Meta-Heuristic Generation of Robust XPath Locators for Web Testing

Maurizio Leotta, Andrea Stocco, Filippo Ricca, Paolo Tonella

Abstract:

Test scripts used for web testing rely on DOM locators, often expressed as XPaths, to identify the active web page elements and the web page data to be used in assertions. When the web application evolves, the major cost incurred for the evolution of the test scripts is due to broken locators, which fail to locate the target element in the new version of the software. We formulate the problem of automatically generating robust XPath locators as a graph exploration problem, for which we provide an optimal, greedy algorithm. Since such an algorithm has exponential time and space complexity, we present also a genetic algorithm.

Digital Object Identifier (DOI):

`http://dx.doi.org/10.1109/SBST.2015.16`

Meta-Heuristic Generation of Robust XPath Locators for Web Testing

Maurizio Leotta[1], Andrea Stocco[1], Filippo Ricca[1], Paolo Tonella[2]
[1] Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
[2] Fondazione Bruno Kessler, Trento, Italy
maurizio.leotta@unige.it, andrea.stocco@dibris.unige.it, filippo.ricca@unige.it, tonella@fbk.eu

***Abstract*—Test scripts used for web testing rely on DOM locators, often expressed as XPaths, to identify the active web page elements and the web page data to be used in assertions. When the web application evolves, the major cost incurred for the evolution of the test scripts is due to broken locators, which fail to locate the target element in the new version of the software. We formulate the problem of automatically generating robust XPath locators as a graph exploration problem, for which we provide an optimal, greedy algorithm. Since such an algorithm has exponential time and space complexity, we present also a genetic algorithm.**

Keywords*—*Web Testing, XPath Locators Robustness, Web Element Locators, DOM-based Locators, Fragile Test.

I. Introduction

DOM locators are widely used in the definition of automated test cases for tools such as Selenium WebDriver. While different tools support different ways to specify DOM locators, without loss of generality we can map all such specifications to XPaths and reason only about XPath locators. During software evolution, such locators may need to be repaired, since they no longer point to the right web page element and this happens to be one of the major costs in web testware evolution [1].

In our previous work, we developed Robula [2], a tool that implements a heuristic to generate robust XPath locators. While preliminary results with Robula are quite promising, the heuristic approach implemented in Robula suffers two main problems: (1) there may exist cases in which it does not return any locator in a reasonable amount of time, because of the exponential complexity of the heuristic it implements; (2) it may return a suboptimal locator even when it has enough time to compute it, because it is not guaranteed to return the optimal locator in case of convergence.

In this paper, we formulate the robust locator generation problem as a graph search problem, so as to make it treatable by means of meta-heuristic algorithms (e.g., genetic algorithms), instead of relying on ad-hoc heuristics (as the one implemented by Robula). Then we present two algorithms: (1) an exact, greedy algorithm, that returns the global optimum, but requires exponential time and space and for this reason (2) an approximate genetic algorithm that tries to quickly find (at least) sub-optimal solutions.

II. The Robust Locators Generation Problem

Locator. Given a target web element e from the web page DOM D, we call a *locator* for e every XPath expression l such that query(l,D) = {e}, i.e., *every XPath expression l able to select uniquely e in D*. Fig. 1, reports, on the upper left corner, a simplified DOM D and a target web element e (the first <p> highlighted in red). Usually, for each web element e, there are many locators. Fig. 1 reports several XPath locators for e underlined in green.

XPath expressions are composed by DOM element names (i.e., tag names) and predicates. The former can be explicitly specified in the XPath (e.g., //p) or left unspecified (e.g., //*). The latter are used to limit the selection only to the DOM elements that contain specific values. In particular, we consider position values (e.g., //p[1]), attribute values (e.g., //p[@id='name']), and texts contained in the DOM elements (e.g., //p[text()='John']). We also consider XPaths containing conjunctions of such predicates (e.g., //p[@id='name' and text()='John']. Let L_e={l : query(l,D)={e}} be the set of all XPath expressions that are locators for e and that are built out of tag names and predicates. Each XPath expression can be one or more levels long. We define the *length* of an XPath as the number of its levels, i.e., it is the number of '/' or '//' symbols (e.g., the length of //p is 1, while for //html/p it is 2). In the following, we refer to the action of specifying a tag name, adding a predicate, or adding a new level to the current XPath expression as a "specialization step", executed by a specialization transformation.

Robust Locator. In previous studies [2], we noticed that the most robust XPath locators, i.e., the ones that are more likely to survive to the evolution of the DOM D, are those that contain as few names, predicates and levels as possible. Indeed, any unnecessary information contained in a locator could vary in the next evolved DOM and thus make the locator broken (an information can be considered unnecessary if the expression still remains a locator after removing it). To quantitatively estimate the complement of robustness, we associate a *fragility coefficient* FC to each XPath expression. The XPath //* has FC=0. Every specialization step, that adds information to //* increases the FC. The maximum fragility is reached by the full absolute XPath, i.e., the expression containing all the names, attributes, texts and position values of the target element e and its ancestors up to the root html (see for instance the expression //html[1]/p[text()='X' and @class='a'][1] in Fig. 1).

In our previous studies [2], we found that adding different predicates to an XPath leads to different levels of fragility. For instance, adding a predicate on the position value makes, usually, the XPath much more fragile than adding a predicate on the ID attribute. Indeed, the position values heavily depend on the structure of the page (e.g., adding a field in a form breaks all the locators for the subsequent fields, if such locators use position values) while the ID is usually meaningful and

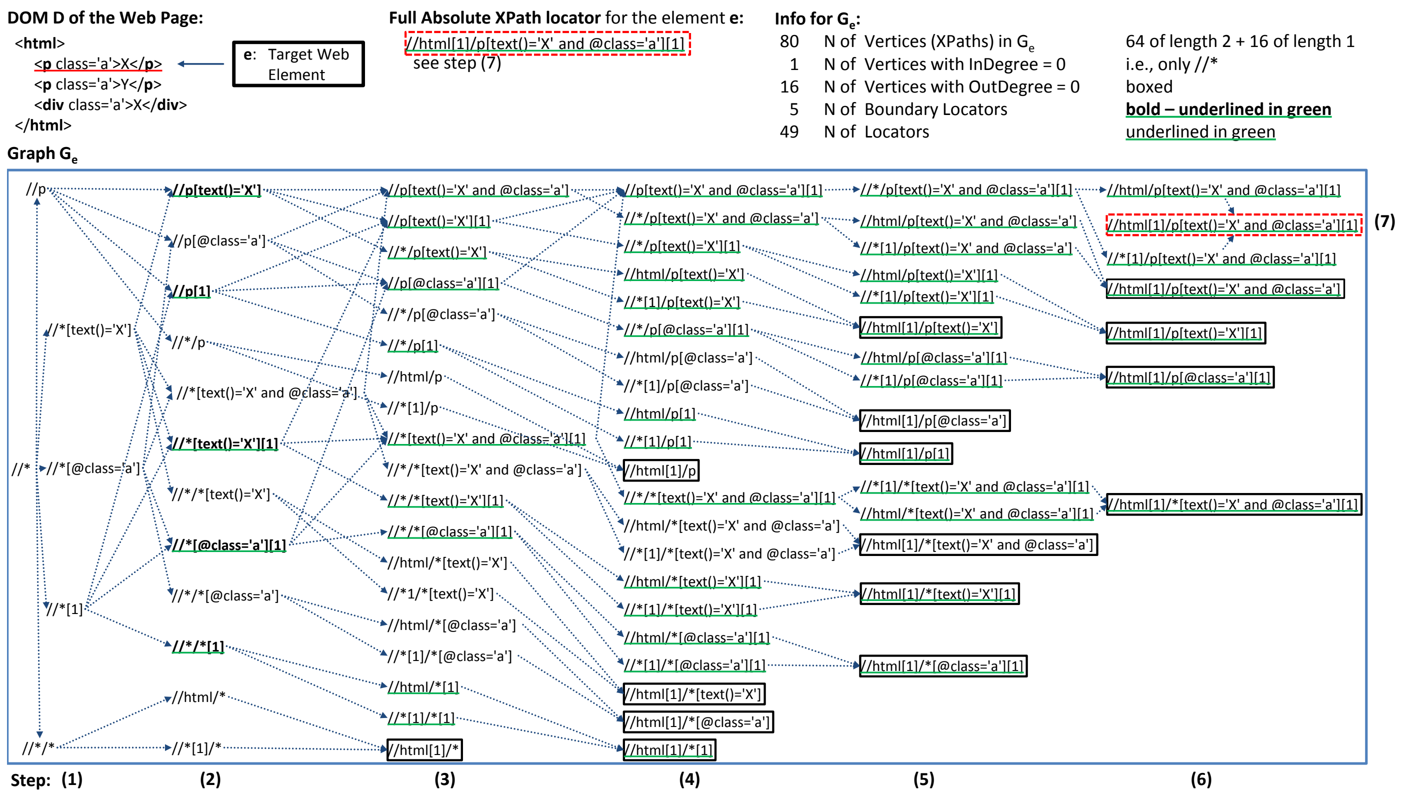
DOM D of the Web Page:
<html>
<p class='a'>X</p>
<p class='a'>Y</p>
<div class='a'>X</div>
</html>
e: Target Web Element
Full Absolute XPath locator for the element e:
//html[1]/p[text()='X' and @class='a'][1]
see step (7)
Info for G_e:
80 N of Vertices (XPaths) in G_e 64 of length 2 + 16 of length 1
1 N of Vertices with InDegree = 0 i.e., only //*
16 N of Vertices with OutDegree = 0 boxed
5 N of Boundary Locators bold – underlined in green
49 N of Locators underlined in green
Graph G_e
//p
//*[text()='X']
//*
//*[@class='a']
//*[1]
//*/*
//p[text()='X']
//p[@class='a']
//p[1]
//*/p
//*[text()='X' and @class='a']
//*[text()='X'][1]
//*/*[text()='X']
//*[@class='a'][1]
//*/*[@class='a']
//*/*[1]
//html/*
//*[1]/*
//p[text()='X' and @class='a']
//p[text()='X'][1]
//*/p[text()='X']
//p[@class='a'][1]
//*/p[@class='a']
//*/p[1]
//html/p
//*[1]/p
//*[text()='X' and @class='a'][1]
//*/*[text()='X' and @class='a']
//*/*[text()='X'][1]
//*/*[@class='a'][1]
//html/*[text()='X']
//*1/*[text()='X']
//html/*[@class='a']
//*[1]/*[@class='a']
//html/*[1]
//*[1]/*[1]
//html[1]/*
//p[text()='X' and @class='a'][1]
//*/p[text()='X' and @class='a']
//*/p[text()='X'][1]
//html/p[text()='X']
//*[1]/p[text()='X']
//*/p[@class='a'][1]
//html/p[@class='a']
//*[1]/p[@class='a']
//html/p[1]
//*[1]/p[1]
//html[1]/p
//*/*[text()='X' and @class='a'][1]
//html/*[text()='X' and @class='a']
//*[1]/*[text()='X' and @class='a']
//html/*[text()='X'][1]
//*[1]/*[text()='X'][1]
//html/*[@class='a'][1]
//*[1]/*[@class='a'][1]
//html[1]/*[text()='X']
//html[1]/*[@class='a']
//html[1]/*[1]
//*/p[text()='X' and @class='a'][1]
//html/p[text()='X' and @class='a']
//*[1]/p[text()='X' and @class='a']
//html/p[text()='X'][1]
//*[1]/p[text()='X'][1]
//html[1]/p[text()='X']
//html/p[@class='a'][1]
//*[1]/p[@class='a'][1]
//html[1]/p[@class='a']
//html[1]/p[1]
//*[1]/*[text()='X' and @class='a'][1]
//html/*[text()='X' and @class='a'][1]
//html[1]/*[text()='X' and @class='a']
//html[1]/*[text()='X'][1]
//html[1]/*[@class='a'][1]
//html/p[text()='X' and @class='a'][1]
//html[1]/p[text()='X' and @class='a'][1]
//*[1]/p[text()='X' and @class='a'][1]
//html[1]/p[text()='X' and @class='a']
//html[1]/p[text()='X'][1]
//html[1]/p[@class='a'][1]
//html[1]/*[text()='X' and @class='a'][1]
(7)
Step: (1) (2) (3) (4) (5) (6)

Fig. 1. XPath expressions Graph

carefully chosen by developers. Thus, the *various specialization transformations increase the value of* FC *differently*.

Specialization Transformation. Fig. 2 reports the considered specialization transformations. They are similar to those used heuristically by our tool ROBULA to generate robust locators [2]. Let us consider the XPaths obtained by starting from //* and applying such transformations exhaustively, in all possible orders and combinations. Let X_e=$\{x$: query(x,D) $\supseteq$ $\{e\}\}$ be the resulting set of XPath expressions. Each XPath $x \in X_e$ may select one or more elements in the DOM D, including always the target element e (see how the transformations work in Fig. 2); hence, $L_e \subseteq X_e$. Starting from //*, each transformation adds a constraint, consisting of tag name or predicate, thus restricting the set of selected elements to a subset of those selected by the original XPath. Hence, the exhaustive exploration leading to X_e is monotonically decreasing in the query output. Moreover, all possible combinations of tag names and predicates are tried in the exploration. As a consequence, no element of L_e can be missed. This means that the set of specialization transformations in Fig. 2 is *complete*, i.e., every unique locator $l \in L_e$ that includes only tag names and predicates for a DOM element e can be generated by repeated application of the transformations.

XPaths Generation Graph. The generation of the set X_e can be described as the generation of a directed graph $G_e(V,E)$, where the vertices in V are the XPath expressions in X_e (i.e., V=X_e) and the oriented edges E represent the execution of the various specialization transformations. Since the derivation process is monotonically decreasing, the graph cannot contain loops; hence, it is a DAG (Directed Acyclic Graph). Fig. 1 shows the DAG G_e for an element of a simplified DOM (the target element <p> is underlined in red in the DOM). The only vertex with in-degree = 0 is //*. All locators L_e for e are underlined in green in the graph G_e. The *number of vertices* in G_e depends on: (1) h the number of levels between e and the root of the DOM (i.e., the length of full absolute locator for e) and (2) the cardinality of each P_i, the set of predicates (i.e., attributes, text and position) that are applied at the i-th level of the full absolute locator for e. More specifically:

$$|V| = \sum_{i=1}^{h} 2^{(\sum_{k=1}^{i} |P_k|)+i} = |X_e| \quad (1)$$

For instance, in Fig. 1, for the target element <p> we have a total number of levels $h = 2$. Counting backward from <p>, these two levels are associated with two sets of predicates: P_1={text()='X', @class='a', position=1} and P_2={position=1}. The XPath expressions that contain only one level (leftmost part of Fig. 1) start either with //* or //p. They are completed with the addition of a subset of the predicates in P_1 (including the case of the empty set, e.g., in the case of //p). Since there are 2 possible starting XPaths, completed by any of the $2^{|P_1|}$ subsets of P_1, in total there are $2 \cdot 2^{|P_1|} = 2^{|P_1|+1} = 2^{3+1} = 2^4 = 16$ possible XPaths. The XPath expressions that contain two levels start with //* or //html, possibly completed with the addition of any subset of predicates from P_2, and followed by the next level, starting with /* or /p and completed with any subset of predicates from P_1. Hence, there are $2 \cdot 2^{|P_2|} \cdot 2 \cdot 2^{|P_1|} = 2^{(|P_1|+|P_2|)+2} = 2^{(3+1)+2} = 2^6 = 64$ possibilities. In total, $|V| = 16 + 64 = 80$.

Given a target element e, the problem of finding the locators for e that minimise FC may require to visit, in the worst case, the entire graph G_e. As apparent from Equation (1), the size of graph G_e is exponential in the number of predicates and in the number of levels. Indeed, in the running example, having 4 predicates and 2 levels leads to a graph G_e with more than $2^{4+2} = 2^6$ vertices.

Defined:

– **w** = the XPath expression to specialize, e.g., $//td$

– **N** = the length (in levels) of **w**
e.g., $//td \Rightarrow$ **N**=1; $//*/td \Rightarrow$ **N**=2;

– **E** = the list of ancestor elements of target element **e** in the considered DOM, starting and including **e**
e.g., $[td, tr, table, body, html]$
E.get(2) returns the element tr

The transformations work as follows:

– **transfAddName** replaces the $*$ in the initial $//*$ with the tag name of the element **E**.get(**N**)
e.g., $//*/td \rightarrow //tr/td$

– **transfAddPredicate** adds the predicates (one at time) of the element **E**.get(**N**) to the highest (i.e., the **N**-th) level in **w**. The same predicate is not inserted twice.
e.g., $//tr/td \rightarrow //tr[@name = \text{'data'}]/td$

– **transfAddLevel** adds $//*$ at the top of **w** (iff **N** < **E**.length())
e.g., $//tr/td \rightarrow //*/tr/td$

Note: the position value that must be used might change depending on the element being selected, which can be either a $*$ or an actual tag name (e.g., td). In fact, in the presence of siblings of different types, the position for $*$ ranges over all such siblings, while the position for a specific tag name ranges only over all siblings having such tag name.

– **transfAddPredicate**: when adding the position value in the XPath locator **w**, the transformation inserts the correct position for locating the element **L**.get(**N**), depending on the use of a $*$ or of the actual tag name (e.g., td).

– **transfAddName**: if the unspecified tag name "*" is constrained by a position value ($//*[2]$), the transformation updates also the position value (e.g., $//*[2]/b \rightarrow //div[1]/b$).

Fig. 2. Specialization Transformations

To determine the fragility coefficient FC for each locator, we associate the edges of G_e with a weight that depends on the specialization transformation represented by the edge (e.g., adding a predicate with a position has a higher weight than adding a predicate with an ID attribute). We define function $f_c : V \rightarrow \mathbb{R}$, that associates each expression $x \in V$ with the fragility coefficient FC $\in \mathbb{R}$, computed as the sum of the edge weights encountered when navigating the graph backwards, from x to //*. Although multiple paths can be found between the root of G_e and the locator of interest, it can be easily shown that they all provide the same value for FC. It can be also noticed that FC can be equivalently computed by simply scanning the names, predicates and levels composing the XPath expression of the locator and summing up the respective weights.

Since specialization transformations add tag names, predicates or levels, they always increase the fragility coefficient FC associated with the current XPath expression by a certain amount (the transformation weight). Hence, *the value of* FC *in each specialization path is strictly monotonically increasing*.

Our **goal** is *to determine one or more elements of L_e that have minimum value of* FC, i.e., the *optimal* (i.e., most robust) candidate locator(s). The vertices of G_e can be partitioned into two. One partition contains the XPath expressions that are not locators, whereas the second contains all the locators for e (the 49 XPaths underlined in green in Fig. 1). The *locators with minimum* FC *are all at the boundary between these two partitions* (the 5 XPaths in bold and underlined in green in Fig. 1). In fact, a locator v that does not belong to the frontier between the two partitions has all its predecessors w that must necessarily be also locators (otherwise v would be at the boundary). Since the transition $w \longrightarrow v$ is associated with a non zero weight, w has lower FC and v cannot be an optimal locator. However, some locators in the boundary could not be optimal locators (e.g., //p[text()='X'] and //p[1] have different FC values if we considered the position more fragile than the text).

III. Algorithms for Generating Robust Locators

A. Greedy Robust Locator Algorithm

The idea behind the greedy algorithm for finding a locator with the lowest value of the FC, is to visit each vertex of the graph G_e starting from the one having the lowest value of FC (i.e., //*) and then moving to the next unvisited vertex, choosing the one associated with the lowest value of FC. Based on function $f_c : V \rightarrow \mathbb{R}$, we can order the set V by increasing values of FC. Different expressions in V may have the same value of FC; hence, multiple permutations of V may satisfy the order. The greedy algorithm visits the graph G_e following one of the possible permutations of V satisfying such order. Algorithm 1 reports a possible implementation.

Algorithm 1: Greedy Robust Locators Algorithm

```
   Input:
   D: DOM of the web page
   e: target web element
   Result:
   l: a robust locator
 1 begin
 2     $L_V$ := $\emptyset$
 3         // the visited vertices list
 4     $L_G$ := "//*"
 5         // the generated vertices list (in increasing order of FC). Each
              unvisited vertex of $G_e$ is first generated and then visited
 6     while $L_G \neq \emptyset$ do
 7         $x$ = $L_G$.removeFirst()
 8         $L_V$.addLast($x$)
 9         if $query(x, D) = \{e\}$ then
10             // $x$ is a locator for $e$
11             return $x$
12         else
13             // $x$ is not a locator for $e$
14             $L_X$ = specialize($x$)
15                 // the list of XPaths generated from $x$ by applying all the
                      admissible transformations of Fig. 2
16             $L_X$.removeElemOf($L_V$)    // removing already visited vertices
17             $L_X$.removeElemOf($L_G$)    // removing already generated vertices
18             foreach $k \in L_X$ do
19                 FC$_k$ = $f_c(k)$
20                 insert $k$ in $L_G$ preserving the order
21                     // i.e., all the elements in $L_G$ preceding $k$ must have a
                          FC<=FC$_k$ while the following ones have FC>FC$_k$
```

The algorithm is ensured to terminate, since in the worst case it returns the full absolute XPath. In fact, repeated applications of the specialization transformations will eventually generate the full absolute XPath, consisting of all the levels with tag names and including all the needed predicates (in case of tag ambiguity, the predicate on the position value is always available to ensure disambiguation). The algorithm is also ensured to return a global optimum (i.e., an XPath with minimum FC). In fact, the list L_G is kept ordered, so that the first encountered locator, returned by the algorithm at line 9, is ensured to have lower or equal FC than the other locators and non-locators in L_G. Non-locators will have an increased FC if transformed into locators by any specialization, while the other locators are equivalent to or worse than the returned one.

The worst-case complexity of the algorithm is exponential in the number of predicates and in the number of levels, because the number of vertices of G_e is exponential in the number of predicates and in the number of levels (see Equation (1)). Indeed, in the worst case the algorithm has to store all the visited vertices in L_V; thus it is exponential in space. Moreover, it might have to visit all the edges in G_e (line 14, each possible specialization step), hence in the worst case it is also exponential in time. The exponential complexity of the greedy algorithm justifies the use of meta-heuristics like genetic algorithms, since on instances of the problem that involve a large number of levels and predicates the greedy approach could not terminate in a reasonable amount of time.

B. Genetic Robust Locator Algorithm

In this section we provide a detailed specification of all the elements needed to define a genetic algorithm (GA) that solves the robust locator generation problem.

Chromosome: an XPath expression (e.g., //p[text()='X']).

Population: a set of chromosomes, i.e., a set of XPath expressions.

Gene: a portion of the chromosome that can be mutated and altered. In the case of robust locator generation, the genes are the various constructs composing the XPaths (i.e., in our case, tag names, predicates, levels).

Mutation: a genetic operator that alters one or more gene values in a chromosome from its initial state. In the case of robust locator generation, we define mutations as the specialization transformations reported in Fig. 2 and their inverses. Thus, it is possible to add or remove a tag name, a predicate (on attributes, text and position values) and add/remove a level. Applying the inverse of a specialization transformation reported in Fig. 2 means navigating the graph backwards (note that not all the specialization transformations can be applied to every XPath). For instance, the expression //html/p can be mutated in //html[1]/p or //*/p, see Fig. 1.

Crossover: a genetic operator that creates new chromosomes by combining a pair of "parent" chromosomes. In the case of robust locator generation, we consider as crossover points the XPath level separators '/'. Thus, in case of two XPaths one level long, crossover cannot be applied. In case of XPaths two levels long or more, the one-point crossover can be applied. For a given crossover point (e.g., the '/' separating first from second level in the two XPaths), all data beyond that point in both XPaths are swapped between the two parent chromosomes. For instance, from the full absolute XPath //html[1]/p[text()='X' and @class='a'][1] ("Step 7" in Fig. 1), and //*/* (the last XPath of column "Step 1"), it is possible to generate in one step: //*/p[text()='X' and @class='a'][1] (the first XPath of column "Step 5") and //html[1]/* (the last XPath of column "Step 3"). Similarly, with XPaths that are three or more levels long, it is possible to apply also the two-point crossover operator. Given two fixed level separators in the XPaths, it swaps everything between the two points of the parent chromosomes, producing two child chromosomes.

Fitness function: We apply two different fitness functions to two subpopulations of the current population *Pop*. Chromosomes are partitioned into *Pop'* and *Pop"*. *Pop'* contains all XPaths from *Pop* that are not locators. These need to be further specialized to select only element e. The best chromosomes in *Pop'* are those that select a small number of DOM elements, including e. *Pop"* contains all XPaths from *Pop* that are locators for e. Among them, we give preference to those with lower FC. Hence, the fitness function *fit* for the XPath expressions x is differentiated for the two subpopulations *Pop'* and *Pop"* as follows:

$$\mathit{fit}(x) = \begin{cases} |\mathrm{query}(x, D)| & \mathrm{query}(x, D) \neq \{e\} \\ f_c(x) & \mathrm{query}(x, D) = \{e\} \end{cases} \tag{2}$$

For the chromosomes of *Pop'* (i.e., the XPaths x for which $\mathrm{query}(x, D) \neq \{e\}$), the fitness function *fit* requires to minimize the number of web page elements selected by x. For the chromosomes of *Pop"* (i.e., the locators of e for which $\mathrm{query}(x, D) = \{e\}$), the fitness function *fit* requires to minimize the fragility coefficient FC.

Hence, trying to minimize the fitness function *fit* means two different things for the two subpopulations *Pop'* and *Pop"*: (1) to add constraints to the expressions that are not locators (*Pop'*); and (2) to simplify as much as possible the locators, trying to remove all unnecessary information (*Pop"*). In practice, both forces push the XPath expressions in the current population towards the boundary between locators and non-locators in the graph G_e, i.e., towards the frontier where the most robust locators are (i.e., those with the lowest value of FC).

The ***initial population*** Pop_0 can be obtained by: (1) generating the full absolute XPath and adding it to Pop_0; (2) adding each XPath obtained by removing each level from the full absolute XPath; (3) adding //*. Thus, the initial population contains both expressions that are locators (e.g., the full absolute XPath) and others that are not locators (e.g., //*). Then the algorithm proceeds iteratively as follows: (1) it computes the fitness function *fit* for each chromosome x; (2) it selects the chromosomes that have lower *fit* values, separately for the two subpopulations *Pop'* and *Pop"* (e.g., taking the best 50% of the locators and the best 50% of the non-locators); (3) it produces the next generation of chromosomes by applying the genetic operators crossover and mutation. The algorithm terminates when the full graph G_e has been explored or the timeout has been reached. Upon termination, it returns the set of locators with minimal FC generated during the evolutionary process.

IV. Conclusions and Future Work

We have formulated the problem of generating robust XPath locators as a graph search problem, for which we have provided an exact and a meta-heuristic solution, based on a genetic algorithm. In our future work, we will implement the algorithms and compare them with each other, with ROBULA, and with the multi-locator [3], both in terms of accuracy and scalability.

References

[1] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Visual vs. DOM-based web locators: An empirical study. In *Proceedings of 14th International Conference on Web Engineering (ICWE 2014)*, volume 8541 of *LNCS*, pages 322–340. Springer, 2014.

[2] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Reducing web test cases aging by means of robust XPath locators. In *Proceedings of 25th International Symposium on Software Reliability Engineering Workshops*, ISSREW 2014, pages 449–454. IEEE, 2014.

[3] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using multi-locators to increase the robustness of web test cases. In *Proceedings of 8th International Conference on Software Testing, Verification and Validation*, ICST 2015. IEEE, 2015.