Generating Feature Usage Scenarios in Client-side Web Applications

Josip Maras[1], Maja Štula[1], and Jan Carlson[2]

[1] University of Split, Croatia,
josip.maras@fesb.hr, maja.stula@fesb.hr
[2] Mälardalen University, Sweden,
jan.carlson@mdh.se

Abstract. Client-side web applications are highly-dynamic event-driven GUI applications where the majority of code is executed as a response to user-generated events. Many software engineering activities (e.g. testing) require sequences of actions (i.e. usage scenarios) that execute the application code with high coverage. Specifying these usage scenarios is a difficult and time-consuming activity. This is especially true when generating usage scenarios for a particular feature because it requires in-depth knowledge of application behavior and understanding of the underlying implementation. In this paper we present a method for automatic generation of feature usage scenarios. The method is based on dynamic analysis and systematic exploration of the application's event and value space. We have evaluated the approach in a case study, and the evaluation shows that the method is capable of identifying usage scenarios for a particular feature. We have also performed the evaluation on a suite of web applications, and the results show that an increase in coverage can be achieved, when compared to the initial coverage obtained by loading the page and executing registered events.

Keywords: Web Applications, Symbolic Execution, GUI Testing

1 Introduction

The client-side of a web application is a highly dynamic, event-driven environment where features manifest at runtime, triggered by sequences of user events – usage scenarios. Specifying these usage scenarios is a difficult and time-consuming activity and in the client-side web application domain, it is made even more complicated due to the fact that the application is a result of interplay of three conceptually different languages (HTML, CSS, and JavaScript), where the most complex one – JavaScript is a highly dynamic scripting language. This makes it difficult to understand feature behaviors and to specify usage scenarios that capture the complete behavior of particular features.

Usage scenarios are most often used in web application testing. Current state of practice is that developers create tests either manually, or with tools such as Selenium[1], which enable recording and replaying usage scenarios designed to

[1] http://docs.seleniumhq.org/

test certain features. This is a time-consuming activity and automating it would offer considerable benefits. Usage scenarios can also be used for reuse – in our recent work [7] we have developed methods for identifying and extracting code and resources of client-side features based on the dynamic analysis of execution traces recorded while executing user-specified usage scenarios. This means that the quality of the extracted feature is highly dependent on the quality of usage scenarios. For this reason, automatic generation of high-coverage usage-scenarios for particular features would be beneficial.

In this work we define a method for generating usage scenarios for a particular feature in a client-side web application. The user selects parts of the page where the target feature manifests, and the process generates usage scenarios that achieve high coverage with respect to the selected parts of the page. The method is based on dynamic analysis and systematic exploration of the application's event and value space. Initial scenarios are created based on events registered during the initialization of the page, and new scenarios are added by executing and dynamically analyzing the execution of already generated scenarios. During scenario execution, all input parameters are symbolically tracked, and all event registrations, as well as all data dependencies between code expressions are logged. New scenarios are generated by modifying event input parameters, and by extending existing scenarios with registered events. Finally, the executed usage scenarios are filtered to reduce their number, with the criteria of still achieving high coverage.

We have evaluated the method on a case-study application, and the evaluation shows that the method is able to generate usage scenarios that target particular application features. We have also run the evaluation on a suite of web applications, and the evaluation shows that an increase in coverage, when compared to the straight-forward approach of loading the page and executing all registered events, is achieved by using systematic exploration of the application's event and value space.

This paper is organized as follows: Section 2 describes related work, while Section 3 presents a conceptual model of client-side web applications that helps us reason about the relationships between features and usage scenarios. Section 4 gives an overview of the feature usage scenario generation process, while Sections 5 and 6 go into more detail about generating and filtering usage scenarios. Section 7 describes the evaluation, while Section 8 presents the conclusion and possible future work.

2 Related Work

Our approach is based on client-side web application testing, where the goal is to create sequences of events that achieve high code coverage.

In [9], Mesbah et. al. describe their approach for automatic testing. The method is based on a crawler [8] that infers a state-flow graph for all client-side user interface states. New states and transitions are created by executing existing event handlers, analyzing the structure of the application and determining if it is changed enough to warrant a new state. The crawling phase is directed either with randomly generated input values or with user-specified values. Various errors are detected (DOM validity, error messages, etc.) by analyzing possible client-side user interface states.

Saxena et al. [10] present a method and a tool – Kudzu. The approach explores the application's event space with GUI exploration (searches the space of all event sequences with a random exploration strategy), and the application's value space by using dynamic symbolic execution. In the process, they have developed a string constraint solver capable of taking into account the specifics of string constraints present in JavaScript programs.

Artemis [2] is an approach for feedback directed testing of JavaScript applications from which we have derived most insights when developing our approach. The approach is based on dynamic analysis of web application execution – the application execution is monitored and all event registrations logged. New test cases are created by extending already existing tests with event registrations and by generating variants of the event input parameters. For generating new event input parameters they use randomly chosen values, and constants collected during the dynamic execution. They also introduce prioritization functions which influence the order in which generated test cases are analyzed.

None of the introduced client-side web application testing approaches enable developers to target specific client-side features, nor do they enable the filtering of generated scenarios in order to minimize the number of necessary usage scenarios. Also, in order to improve coverage, we use the systematic exploration of the application's event-space (similar to [2]) and combine it with symbolic execution (similar to [10]). On top of this, we track application dependencies by the means of a dependency graph [7], which enables us to accurately capture dependencies between different events, and to create event chains.

In the domain of testing server-side web applications, there exists the SWAT tool [1], which uses search-based testing. In their approach, random inputs to the web application are generated with additionally incorporated constant seeding (gathered by statically analyzing the source code), and by dynamically mining values from the execution. Although some parts of the approach could be adopted to fit the domain of client-side applications, their method is specially developed to deal with constraints inherent in server-side applications.

3 A Conceptual Model of the Client-side Application

In this section we present a conceptual model of client-side web applications (Figure 1) that will be used to reason about generating usage scenarios for a particular feature. A feature is an abstract notion representing a distinguishable part of the system behavior that is manifested at runtime, when a user preforms a certain sequence of actions, i.e. a usage scenario [3].

A client-side application can be viewed as a collection of visually and behaviorally distinct UI elements (or UI controls). A UI control is primarily defined in terms of its structure, but it also includes the behavior on that structure. For example, in the case study shown in Figure 3, Section 7, each marked section of the page can be considered as a UI control.

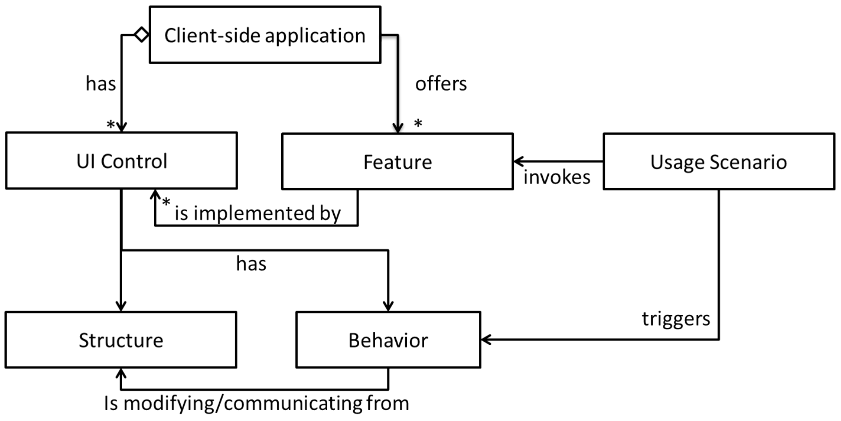


Fig. 1. Client-side web application conceptual model

A client-side application offers a number of features. Since client-side web applications are UI applications to server-side applications, a feature is manifested through a number of structural changes on the client-side and/or communications with the server-side. Because a UI control encapsulates structure and the behavior on that structure, and since features can cross-cut between different parts of the application, we define that a single feature is implemented by at least one UI control (Figure 1). A UI control implements a feature by reacting to user-generated events by modifying its structure, and/or communicating with the server from that structure. We utilize this relationship between features and UI controls – since features are abstract, and UI controls concrete, when generating usage scenarios for a particular feature, we are generating usage scenarios for the implementing UI controls.

3.1 Terminology

An event e is defined as a tuple $e = \langle h, t \rangle$, where h is an object on which the event occurs (e.g. an HTML node, or the global window or document objects), and where t is an event type. At run-time, when an event is raised it is parametrized with properties of three different types [2]: *i)* event properties – a map from strings (property names) to numbers, booleans, strings and DOM nodes, *ii)* form properties, which provide string values for HTML form fields, and *iii)* the execution environment properties, which represent values for the browser's state that can be influenced by the user (e.g. window size). A parametrized event e^p consists of an event e and parameters p associated with that event.

The goal of the process is to compute a set U of usage scenarios: $U = \{u_0, u_1, ..., u_n\}$ that achieves high coverage of a given feature. A usage scenario u_i is defined as a sequence of parametrized events $u_i = \langle {e^p}_0, {e^p}_1, ..., {e^p}_m\rangle$. A scenario u_i exercises the behavior of a given feature if every parametrized event ${e^p}_i \in \langle {e^p}_0, {e^p}_1, ..., {e^p}_m\rangle$ is related to at least one UI control that implements the feature. A parametrized event is related to a UI control if: *i)* it is called on an html node that is a part of the UI control; *ii)* it modifies the structure of the UI control; *iii)* in the case of ajax events, if there is a data dependency from the request to the structure of the UI control, *iv)* it influences the execution of an event related to a UI control.

4 Overview of the Usage Scenario Generation Process

Client-side applications are highly dynamic and event-driven, and the appropriate way of reasoning about their control-flow is through dynamic analysis. As input the process receives the source code of the application, and a set of UI control selectors (e.g. css selectors, xPath expressions) that specify the UI controls that implement the feature of interest. The process consists of two phases: *i)* generating usage scenarios, and *ii)* filtering usage scenarios (Figure 2).

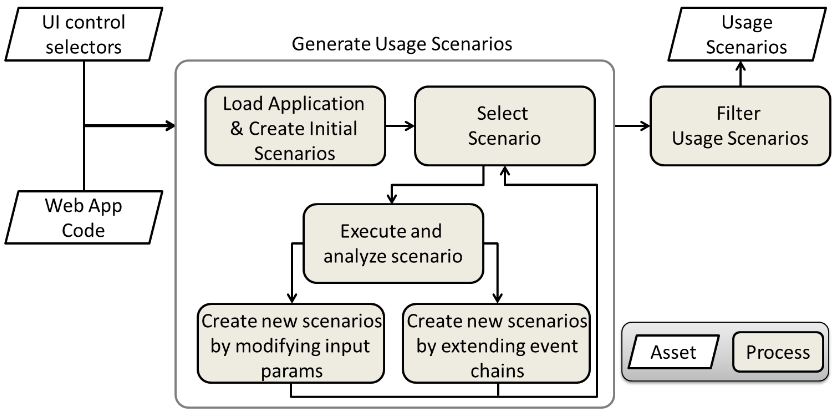

UI control selectors
Web App Code
Generate Usage Scenarios
Load Application & Create Initial Scenarios
Select Scenario
Execute and analyze scenario
Create new scenarios by modifying input params
Create new scenarios by extending event chains
Filter Usage Scenarios
Usage Scenarios
Asset
Process

Fig. 2. The process of generating feature usage scenarios

The phase of usage scenario generation starts by initializing the web page – a stage of the execution not dependent on user input. During page initialization, a number of events can be registered, and these events are the basis for the creation of initial usage scenarios. For each event registered in the initialization phase, a new usage scenario, with default event parameter values is created. Our approach then proceeds by selecting a usage scenarios, executing it, and dynamically analyzing the execution. New usage scenarios are created in two different ways: *i)* by modifying the usage scenario event input parameters – we track how the event input parameters influence the control-flow, and new usage scenarios are generated by modifying those inputs; *ii)* by extending event chains, either with new instances of previously executed events whose execution depends on the variables and objects modified during the execution of the scenario, or with newly registered events with default parameter values. New usage scenarios are created and analyzed until a certain coverage is achieved, a given time-budget expended, or a target number of scenarios have been generated.

In the second phase – usage scenario filtering – execution traces of all executed usage scenarios are analyzed, and the computed set of usage scenarios is filtered by removing scenarios that do not contribute to the feature behavior, and scenarios whose removal does not lower the overall coverage.

5 Generating Usage Scenarios

In this section we give a detailed description of how new usage scenarios are created, and for this we will use the example shown in Listing 1.1.

The example application has two features: Feature 1, implemented with the UI control defined by the first square (node with id *fc*, line 7), which consists of two behaviors: *i)* when the user clicks on the square with the left mouse button, the application subscribes to the mouse move events which change the color of the first square background depending on the position of the mouse, *ii)* counts the number of middle mouse button clicks on the first square, and outputs whether this number is even or odd; and Feature 2, implemented with the UI control defined by the second square (node with id *sc*, line 7), with a behavior: *i)* when the user clicks on the second square it outputs the current mouse position. This is an example of an event-driven application where code coverage depends both on the events raised by the user, and the properties of the raised events (e.g. which mouse button was clicked). Throughout this section we will show how the process generates usage scenarios that target the first feature.

5.1 Generating initial usage scenarios

The start of the whole process is the execution of the page loading phase with the goal of obtaining registered events which will be used as a basis for defining initial usage scenarios (Algorithm 1).

For each event registered at the end of the loading phase, the process assigns default parameters to the event (e.g. for mouse clicks this means setting the pressed button to the left mouse button, the position of the mouse to the middle of the clicked on element; setting empty strings for HTML input elements, etc.), and creates a usage scenario (u) with that parametrized event.

Example. In the example from Listing 1.1 this means the creation of two usage scenarios with one event, based on the *onmousedown* event registration from line 13, Listing 1.1 – $u_0 = \langle\langle \#fc, onmousedown\rangle, \{which : 1\}\rangle$ (left mouse button is the default button in mouse events, represented by the value 1 of the *which*

```
1  <html><head>
2   <style>
3    .c{ width: 100px; height: 100px;}
4    #fc{background:rgb(255,0,0);} #sc{background:rgb(0,0,255);}
5   </style></head>
6  <body>
7   <div id="fc" class="c"></div><div id="sc" class="c"></div>
8   <script>
9    var fc = document.getElementById("fc");
10   var sc = document.getElementById("sc");
11   var fs = document.getElementById("fs");
12   var clicks = 0;
13   fc.onmousedown = function(e) {
14    if(e.which == 1)
15     fc.onmousemove = function(e) {
16      var val = e.pageX % 256;
17      this.style.background="rgb("+val+","+val+","+val+")";
18     }
19     else if(e.which == 2)
20      if(++clicks % 2 == 0)
21       this.textContent = "Even";
22      else
23       this.textContent = "Odd";
24   }
25   sc.onclick = function(e) {
26    this.textContent = e.pageX + ";" + e.pageY;
27   }
28  </script></body></html>
```

Listing 1.1. Example application

Algorithm 1 generateInitialScenarios($webAppCode$)

```
1: $executionInfo \leftarrow$ loadPage($webAppCode$)
2: $U \leftarrow$ empty
3: **for all** $e$ : getEventRegs($executionInfo$) **do**
4:     $e^p \leftarrow$ parametrizeWithDefaults($e$)
5:     $u \leftarrow$ createEmptyScenario()
6:     $u \leftarrow$ appendEventToScenario($u$, $e^p$)
7:     $U \leftarrow$ appendScenario($U$, $u$)
8: **end for**
```

property), and based on the *onclick* mouse registration from line 25, Listing 1.1 – $u_1 = \langle\langle \#sc, onclick\rangle, \{pageX : 50, pageY : 150\}\rangle$ (the click is initially executed in the middle of the element).

5.2 Generating Scenarios by exploring the value space

In order to generate scenarios by exploring the value space, we modify event parameters by using concolic testing [4,11]. The main idea is to execute the usage scenario both with concrete (e.g. default values for the initially created usage scenarios) and symbolic values for event input parameters. During the execution all encountered control-flow branches (e.g. if statements, conditional expressions, etc.) whose branching conditions are expressions that contain symbolic variables are added to the so called path-constraint, which carries information about how the control-flow of the execution depends on the input parameters. In order to build a scenario that exercises another path through the application we have to modify the input parameters based on the path constraint. This is usually done by systematically negating the constraints that compose the path-constraint, and in our approach we use generational search [5]. Constraints obtained in this way are solved with a constraint solver, which gives new event input parameter values that exercise different execution paths. Currently we are using Choco [6] – an of the shelf constraint solver.

Algorithm 2 createByModifyingPathConstraint(u, U, *executionInfo*)

```
1: for all $invertedFormula$ : getInvertedFormulas(getPathConstraint($executionInfo$))
   do
2:     $result \leftarrow$ solveFormula($invertedFormula$)
3:     if $result \neq null$ then
4:         $\langle e_0, e_1, ..., e_n \rangle \leftarrow$ getAffectedEvents($u$, $result$)
5:         $\langle {e^p}_0, {e^p}_1, ..., {e^p}_n \rangle \leftarrow$ parametrizeEvents($\langle e_0, e_1, ..., e_n \rangle$, $result$)
6:         $U \leftarrow$ appendScenario($U$, createScenario($\langle {e^p}_0, {e^p}_1, ..., {e^p}_n \rangle$))
7:     end if
8: end for
```

Determining default parameter domains – In addition to the constraints gathered during concolic execution, some of the event parameters always fall into a certain domain (e.g. the *which* property of the mouse event handler can have only three values: 1, 2, or 3; or the mouse position parameters, such as pageX and pageY, are constrained by the position of the element the event occurs upon). For this reason, when constructing the constraint that will be sent to the solver, a constraint that captures this domain of each parameter is also added.

Example. After the execution of the first usage scenario, we study its path constraint obtained from executing the if statement from Line 14, Listing 1.1: $which = 1$. In order to cover another execution path through the application we invert that constraint and obtain ($which \neq 1$) and add the constraints inherent to the *which* property: $which = 1 \vee which = 2 \vee which = 3$. For these constraints the constraint solver obtains the result $which = 3$, and the new scenario $u_2 = \langle \langle \#fc, onmousedown \rangle, \{which : 3\} \rangle$ is generated. When we execute the usage scenario u_2 the resulting path constraint is $which \neq 1 \wedge which \neq 2$, because both the condition of the if statement in Line 14, and the condition of the if statement in line 19 were evaluated to false. By inverting these constraints we obtain two constraints: $\mathit{which} \neq 1 \wedge \mathit{which} = 2$; and $\mathit{which} = 1$, and using the constraint solver we get two solutions: $\mathit{which} = 2$ and $\mathit{which} = 1$. The solution $\mathit{which} = 1$ is discarded since the scenario with the exact parameters already exists, and out of $\mathit{which} = 2$ we obtain a new scenario $u_3 = \langle \langle \#fc, \mathit{onmousedown} \rangle, \{\mathit{which} : 2\} \rangle$.

5.3 Generating Scenarios by exploring the event-space

When generating scenarios by exploring the event-space the goal is to extend event chains, either with events newly registered during the execution of a scenario, or with already executed events that are still registered at the end of scenario execution. Algorithm 3 gives more detail about the whole process.

Algorithm 3 createByExtendingEvents(u, U, *executionInfo*)

```
1: for all $e$ : getEventRegs($executionInfo$) do
2:     if wasInstanceExecuted($e$, $U$) then
3:         for all $e^p$ : getPreviousParametrizations($e$, $U$) do
4:             if connectionExists($executionInfo$, $e^p$) then
5:                 $u_n \leftarrow$ createCopy($u$)
6:                 $u_n \leftarrow$ appendEventToScenario($u_n$, $e^p$)
7:                 $U \leftarrow$ appendScenario($U$, $u_n$)
8:             end if
9:         end for
10:    else
11:        $u_n \leftarrow$ createCopy($u$)
12:        $u_n \leftarrow$ appendEventToScenario($u_n$, parametrizeWithDefaults($e$))
13:        $U \leftarrow$ appendScenario($U$, $u_n$)
14:    end if
15: end for
```

After the execution of a scenario the process traverses all events that are still registered at the end of the execution. If the event has already been executed (at least one parametrization of that event already exists in previously executed scenarios) then all execution logs of those events parametrizations are traversed. During the execution of each scenario we build a dependency graph [7] which captures the dependencies between code constructs that exist in a scenario. The insight that we use here is: there is a potential connection between an event and a scenario if the scenario modifies variables and/or objects on which the control-flow of the event, either directly, or indirectly, depends on (influences the branching conditions). If a connection exists between the execution info of the parametrized event and the execution info of the current scenario, then a new scenario is created by appending the parametrized event to the parametrized events from the current scenario. If the event has not yet been executed, then the process is similar to the process of generating initial usage scenarios – the newly registered event is parametrized with default parameters, and a new scenario is created by appending the parametrized event to the events from the current scenario.

Example. When analyzing the execution of the u_0 scenario, a new event, which has not been executed so far, is registered in Line 15, Listing 1.1 – $\langle \#fc, onmousemove\rangle$. This leads to the creation of a new usage scenario: $u_4 = \langle\langle$ *#fc*, *onmousedown*, $\{$*which*: 1$\}\rangle$; $\langle$*#fc*, *onmousemove*, $\{$*pageX*: 50, *pageY*: 50$\}\rangle\rangle$. If we also study the process after the execution of $u_2 = \langle\langle$ *#fc*, *onmousedown*, $\{$*which*: 2$\}\rangle\rangle$ scenario, we can see that the event $\langle \#fc, onmousedown\rangle$, $\{$*which*: 2$\}$ writes to the variable *clicks*, created outside of the event context, at line 20, Listing 1.1. That same variable influences the control flow of the event (there exists a data dependency from the variable *clicks* to the if statement condition) – u_2 is dependent on itself – a new scenario u_5 is created: $u_5 = \langle\langle$ *#container*, *onmousedown*, $\{$*which*: 2$\}\rangle$; $\langle$*#container*, *onmousedown*, $\{$*which*: 2$\}\rangle$.

5.4 Prioritizing Scenarios

The algorithms described in the previous sections create new usage scenarios by systematically exploring the event and value space of the application. This means that the number of generated scenarios considerably grows with application complexity. For this reason we determine the next scenario that will be executed and analyzed based on the following procedure: if there is a non-analyzed scenario created by exploring the value space, or a scenario whose last event has not so far been executed, the process selects it. If there are no such scenarios, i.e. only the scenarios created by extending the event chain with already executed events are available, then select the next scenario randomly with the following prioritization function:

$$P = 1 - \frac{\sum\limits_{i=0}^{m} cov(e_i)}{m+1}$$

The formula is based on the intuition that executing scenarios with events that have already achieved high code coverage is likely to be less useful than executing scenarios with events with low coverage [2]. After the execution of every scenario, for every function visited during the evaluation of each event e, we recalculate the branch coverage achieved so far. We then use the prioritization function to guide the random selection of the next usage scenario that will be executed and analyzed. In the prioritization function: *cov* represents event branch coverage achieved so far.

6 Filtering Scenarios

In order to achieve high coverage, the process generates a number of scenarios. However, we are typically interested in obtaining a minimal number of scenarios that still achieve the same coverage. The main idea of this part of the process is to remove events that are not related to the UI controls that implement the feature (see Section 3.1), and to reduce the number of scenarios based on scenario coverage.

Algorithm 4 filterUsageScenarios(U, *selectors*)

```
1: for all $u_i \in U$ do
2:     if notRelatedToFeature($u_i$, $selectors$) then
3:         $U \leftarrow$ removeScenario($U, u_i$)
4:     end if
5: end for
6: $jointCoverage \leftarrow$ getJointCoverage($U$)
7: for all $u \in$ sortDescendingByNoOfEvents($U$) do
8:     if canScenarioBeRemoved($u$, $jointCoverage$) then
9:         $jointCoverage \leftarrow$ removeScenarioCoverage($jointCoverage$, $u$))
10:        $U \leftarrow$ removeScenario($U, u$)
11:    end if
12: end for
```

For every executed scenario, the process checks whether the scenario is related to the specified UI controls (Section 3.1) – if it is not, the scenario is filtered away. The process then calculates joint scenario coverage, which is a map that shows, for each code expression, how many scenarios have executed that expression. Then, all scenarios are traversed in descending order, starting from the scenario with the longest event chain. For each scenario, the algorithm checks whether the joint coverage would remain the same if the expressions executed by the scenario would be removed. If so, the scenario is removed from the set of scenarios, and its coverage from *jointCoverage*.

Example. In the example application, the scenario generation phase has generated the following six scenarios:

– $u_0 = \langle\langle \#fc, onmousedown\rangle, \{which : 1\}\rangle;\ cov_0 = \{9-15, 25\}$
– $u_1 = \langle\langle \#sc, onclick\rangle, \{pageX : 50, pageY : 150\}\rangle;\ cov_1 = \{9-13, 25, 26\}$
– $u_2 = \langle\langle \#fc, onmousedown\rangle, \{which : 3\}\rangle;\ cov_2 = \{9-14, 19, 25\}$
– $u_3 = \langle\langle\ \#fc,\ onmousedown,\ \{which\text{: } 2\}\rangle\rangle;\ cov_3 = \{9-14, 19, 20, 21, 25\}$
– $u_4 = \langle\langle\ \#fc,\ onmousedown,\ \{which\text{: } 1\}\rangle;\ \langle\#fc,\ onmousemove,\ \{pageX\text{: } 50,\ pageY\text{: } 50\}\rangle\rangle;\ cov_4 = \{9-17, 25\}$
– $u_5 = \langle\langle\ \#fc,\ onmousedown,\ \{which\text{: } 2\}\rangle;\ \langle\#fc,\ onmousedown,\ \{which\text{: } 2\}\rangle;\ cov_5 = \{9-14, 19, 20, 21, 23, 25\}$

First all scenarios are traversed in order to remove the ones that do not contribute to the feature. In this case, this means the removal of scenario u_1 because it neither occurs on, nor does it modify the selected UI control (#*fc*). Next, a joint coverage for the remaining scenarios is calculated. Here, we will discuss in terms of code lines, but the algorithm in general works on AST nodes. Joint coverage, from the perspective of executed lines, for the remaining scenarios u_0, u_2, u_3, u_4, u_5 is: 9-14→5, 15→2, 16-17→1, 19→3, 20→2, 21→2, 23→1, 25→5.

First we process the scenario u_5, which can not be removed from the set because it is the only scenario that executes line 23. Similarly, u_4 can not be removed because no other scenario executes lines 16 and 17. Scenario u_3 can be removed, because all of its lines are executed by at least one other scenario. After the removal of u_3 the joint coverage is: 9-14$\rightarrow$4, 15$\rightarrow$2, 16-17$\rightarrow$1, 19$\rightarrow$2, 20$\rightarrow$1, 21$\rightarrow$1, 23$\rightarrow$1, 25$\rightarrow$4. Similarly, u_2 and u_0 can also be removed.

7 Evaluation

We have performed two types of evaluation: *i)* on a case study application, where we study how the process is able to generate feature usage scenarios, and *ii)* on a suite of web applications, where we study the coverage the process was able to achieve when generating test cases. All results were obtained with the Firecrow tool[1] which implements the algorithms described in this paper.

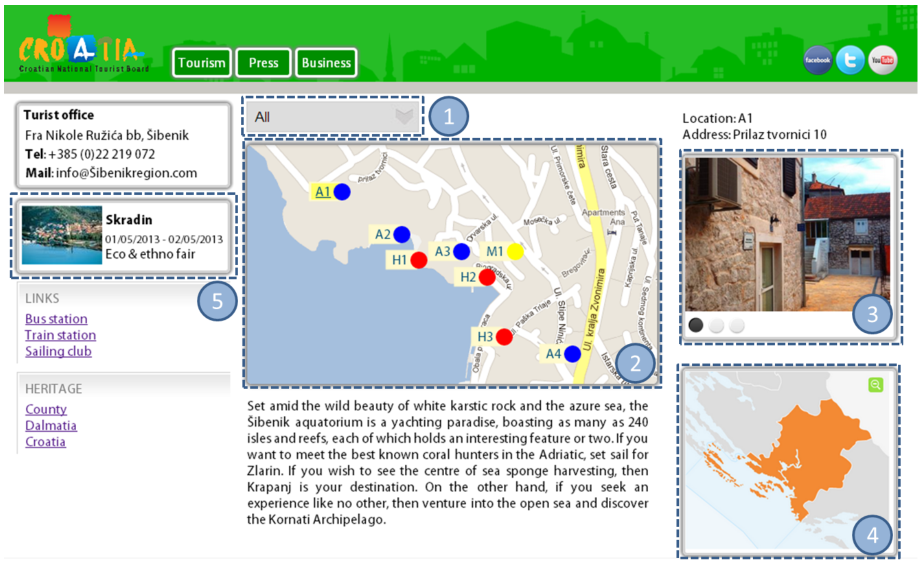
7.1 Generating Feature Usage Scenarios – a case study

Consider the example application shown in Figure 3 which represents a tourist information application that enables the user to: *i)* toggle between different types of accommodation (by using the select menu marked with 1, or by pressing keyboard keys: e.g. A – Apartments, or H – hotels), *ii)* to select map locations (marked with 2) with mouse clicks which will change the information and photos displayed in the photos section (marked with 3); *iii)* to toggle between different photos (marked with 3) by clicking on buttons, or by pressing keyboard buttons (e.g. 1 for the first photo, 2 for the second photo); *iv)* to toggle between different county map zoom levels (marked with 4) by clicking on the county map; *v)* to automatically cycle between different event information (marked with 5).

The example application has three distinct high-level features: *i)* selecting the map location and viewing its information (sections marked with 1, 2, and 3); *ii)* toggling between different county map zoom levels (marked with 4); and *iii)* viewing event information (marked with 5). Even in the case of these relatively simple features, specifying usage scenarios with high coverage is a time-consuming activity that requires in-depth knowledge of application behavior and the understanding of the underlying implementation. For example, a developer who wants to specify a usage scenario that exercises the complete behavior of the first feature has to be aware of different ways the location can be selected (by mouse clicking on the location point in the map, by changing the type of displayed locations through the select box, or by pressing keyboard keys), and of different ways the photos (marked with 3) can be toggled (either with mouse clicks on different buttons, or with keyboard presses).

We have initialized the process for each of the features with the results shown in Table 1. For each feature, the process was able to achieve full coverage (in general this does not have to be the case), and it was successful in generating usage

[1] https://github.com/jomaras/Firecrow



Turist office
Fra Nikole Ružića bb, Šibenik
Tel: +385 (0)22 219 072
Mail: info@Šibenikregion.com

Skradin
01/05/2013 - 02/05/2013
Eco & ethno fair

Location: A1
Address: Prilaz tvornici 10

Set amid the wild beauty of white karstic rock and the azure sea, the Šibenik aquatorium is a yachting paradise, boasting as many as 240 isles and reefs, each of which holds an interesting feature or two. If you want to meet the best known coral hunters in the Adriatic, set sail for Zlarin. If you wish to see the centre of sea sponge harvesting, then Krapanj is your destination. On the other hand, if you seek an experience like no other, then venture into the open sea and discover the Kornati Archipelago.

Fig. 3. Case study application

Table 1. A case study of generating feature usage scenarios

Feature	All Scenarios	Kept Scenarios	Gen. events	User events
Feature 1	25	12	12	12
Feature 2	25	1	2	2
Feature 3	25	1	1	1

scenarios that target specific UI controls. The table shows how many scenarios the process generated in order to achieve full coverage (column All Scenarios), how many scenarios were kept after the filtering process (Kept Scenarios), and how many events in total the filtered scenarios have (Gen. events). The table also shows the minimum number of events, we were able to find, to achieve full coverage. In this application, the process was able to generate feature scenarios which in total have the minimal number of events we were able to determine by studying the application code. In general, since scenarios can be picked randomly from the set of generated scenarios, the generated sequences of events in all analyzed scenarios are not necessarily minimal.

7.2 Generating usage scenarios for the whole page

For this experiment we have evaluated the approach by generating 100 tests for a suite of web applications, most of them obtained from 10k and 1k JavaScript challenges[1]. The code of all applications, and the generated scenarios can be ob-

[1] http://10k.aneventapart.com/ and http://js1k.com/

tained from: *www.fesb.hr/∼jomaras/download/usageScenarioGenerator.zip*. Table 2 shows the results. For each application it shows the lines of code (LOC), statement coverage that can be achieved just by loading the page (L-Cov), coverage that can be achieved by executing the initially registered events with default parameters (I-Cov), coverage the process was able to achieve (A-Cov), and statement coverage that we were able to achieve by constructing event chains manually (M-Cov). The table also shows how many scenarios were kept after the filtering phase (Kept), and how many events have the final generated scenarios together. On average, the process is able to achieve additional 17,6% coverage when compared to the coverage achieved by loading the page and executing all registered events.

Table 2. Experiment results for generating 100 usage scenarios that target whole pages: LOC - Lines of Code, L-Cov – statement coverage on page load, I-Cov – statement coverage on executing initially registered events, A-Cov – Achieved Coverage, M-Cov – Maximum coverage we were manually able to achieve, Kept – Number of remaining scenarios after filtering, Gen. Events - total number of generated events

App	LOC	L-Cov	I-Cov	A-Cov	M-Cov	Kept	Gen. Events
Snake	223	57,5%	63,7%	90%	98,36%	3	14
Prism	401	56,5%	70,5%	82,5%	94%	7	17
Jump	313	63,2%	65,8%	70,32%	98,23%	2	11
Agency	303	35,1%	57,4%	100%	100%	12	12
Slider	128	45,6%	71,7%	77,17%	86,41%	3	9
Minesweeper	175	59,1%	85,2%	93,91%	95,97%	6	7
3DMaker	385	18,9%	31%	42,59%	94,2%	2	8
floatwar	457	17,1%	45%	64,47%	93,7%	2	9
snowpar	352	19%	61,8%	81,5%	88,42%	19	22
3DModel	2567	17,8%	55,6%	81,8%	81,8%	24	24

8 Conclusion

Usage scenarios that execute application features with high coverage are used in many software engineering activities, such as testing, or reuse. Manually specifying these usage scenarios is a time-consuming activity, and automating it would bring considerable benefits. In this paper we have presented an automatic method for generating feature usage scenarios. The method works by systematically exploring the event and value space of the application. In order to create high-coverage scenarios we utilize techniques such as symbolic execution, and dependency tracking. In order to reduce the number of generated scenarios, we analyze the relationships between the scenarios and features, and remove all non-related scenarios. We also subsume scenarios based on their coverage. We have evaluated the method on a case study application, and the evaluation shows that the method is able to generate scenarios that target certain application features. We have also performed the evaluation on a suite of web applications, and the results show that an increase of code coverage, when compared to the initial coverage achieved simply by loading the page and executing all registered events, can be achieved.

For future work we plan to expand the usage scenario process to generate tests which take into account the server-side code, and we plan to perform the evaluation on a larger set of web applications. Since one motivation for developing this approach was to support the identification of feature code by automatically generating high-coverage usage scenarios, we plan to utilize this method in the development of an automatic feature identification process (by extending [7]).

References

1. N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *Automated Software Engineering, ASE'11. 26th International Conference on*, pages 3–12. IEEE Computer Society, 2011.
2. S. Artzi, J. Dolby, S.H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Software Engineering, ICSE 2011, 33rd International Conference on*, pages 571–580. ACM, 2011.
3. T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29(3):210–224, 2003.
4. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
5. P. Godefroid, M. Y. Levin, D. Molnar, et al. Automated whitebox fuzz testing. NDSS, 2008.
6. N. Jussien, G. Rochart, and X. Lorca. The choco constraint programming solver. In *Open-Source Software for Integer and Contraint Programming, CPAIOR08 workshop on*, 2008.
7. J. Maras, J. Carlson, and I. Crnkovic. Extracting client-side web application code. In *World Wide Web, WWW'12, 21st international conference on*, pages 819–828. ACM, 2012.
8. A. Mesbah, E. Bozdag, and A. van Deursen. Crawling ajax by inferring user interface state changes. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on*, pages 122–134. IEEE, 2008.
9. A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *Software Engineering, IEEE Transactions on*, 38(1):35–53, 2012.
10. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.
11. K. Sen, D .Marinov, and G. Agha. *CUTE: a concolic unit testing engine for C*, volume 30. ACM, 2005.