	82.1	84.4	10.8	11.8
PSM (%)	14.4	-	15.8	-	20.1	-	18.5	-	5.3	-	26.7	-	13.2	-	5.3	-	13.5	-

HWS: Human-Written Scripts.
PA: Proposed Approach

The results for APP3 is also about 98% of automatically generated test steps. Feature F1 in APP2 and APP3 is implemented with constant and similar attributes for all products of the system which facilitates test script generation. The results for 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 and the answer to the research question RQ1 for test generation is positive. 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) 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: Modify, which requires modification of a GUI element, test data, or a test step, and Confirm, which includes selecting, deleting or confirming suggested test data or GUI elements, and deleting an optional test step.

For evaluating efficiency of the proposed approach, cost of human intervention is measured in terms of time spent on each operation; and hence the number of operations are counted. The average number of modified, confirmed, and automatic operations for all the three experiments described above are shown in Fig. 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. Based on these results, the answer to the research question RQ2 is positive and the proposed approach is promising in terms of amount of automation it provides.

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.

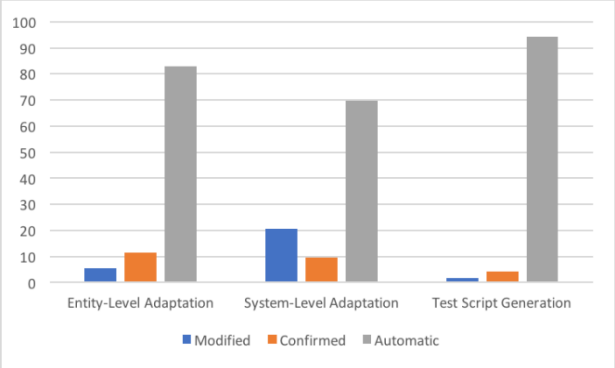

Fig. 5 Operation cost of the proposed algorithms

5) Fault detection

In this section an experiment for measuring the effect of the proposed approach on test coverage and fault detection is described. In 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 a 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 shown in Fig. 6:

Direct: is to directly create a test script for this feature in the application under test (i.e., Digikala website in this experiment) and then annotate it to be an Abstract Test Script ready for test generation process.

Indirect: is to reuse an existing Test Interface in another application (shown in grey) and adapt it to the application under test using the proposed system-level adaptation algorithm. The Banimode application also provides Sort feature based on most discount attribute. In the previous experiment the required Test Interface for this feature in Banimode application was created and annotated. Therefore, it is possible to reuse this Test Interface and adapt it for Digikala application using the proposed system-level adaptation algorithm. This adaptation required only three manual operations: two modification and one confirmation operations.

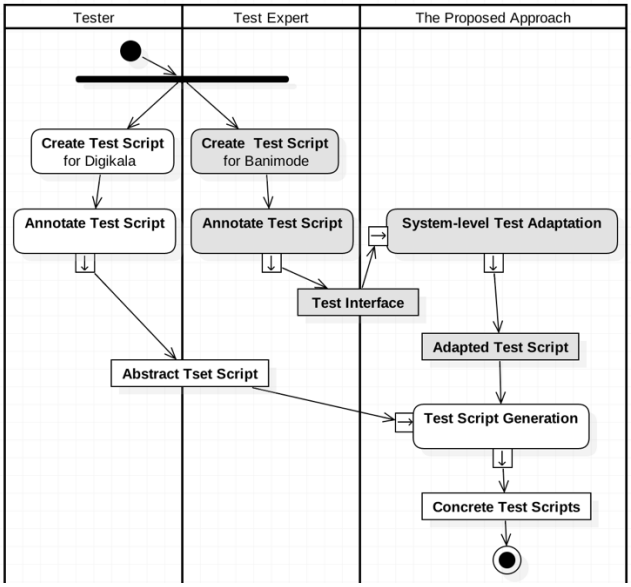

Fig. 6 Different ways of creating an entity-less script for test generation process

Both ATSs were used by the test script generation process to create multiple test scripts for a set of 20 DGUIs. To evaluate the proposed test script generation, the percentage of the faults discovered by the generated test scripts is compared to the faults discovered b human-written script. The results are shown in Fig. 7. The test scripts produced by the testers achieve the highest result, but at a price of a high effort. The proposed approach using direct annotated test scripts performs as well as human-written test scripts. These scripts outperform the proposed approach using indirect test scripts (system-level adapted scripts). These results provide a positive answer for the research question RQ3.

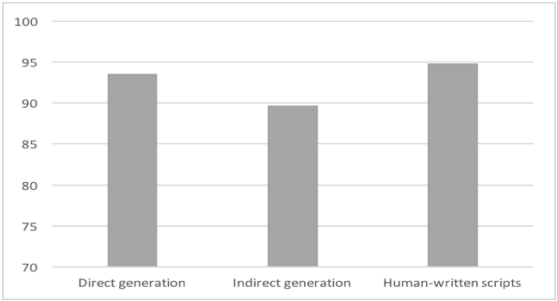

Fig. 7 Comparison of average fault detection rate

6) *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 20 GUIs are selected from two general purpose e-commerce web applications Digikala and Amazon. These two applications are used because they include a diverse range of entities and support various attributes with different data options for each entity. The results show that for 84% of the attributes, there is a subject with that attribute linking with at least one of its data options. Among 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 of searching the semantic web for these similar attributes shows that for 89% of these attributes, predicates representing them are similar. These results provide good potential for the proposed entity-level element adaptation and thus the answer to the research question RQ4 is positive.

F. Discussion

1) Annotated test scripts

The proposed adaptation and generation algorithms take as input a manually annotated test script, and as a result, the quality of the input test script can affect the results of the proposed algorithms. For example, in system-level adaptation, it is possible that a test script written for testing a feature F on application AUT1 is successfully adapted to test similar feature F' on application AUT2 while test script written for testing feature F' on application AUT2 cannot necessarily be adapted to test F on application AUT1. In the proposed approach the annotations are investigated in the preprocessing phase, but it is also helpful to assess the comprehensiveness and the quality of input test scripts.

2) Test data generation

The proposed approach uses the data options provided in the GUIs to generate scripts for testing a feature on a specific entity. This is applicable for scripts that only require valid test inputs from the possible data options that are provided in the GUI. For example, test scripts that only perform clicking a link from a list of provided links, or selecting one of the presented check boxes. For other scripts that can accept invalid test inputs, such as entering invalid username in the input text box, the proposed approach cannot automatically generate required test data. This type of test data can be provided by integrating approaches like realistic test input generation [mariani] to the proposed approach.

3) Threats to Validity

In this section, we discuss possible threats that might have affected the validity of our experiment. Threats to construct validity are mainly concerned with inaccurate measurements used in the experiments. In this work, the efficiency of the proposed approach is measured with the extent of required manual modification or confirmation by the user. Given that the effort required for modifications may be different in every situation, only counting the number of modifications may pose threats to our measurements. The measurement of effort needed for test script annotation may have the same weakness as we only measure the percentage of annotated test steps while annotating a test step with different annotation categories requires different amount of time and effort.

Threats to internal validity are mainly concerned with the

uncontrolled factors that may have affected the experimental results. In our experiments, the main threat to internal validity concerns the correctness of the input test scripts and their annotations. The input ATSs are created manually by a limited number of test experts, and therefore, it may pose threats to validity of our measurements.

Threats to external validity are associated with the generalizability of the results to other situations. A threat to the external validity of our experiment concerns the subject applications used in the experiments and the generalizability of the results to other web applications. To address this threat, we selected our experimental subjects from real-world applications with diversity in supported entities with different attributes for each entity.

V. Conclusion and Future Work

This work was motivated by two observation: first, web applications in a domain provide common features and implement these features in a similar way; second, many of the web applications provide features that have been implemented for multiple entities and their 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 on 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 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 for automatically obtaining its required information.

The initial experiments demonstrate that the proposed approach is promising. However, it can be improved in different directions which are scheduled as our future works. For instance,

- the proposed test script generation algorithm can be improved to generate new test scripts based on a correct combination of existing test scripts.
- the proposed annotations can be extended to define dependencies among test scripts. Then, these dependencies can be utilized to manage and prioritize running test scripts based on an optimal sequence to increase efficiency of resources.
- the way the semantic web sources are used by the proposed approach can be improved by identifying more promising sources and also providing more effective mechanisms for retrieving required information from these sources.

VI. References

[1] J.-W. Lin, N. Salehnamadi, and S. Malek, “Test automation in open-source android apps: A large-scale empirical study,” in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1078–1089.

[2] 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*, 2014, pp. 67–78, doi: 10.1145/2642937.2642991.

[3] C.-H. Liu and P.-H. Chen, “A Crawling Approach of Hierarchical GUI Model Generation for Android Applications,” *J. Internet Technol.*, vol. 19, no. 5, pp. 1613–1623, 2018.

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

[5] S. Fischer, G. K. Michelon, R. Ramler, L. Linsbauer, and A. Egyed, “Automated test reuse for highly configurable software,” *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 5295–5332, 2020.

[6] M. Mirzaaghaei, F. Pastore, and M. Pezzè, “Supporting test suite evolution through test case adaptation,” *Proc. - IEEE 5th Int. Conf. Softw. Testing, Verif. Validation, ICST 2012*, vol. 2, pp. 231–240, 2012, doi: 10.1109/ICST.2012.103.

[7] M. Mussa and F. Khendek, “Model-based test cases reuse and optimization,” in *Advances in Computers*, vol. 113, Elsevier, 2019, pp. 47–87.

[8] H. M. Idrus and others, “Tacit Knowledge in Software Testing: A Systematic Review,” in *2019 6th International Conference on Research and Innovation in Information Systems (ICRIIS)*, 2019, pp. 1–6.

[9] C. P. C. Maciel, É. F. de Souza, N. L. Vijaykumar, R. de Almeida Falbo, G. V. Meinerz, and K. R. Felizardo, “An Empirical Study on the Knowledge Management Practice in Software Testing.,” in *CIbSE*, 2018, pp. 29–42.

[10] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, “SITAR: GUI Test Script Repair,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 2, pp. 170–186, 2016, doi: 10.1109/TSE.2015.2454510.

[11] M. Dadkhah, S. Araban, and S. Paydar, “A systematic literature review on semantic web enabled software testing,” *J. Syst. Softw.*, vol. 162, p. 110485, 2020, doi: 10.1016/j.jss.2019.110485.

[12] J. Tao, Y. Li, F. Wotawa, H. Felbinger, and M. Nica, “On the Industrial Application of Combinatorial Testing for Autonomous Driving Functions,” in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019, pp. 234–240, doi: 10.1109/ICSTW.2019.00058.

[13] S. Mekruksavanich, “Ontology-assisted structural design flaw detection of object-oriented software,” in *The Joint International Symposium on Artificial Intelligence and Natural Language Processing*, 2017, pp. 119–128.

[14] S. Ul Haq and U. Qamar, “Ontology Based Test Case Generation for Black Box Testing,” in *Proceedings of the 2019 8th International Conference on Educational and Information Technology*, 2019, pp. 236–241.

[15] A. Moitra *et al.*, “Automating requirements analysis and test case generation,” *Requir. Eng.*, pp. 1–24, 2019.

[16] 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*, 2019, pp. 616–632.

[17] R. Tonjes, E. S. Reetz, M. Fischer, and D. Kuemper, "Automated Testing of Context-Aware Applications," in *2015 IEEE 82nd Vehicular Technology Conference (VTC2015-Fall)*, 2015, pp. 1–5, doi: 10.1109/VTCFall.2015.7390847.

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

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

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

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

[22] 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.

[23] R. Beckwith, C. Fellbaum, D. Gross, and G. A. Miller, "WordNet: A lexical database organized on psycholinguistic principles," in *Lexical acquisition: Exploiting on-line resources to build a lexicon*, Psychology Press, 2021, pp. 211–232.

[24] S. Paydar and M. Kahani, "A Semi-Automated Approach to Adapt Activity Diagrams for New Use," *Inf. Softw. Technol.*, no. June, 2014, doi: 10.1016/j.infsof.2014.06.007.

[25] Z. Szatmári, J. Oláh, and I. Majzik, "Ontology-based test data generation using metaheuristics," in *ICINCO 2011 - Proceedings of the 8th International Conference on Informatics in Control, Automation and Robotics*, 2011, vol. 2, pp. 217–222, [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-80052583055%7B%5C&%7DpartnerID=40%7B%5C&%7Dmd5=7d30797a30e6213a460d147573beb925.

[26] C. D. Nguyen, A. Perini, and P. Tonella, "Experimental evaluation of ontology-based test generation for multi-agent systems," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009, vol. 5386, pp. 60–73, doi: 10.1007/978-3-642-01338-6_5.