Cost Analysis and Tradeoffs in Regression Testing using FSMWeb

Seif Azghandi
Department of Computer Science, University of Denver, 2360 S. Gaylord Denver, CO 80208 USA
Email:sazghand@cs.du.edu

***Abstract*—Web applications have become software commodities of choice due to advances in internet, and wireless communications. Web applications need to be tested during new development, and thereafter during maintenance when presented with changes. Models can be used to represent the desired behavior or to represent the desired testing strategies and testing environment. FSMWeb is a black box model-based testing, and regression testing approach for web applications. This paper elaborates, extends the previous works on FSMWeb's features, and introduces patching as a regression testing technique in which test cases are repaired (patched) versus being fully regenerated. Patching may lead to cost saving when regression testing. These enhancements lead to construction, and analysis of cost models used to compare various regression testing approaches resulting to selection of a regression testing approach that achieves a favorable (reduced) cost. The determination of favorable cost is subject to number of assumptions, and tradeoffs.**

***Index Terms*—FSMWeb, regression testing, patching, cost model, tradeoffs.**

I. Introduction

The FSMWeb test model was first introduced by Andrews et al. [1] as a test model for web applications. The authors chose to use the Finite State Machine (FSM) as a test model due to the nature of most web applications. Namely, the application services are fulfilled via navigation among the web pages. FSM has inherent limitations including a tendency towards state explosion [2]. FSMWeb is constructed with two main features to overcome FSM's limitation: 1) the creation of FSMs in a *hierarchical* structure, and 2) the annotation of transitions (edges) among the FSMs with input parameters and predicates followed by an action (e.g. press of a button). The use of predicates achieves a higher rate of compression than the annotation of inputs on transitions (edges) used in Extended FSM (EFSM) [3]. The two aforementioned features in FSMWeb achieve both a high rate of state compression and scalability [4]. The presence of an action at the end of a string of input parameters implies a transition to the succeeding state. In this way, the model *only* represents the main functionality, also known as the *happy path*. The handling of failures and error recovery are subjects of additional enhancements to FSMWeb and is left for future research. The FSMWeb data structure is comprised of nodes (states) and edges (transitions). The FSMWeb is bounded by a start web page node (W_s) and a terminating web page node (W_e) and serve in test case generation. W_s and W_e demarcate FSMWeb's boundaries; W_s and W_e could be dummy or physical nodes [1]. Except W_s and W_e, all other nodes are either *cluster* or *Logical Web Page* - LWP nodes (A LWP is comprised of a state and set of inputs followed by an action (e.g., press of a button). A physical web page contains one or may contain many LWPs) [1], [5]. Figures 1a, 1b, 1c, and 1d represent FSMWeb in various levels of abstraction (i.e. Abstract FSM - AFSM [1]), refinement, or in between. The use of subsystems (ss) and components (c) terms is semantic and implies the existence of subsystems in a more abstract level in relation to components. However, structurally both subsystems and components are cluster nodes that when decomposed result in a set of nodes (e.g. subcluster, and/or LWP) and edges that connect the nodes. In this paper, use of clusters achieves state compression and therefore scalability as a result [4]. There could exist many cluster hierarchies in FSMWeb based on complexity of a web application. For example, W_s could represent a physical portal page for a commercial airline web application, and analogously W_e could represent a web page with an end-of-application exit message; AFSM could represent the entire commercial *airline operations*; *reservation*, *marketing*, *maintenance* could be represented by subsystems; *flight schedule*, *pricing*, *seating* could represent components of *reservation* subsystem, and *flight reservation* (by inputting the traveler's information) could be represented by one or many LWPs.

The cluster nodes includes two *dummy* entry (n_{DI}) and exit (n_{DO}) nodes. The n_{DI} and n_{DO} nodes are means of keeping the various (including graphical) representations of the model in a Single Entry/Single Exit (SESE) [6] configuration and aids in the test generation process in the different levels of hierarchy. The justification for use of dummy nodes is explained in the later sections. For a given thread of hierarchy, the successive (recursive) decomposition of cluster nodes results to a set of LWP node(s) and connecting edge(s). The successive decomposition of *all* the cluster nodes in the model results in the most refined representation of the model in which, except for the W_s and W_e nodes, all other nodes are LWP and are connected through edges. Edges represent the input parameters, the predicates, are followed by an action, and are always represented in the most refined level (not subject to decomposition). Edges are classified as either 1) *external*, or 2) *internal* (inspired by [7]). An edge is

Manuscript received September 10, 2013.

doi:10.4304/jsw.9.7.1790-1801

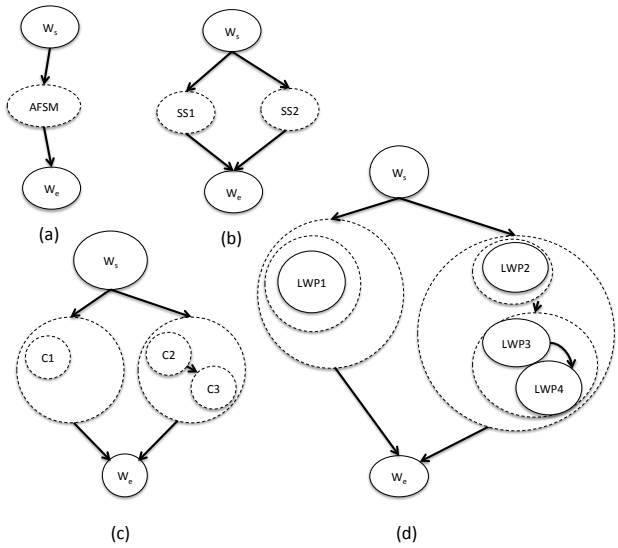

Fig. 1: The FSMWeb model depicted at subsystem, component, and LWP levels

classified as an external edge when the edge is incident on a cluster node (i.e. either entering into a cluster node or exiting out of a cluster node). If an edge is not external then it is internal (i.e. an edge that connects two LWP nodes). In the most refined representation of FSMWeb (i.e. when all nodes are of type LWP, except the W_s and W_e nodes), classification of edges into *external* or *internal* [7] aids in determining the cluster node boundaries. The *external* and *internal* classification of edges is used to assess the scope of the impact on test cases in the presence of change to edges. Except for W_s and W_e nodes, each and every hierarchical level of FSMWeb could be comprised of a mix (heterogeneous) of cluster and LWP nodes.

FSMWeb could also be used for maintenance work. Maintenance may involve fixing errors (*corrective*), enhancing features (*perfective*), or modifying or incorporating new features as required (*adaptive*) [5]. Regardless of the nature of the changes, web applications need to be regression tested to ensure that the intended functionalities are fulfilled. The main goal in regression testing is to reveal as many faults as possible in the SUT (System Under Test) while reducing the required resources (personnel, costs, time to delivery, etc.). In large applications, *selective regression testing* is preferred in comparison to *retest-all* [8] since the former achieves a reduction in the required resources assuming that the number of affected test cases due to changes are small in comparison to the total number of test cases. Andrews et al. [5] applied *classification* to FSMWeb as a vital step in *selective retesting* in order to classify the test cases that are no longer valid (*obsolete*), the test cases that have not been altered but need to be re-run to gain confidence for their validity (*retestable*), and the test cases that have not been affected (*reusable*) (reusable classification was first introduced by Leung et al. [9]). The classification of test cases are followed by executing the retestable test cases that results to two additional groups of obsolete and reusable test cases based on whether the execution of retestable test cases fails or succeeds. Both the obsolete test cases resulted from classification step and the test cases resulted from executing the retestable test cases need to be replaced by replacement test cases to maintain the existing functionalities. Also, there may be a need for generation of test cases for new functionalities [10], [11]. A salient feature of this paper is a use of patching as a regression testing scheme. Patching focuses on repairing obsolete test cases resulting from a test case's classification. In this paper, the prevailing assumption is that *test case generation (or regeneration) incurs higher cost (effort) in comparison to patching under certain circumstances (minor changes)*. The preceding is a major assumption by which four different regression testing cost models are compared in this paper. The cost models are constructed for regression testing approaches using *classification*, *classification* followed by *patching*, *retest-all*, and *brute force*. The computations, and analysis (comparisons) of these costs are subject to certain assumptions, and/or tradeoffs.

The contribution of this paper is as follows: 1) formalization of the FSMWeb and formalization of test case generation (these works are missing in [1] and are vital steps in carrying forward), 2) introduction of patching under certain conditions as an alternative to full test case regeneration, and 3) formulation of cost models through which the aforementioned regression testing approaches are compared resulting to making a decision to select the most favorable regression testing approach. This paper is organized as follows: Section 2 describes the issues that sets this work apart from the other related works. Section 3 formalizes FSMWeb model, describes FSMWeb's model characteristics (properties), and devise a set of rules for its construction and maintenance. Section 4 explains and formalizes test generation using FSMWeb's model. Section 5 describes test case classification in regression testing using FSMWeb. Section 6 introduces test case patching as an alternative to full test case regeneration under some conditions. Section 7 proposes cost models used in regression testing in order to examine tradeoffs through cost analysis by comparing the *selective retesting* using *classification*, *patching*, *retest-all*, and *brute force* regression testing approaches. Section 8 draws conclusions and suggests future work. Section 9 pays tribute to individuals that helped in embellishing this paper.

II. Related Works

This section includes some of the recent related web-based application testing/regression testing works in a chronological order starting with the oldest papers.

Leung and White [9] proposed a cost model to compare selective retesting versus retest-all techniques. Malishevsky et al. [12] presented a cost-benefit and trade offs for regression testing concentrating on test case selection, test suite reduction, and test case prioritization.

The Authors' technique is *evaluative*(versus predictive) and is not clear if their approach includes regression testing for web applications. The works by Binkley [13]–[15] addressed reduction in regression testing cost using dependence graph/slicing, considering semantic differences/similarities, and employing algorithms (e.g. common execution patterns). Sant et al. [16] used logged user data to construct web application's models using *Markov assumptions* resulting to automatically generating test cases. Neto et al. [17] presented a survey of model-based testing approaches classifying testing levels, software domains, level of automation, etc. Do and Rothermel [18], [19] proposed a cost-benefit model for regression testing techniques among which is the cost of repairing obsolete test cases (CO_r). However, the works in [18], [19] do not entail the make up of the CO_r. Park et al. [20] presented historical value based approach for effective regression testing using prioritization that is cost aware. The authors used Average Percentage of Faults Detected ([21]) as a measure correctness for their approach. Marchetto et al. [22] proposed a state-based testing for Ajax-built web applications. This work involved manipulation of Document Object Model (DOM) of the web page and abstraction in deriving a state model. Test cases were resulted from the state model by deriving them from semantically interacting events. Alshahwan et al. [23] proposed automated session data repair in web applications' maintenance work. The authors used white-box testing approach to detect changes followed by repair actions. Memon's work [24] concentrated on Graphical User Interface (GUI) regression testing. The author used *repairing transformation* to salvage obsolete uses cases. The author's work involved white-box approach for test repairs. The preceding work was followed by Huang et al. [25] in repairing test cases using Genetic Algorithm (GA). Briand et al. [26] presented regression test selection automation hinging upon UML designs. The authors' approach was based on design changes and the three test categories of reusable, retestable, and obsolete. Mesbah et al. [27] proposed a technique to cope with non-deterministic behavior of Ajax (web) application when regression testing. The authors suggested to use a set of oracle comparators, template generators, and visualizations for test failures. Sprenkle et al. [28] used logged user behavior to construct a usage-based model of web application navigation. This work resulted to creating abstract test cases. Artzi et al. [29] presented an automated test generation for JavaScript by constructing a feed-back framework. Marback et al. [30] used a regression testing approach for web applications written on PHP. The approach used dependence graphs employing abstract syntax trees to patch test cases considering numeric and string input values. Sampath et al. [31] proposed formalization work of test case prioritization, reduction/minimization, and selection using hybrid criteria (remix of Rank, Merge, and Choice). The authors claimed outperformance of hybrid criteria over the constituent criteria making up the hybrid criteria.

The work in this paper is different from the preceding works despite of the overlapping topic coverage. This difference is due to hierarchical feature of FSMWeb which advocates state compression and hence scalability as a result. In regression testing, this feature may help to localize (by considering external edges) the affected nodes and edges under certain conditions and therefore help to isolate the test cases that are affected by changes. Finally, test case patching (versus fully regenerating the obsolete test cases) for minor changes and under special circumstances may contribute to cost saving when regression testing.

III. FSMWeb Test Model

The FSMWeb's data structure model is a Hierarchical predicate FSM (HFSM) test ready model for web applications and is defined as follows:

$$HFSM = \{M_l\}_{l=0}^{n} \quad (1)$$

where M represents a model (i.e. FSM), and the subscript l = 0, 1, 2, 3, ..., n indicates the level where the FSM is located with M_0 representing the most abstract level (i.e. AFSM = M_0 [1]). The models (M_l) consist of the following parameters

$$M_l = < S, P, T >$$

where S is the *set of states* (s ∈ S), s_0 ∈ S is the initial state, s_f ∈ S is the final state, and P is the *set of predicates* (p ∈ P). Ammann et al. [6] has defined *predicate* as an expression and a set of constraints that evaluates to a boolean value (i.e. *true* or *false*). In this paper, we only model valid actions (See "*happy path*" in section I). So, each and every predicate *must* evaluate to *true*.

$$\text{P} = < I, ACT >$$

, I is a set of inputs (i ∈ I), and ACT is a set of actions (act ∈ ACT). An action (act) *triggers* a transition given properly formatted inputs (see [1] for input constraints and types).

$$\text{p} = i_1, i_2, i_3, ..., i_n, act$$

T is defined as *a transition function* where T: S × P → S (the set of edges, E can be constructed via the transition function T). The formal definition of FSMWeb's data structure *excludes* the output parameter (O) since the states (S) *cover* the outputs (O ⊆ S). Hence, states (S) serve as *partial test oracles* (i.e. $s_i \times \{i_{i1}, i_{i2}, i_{i3}, ...\} \rightarrow s_{i+1}$, is considered a partial test oracle since otherwise s_{i+1} would have not occurred). States (S) are comprised of Logical Web Pages (LWPs) and clusters (CL), where S = LWP ∪ CL and LWP ∩ CL = ∅.

A. FSMWeb as a Partitioned Test Ready Model

In FSMWeb, low-coupling among subsystems (cluster nodes) or among components (cluster nodes) [1], [5] is the desirable criterion to maintain minimality in test sequence

generation work. Partitioning is not always possible since, in some cases, individual web pages may represent a single function (e.g., login page). However, a single function that is represented by a subsystem could be viewed by the process of applying a *chain of abstraction* [7].

The partitioning decomposes a web application into its constituent elements of web pages (*page*), components (*comp*), and subsystems (ss). The essential feature to enforce at the time of partitioning is to ensure that elements (i.e. pages, components, and subsystems) are *disjoint* with respect to one another both in the peer levels and through out the hierarchical levels. Stated differently, there should not exist any replicated nodes (cluster or LWP) in FSMWeb model. The enforcement of disjoint feature at the time of partitioning is a factor in keeping the number of the nodes in a manageable quantity, and consequently leading to FSMWeb scalability. An FSMWeb partitioning is specific to a web environment in contrast to AH-Graphs [7] in which the nodes' types for a given level of hierarchy are homogenous and the refinement of the graph could go on in an unpredictable manner.

B. FSMWeb Properties

In this section, we devise a set of FSMWeb properties by restating the preceding sections. The following properties must be observed during FSMWeb's model construction and maintenance.

1) **Graph Type:** FSMWeb is a Deterministic Annotated/predicated Directed Graph (DADG). Annotations/predicates on edges describe a set of constraints for the transitions.
2) **Model Boundaries:** The starting page W_s and the terminating page W_e demarcate the model boundaries. W_s and W_e may be represented by physical web pages or could be dummy nodes [1]. W_s may have *in/out* edges while W_e has only *in* edges. A *full* test path is defined as sequence of one or many edge-node-edge, bounded by W_s and W_e.
3) **Reachability:** Except for the W_s and W_e, all other nodes must be reachable from W_s and must ultimately lead to W_e (i.e. FSMWeb cannot contain disconnected subgraphs/orphan nodes).
4) **FSMWeb's Contingent Node-Existence Relationship:** Since the successive decomposition of FSMWeb, except the W_s and W_e, must be entirely comprised of LWPs and edges (transitions), hence, the presence of a LWP in a thread of hierarchy implies the potential existence of subsystem(s) and/or component(s). i.e. the *existence* of a subsystem and/or component is *contingent* upon the eventual presence of a LWP in a thread of hierarchy. Otherwise, their presence are not allowed. Further, roles of subsystems and components for a given thread of a hierarchy are interchangeable based on semantic of the software application under study. The items 4a, 4b, and 4c are the formalization of the *FSMWeb's contingent node-existence relationships* and are summarized in Table I. The following relations exist among the subsystems, components, and LWPs in FSMWeb:
   a) The presence of subsystems (*ss*)/components (c) implies that subsystems/components lead to one or more LWPs (*page*). Otherwise, subsystems/components cannot exist.
   $$(\forall ss \in (f) = \emptyset) \vee ((\forall c_i \in comp(ss) \vee ss) = \emptyset) \mid (page_i(c_i) = \emptyset)$$
   $$\vee$$
   $$(\forall ss \in (f) \neq \emptyset) \cap ((\forall c_i \in comp(ss) \cap ss) \neq \emptyset) \mid (page_i(c_i) \neq \emptyset)$$
   where ss, c, and page are instances of subsystems, components, and web pages (LWPs), respectively.
   b) Components cannot exist without their corresponding subsystems (components do not have intrinsic value of their own. Their existence is to facilitate creation of layer(s) of indirection when a subsystem needs to be further decomposed due to its level of complexity).
   $$(\forall c_i \in comp(ss) \neq \emptyset) \vee (\forall c_i \in comp(ss) = \emptyset) \mid (\forall ss \in f) \neq \emptyset)$$
   c) Presence of a subsystem implies existence of one or more LWPs irrespective of component's existence.
   $$((\forall ss \in f) \neq \emptyset) \vee ((\forall ss \in f) = \emptyset) \mid (\forall page_i \in c_i) \neq \emptyset)$$

   In Table I, the designation of zero and one under the columns, ***ss***, ***c***, and ***page***, represent absence or presence of subsystems, components, and pages considering their inter-dependent relationships.

TABLE I: FSMWeb's Contingent Node-Existence Relationship

Row #	*ss*	*c*	*LWP*	Result	Explanation
1	0	0	0	Allowed	An empty hierarchical path (null case)
2	0	0	1	Allowed	A single function
3	0	1	0	Not Allowed	A clustered node *must* terminate to a LWP node
4	0	1	1	Allowed	Roles of components and subsystems are *interchangeable*
5	1	0	0	Not Allowed	A Subsystem can not exist without its constituent LWPs
6	1	0	1	Allowed	Roles of components and subsystems are *interchangeable*
7	1	1	0	Not Allowed	clustered nodes *must* terminate to a LWP node
8	1	1	1	Allowed	All levels exist

5) **Heterogeneity in Hierarchical Levels** : The entities (subsystems, components, LWPs) may coexist alongside of the other entities at the same hierarchical level. Entities at the same hierarchical level may navigate to the other entities at the same or different hierarchical levels.
6) **Coupling**: Given the two states S_i, and S_j (i $\neq$ j) , we say S_i is the *predecessor* node, if S_i occurs

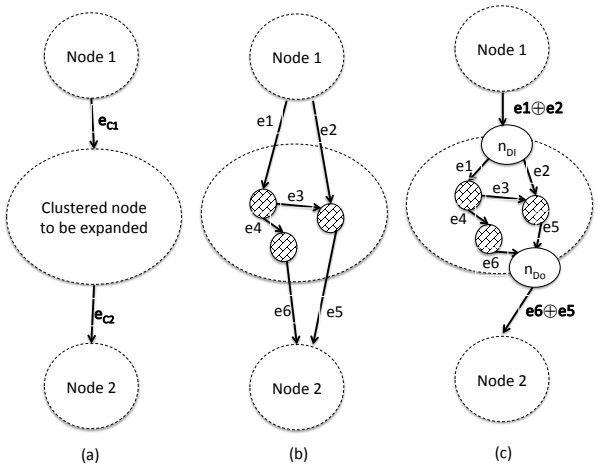

Fig. 2: Transformation of Multi-edge Connection Graph to SESE Graph

before S_j with respect to W_s for a given full test path. In this case, S_j is the *successor* node in relation to S_i with respect to W_s in a full test path. We say S_j is *coupled* to S_i if changes in S_i affect state of S_j (Ammann et al. [6] uses the term *dominant* to refer to predecessor node, when coupling is involved). FSMWeb must be constructed in such a way (considering the application's semantics) in order for the dependent nodes to follow the node on which they depend. Modifications to predecessor node(s) may affect the succeeding node(s).

7) **Classification of Edges**: Similar to [7], the edges are classified into 1) *external* and 2) *internal*. An edge connecting a pair of LWP nodes is considered an *internal* edge. Otherwise, the edge is said to be *external*.
8) **Single Entry/Single Exit (SESE)**: All transitions (edges) among nodes are represented through a single edge (entering or exiting *to* and *from* nodes). The Single Entry/Single Exit (SESE) [6] edges serve to keep FSMWeb model in a simpler graphical representation as opposed to multiple-edge connection among nodes (see Fig. 2(b)). The n_{DI} and n_{DO} represent *dummy in* and *dummy out* nodes, respectively. n_{DI}, and n_{DO} could be viewed as mechanisms through which a single incoming edge *expands* into one or many edges upon entering into a cluster node and subsequently *collapses* multiple edges into a single edge upon exiting the cluster node. The annotation of external edges with *exclusive OR* symbol ($\oplus$), indicates that the single edge (*in* or *out*) incident on a cluster node maps to one and only one edge within the cluster node. The use of *exclusive OR* as an edge-subscript is strictly for simpler graphical representation purposes to enforce SESE property - the actual selection of a single edge from many choices of edges occurs within the action (*act*) parameter of the annotated edge.

C. FSMWeb Graphical Notations

The followings is a set of graphical notations as aids when depicting FSMWeb during model construction and maintenance.

1) Nodes are depicted by ovals.
   a) Clustered/sub-cluster nodes are depicted by dashed contour lines (implying decomposability).
   b) LWP nodes are depicted by solid contour lines (implying non-decomposability).
   c) Navigation among nodes (cluster and/or LWPs) are represented through single edges.
2) Edges are depicted with solid (implying non-decomposability) directed arrows where the arrow direction implies direction of the transition. The direction of an arrow indicates whether the edge is entering into (in) a node or exiting out of a node.
3) A subscripted-Exclusive OR (XOR) notation for an *in/out* edge for a cluster node is used if the in/out edge through a dummy n_{DI}/n_{DO} fans out to the different nodes within that cluster node (see Fig. 2 (c)). For example, an incoming XOR-subscripted edge, $e_{a\oplus b\oplus c}$, fans out to distinct edges of e_a, e_b, and e_c within the cluster node. A bracketed variation of XOR-subscripted edge notation is used when the fan out includes more than three edges within the cluster node. For example, an incoming XOR-subscripted edge, $e_{[a1\oplus a5]}$ fans out to distinct edges of e_{a1}, e_{a2}, e_{a3}, e_{a4}, and e_{a5} within the cluster node.
4) The nodes, and the edges are depicted in **bold** when they are affected in the presence of changes to FSMWeb. For example, a newly added cluster node will be shown in a contour line and bold, etc.

IV. Test Generation in FSMWeb

Generally, the process of test generation starts with selecting one or many test coverage criterion/criteria (*crit*) that fulfill a set of test requirements (*TR*). The individual test requirements ($tr \in TR$) may or may not utilize the same test coverage criterion [6]. Nonetheless, the test coverage criterion *must* fulfill the given test requirement (the reverse holds true as well: a test requirement is *realized* through proper selection of a test criterion) in order to be *feasible*; Otherwise, it is *infeasible*. i.e. The selection of *improper* coverage criterion for a test requirement leads to infeasibility. In this paper, we assume that coverage criteria are properly selected for their respective test requirements. In [6] there is a listing of coverage criteria and subsumption relationships. In this paper, we avoid test coverage criteria that involve use of *value* (e.g. definition-use pairs, etc.) since the focus of this work are abstract test cases.

A. Formalizing FSMWeb Test Generation

A test case is a set of inputs corresponding (through test case execution) to a set of *expected* (test oracle)

outputs. In FSMWeb, the shortest test path is identified by a current state (s_i), an *out* edge from s_i (e_i), and the next state (s_{i+1}). Longer test paths are realized through cascading: assuming that s_{i+1} is the new current state, e_{i+1} is the new *out* edge, and s_{i+2} is the next new state, etc. A test path bounded by W_s and W_e, is a *full* test path and contains only LWPs connected by internal edges; otherwise, it is a *partial* test path. In this paper, the goal is to produce a test suite comprised of all full test paths. In FSMWeb there are two somewhat similar methods in which test cases could be generated. However, both of the methods result to the generation of identical suite of test cases.

The first method, starts at the most refined level of FSMWeb (i.e. where there are only LWPs and connecting (internal) edges) and generates partial test paths bounded by n_{DI} and n_{DO} nodes. The number of partial test paths depend on both the direction of edges and the coverage criterion (we are considering edge coverage in this paper). The next step is to consider the more abstract node (moving toward more abstract levels) and *prepend* node(s) and edge(s) on the n_{DI} side to the partial paths and *append* node(s) and edge(s) on the n_{DO} side to the partial paths, etc. This process terminates when the prepending/appending actions encounter W_s, W_e nodes. The first method could be viewed as full path test generation *from inside out*.

The second method considers the *outside in* approach where the test paths are initially bounded by W_s and W_e but contain *at least* one cluster node. By successively expanding the cluster nodes and prepending/appending the partial test paths, the full test paths are generated. In this section, we formalize the test generation (TG) outlined in [1] by incorporating the selection of coverage criteria (*crit*) and partial path aggregation (*agg*) parameters into the model's data structure (equation 1). Partial paths are the test paths that are not bounded by W_s and W_e nodes. The aggregation is the consolidation of the partial paths (through appending and/or prepending) the nodes in such a way to elongate the test paths. A full test path is a test path that is bounded by W_s and W_e nodes, contains only LWPS, and connected by edges.

Now, we formalize TG as a *test generation function* where TG: HFSM $\times$ $\{crit_l, agg_l\}_{l=0}^{n} \rightarrow$ TC
, where *l* represents the various levels of abstractions, and TC is a set of test cases. Substituting equation 1 into the aforementioned test generation function (TG), we obtain

$$TG : \{M_l, crit_l, agg_l\}_{l=0}^{n} \rightarrow TC \qquad (2)$$

Note: equation 2 has ignored the test value selection step outlined in [1] since the focus of this work involves abstract test cases. It is possible to apply different coverage criteria in the different levels of hierarchy for the same hierarchical thread of FSMWeb. However, use of the coverage criteria belonging to the same subsumption coverage hierarchy [6] in the same thread of the model's hierarchy may produce the same set of partial test paths (and perhaps the same set of test cases). We call this side effect *coverage masking* where the choice of a particular coverage criterion may be masked by another coverage criterion under certain conditions due to hierarchical nature of FSMWeb. For example, when edge coverage criterion and edge-pair coverage criterion (both belonging to the same level of subsumption coverage hierarchy [6]) are applied in the same thread of hierarchy, they may produce the same test case (the same test path) depending on the specific order of abstraction levels in which they are applied. Test case generation could be done manually or could be automated using the external edges and n_{DI} and n_{DO} nodes.

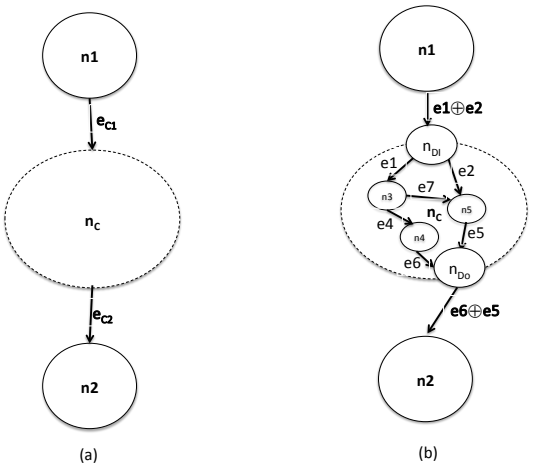

Fig. 3: Test Cases Generation

B. Test Case Generation Example

Using Fig. 3, we illustrate the process of test case generation in FSMWeb using the first method. The aforementioned figures depict the n_c cluster node *before*, and *after* expansion. Since n_c node after expansion only contains LWP nodes and edges, we create the partial paths for n_c cluster node. We choose the edge coverage as a coverage criterion of choice for this example.

1) partial-path 1: n_{DI}, e_1, n_3, e_4, n_4, e_6, n_{DO}
2) partial-path 2: n_{DI}, e_2, n_5, e_5, n_{DO}
3) partial-path 3: n_{DI}, e_1, n_3, e_7, n_5, e_5, n_{DO}

The next step is to *aggregate* n_1, and n_2 (by replacing n_{DI} and n_{DO} with n_1 and n_2, respectively) with partial-path 1 through partial-path 3, to obtain new partial-path 1A through partial-path 1C.

1) partial-path 1A: n_1, e_1, n_3, e_4, n_4, e_6, n_2
2) partial-path 1B: n_1, e_2, n_5, e_5, n_2
3) partial-path 1C: n_1, e_1, n_3, e_7, n_5, e_5, n_2

The last step is to *aggregate* (by prepending W_s followed by e_s, and appending e_e followed by appending W_e to) partial-path 1A through partial-path 1C, to obtain test paths TC1 through TC3 as follows:

1) TC1: W_s, e_s, n_1, e_1, n_3, e_4, n_4, e_6, n_2, e_e, W_e
2) TC2: W_s, e_s, n_1, e_2, n_5, e_5, n_2, e_e, W_e
3) TC3: W_s, e_s, n_1, e_1, n_3, e_7, n_5, e_5, n_2, e_e, W_e

, where e_s, and e_e are outgoing, and incoming edges from/to W_s and W_e, respectively. W_s, W_e, e_s, and e_e are not shown in Fig. 2a and Fig. 2b for the purpose of brevity.

V. Regression Testing using Classification

The test case classification approach introduced by Andrews et al. [5] uses a general regression testing framework [8]. We formalize types of changes to the FSMWeb models based on changes to the web application (its inputs, Logical Web Pages - LWPs, and navigation between them). We adopt the approach in [9] to classify tests into *reusable*, *retestable*, and *obsolete* tests by specifying rules for classification based on the types of changes to the model. In this context, the regression testing problem for the FSMWeb model becomes the following:
Given a FSMWeb model, *HFSM*, a test set *T* (used to test *HFSM*) and a modified version of *HFSM*, *HFSM′*, find a way of making use of *T*, to gain sufficient confidence in the correctness of the application modeled by *HFSM′*.
Similar to [8], we propose the following steps as an outline of our approach to solve this problem:

1) Identify and classify changes made to *HFSM* that result in *HFSM′*. Specify the effect of each type of change on tests $t \in T$. Using these rules, classify tests in T into mutually exclusive sets of test cases: obsolete (O_T), retestable (R_T), and reusable (U_T).
2) Use the results of step 1 to select a set of retestable test cases $T' \subseteq R_T$ that may reveal change-related faults in *HFSM′*.
3) Use T' to test *HFSM′*.
4) Determine if any parts of the system have not been tested adequately and generate a new set of tests T''.
5) Test the web application with T''.

Generally speaking, the two types of changes are node changes and edge changes. After these changes have been identified, they need to be classified as to their impact on existing test cases. Further, these changes could be path affecting changes (PAC) or non-path affecting (NPC) changes. This determines whether tests are obsolete, retestable, or reusable.

The test paths that visit deleted or modified nodes and/or tour the deleted or modified edges become obsolete. The test paths that have node(s) or edge(s) added and are PAC become obsolete (o). The test paths that have node(s) or edge(s) added and are NPC become retestable (r). The test cases that are not obsolete nor retestable are reusable (u) [5].The interested reader should consult section 4 in [5] for test case classification details. Table II is a summary of the aforementioned discussion.

TABLE II: Test Case Classification into Obsolete (o), Retestable (r) based on Edge Coverage

		PAC	NPC
NODE	ADD	o	r
	DEL	o	o
	MOD	o	o
EDGE	ADD	o	r
	DEL	o	o
	MOD	o	o

VI. Test Case Patching

In most regression testing approaches [3], [32], [33], the test cases affected by changes (whether major or minor) are regenerated. However, the test cases affected by changes could be repaired [24] rather than performing a full test case regeneration. This paper refers to repairing as test case patching. Patching requires test case classification [5] as a vital step in improving testing costs. Patching becomes favorable when only *minor* changes are needed for a large amount of obsolete test cases. The term, *minor* is context sensitive with respect to the nature of a change. For example, a change involving variable name modification affecting many test cases is considered minor (since fixing entails replacing all the previous variable names with the new variable names) whereas a variable used for computations or for logical considerations (e.g. branching to different pieces of code, etc.) is not considered to be a minor change because the rest of the test path may be affected as a result of the change. Patching may lead to cost saving depending on nature of the change and the number of affected test cases. The following section enumerates patching rules as a set of guidelines when patching may or may not lead to cost savings.

A. Rules of Patching

The followings are rules of thumb when patching is favorable versus full test case regeneration:

1) When the change is a non-path affecting change (NPC).
2) When there are a *small* number of changes. *small* could be defined as a ratio of the number of changes over the number of test cases and compared against a desired threshold.
3) When the changes are *local* (versus design changes that may affect many test cases, many elements per test case, and/or cross cluster node boundaries).
4) When the changes do not affect the test path length (i.e some changes may elongate or shorten test path lengths).

The followings are rules of thumb when patching is *NOT* favorable versus full test case regeneration:

1) When the change is a path affecting change (PAC).
   a) PAC occurs towards the beginning of the test path (making most of the test path unusable).
   b) The change affects a shared element across many test cases. The change propagates (RIP: *Reachability*, *Infections*, and *Propagation* [6]) to other test cases as well.

c) When the changes involve design or high-level requirement changes (e.g. use case).

B. Patching Procedure

The following is the procedural (algorithmic) approach to patching for a given set of obsolete test cases. Patching does not apply to obsolete test cases that are no longer valid due to high level requirement changes.:

1) When the change (adding, deleting, modifying) involves an edge or a LWP node:
   a) Identify the affected test case corresponding to the change, and locate the change (e.g. the affected edge).
   b) Determine whether the change is PAC or NPC.
   c) If PAC, generate a new partial path starting from the affected element (edge or LWP node) to the end of the test path replacing the affected partial path. This applies to longer test path lengths. For short path lengths test case regeneration is more appropriate.
   d) If NPC, replace the affected element (edge or LWP node) in the test path with a newly generated element.
2) The cases involving multiple edges or LWP nodes of type NPC follow the handling of multiple cases of single edge/LWP nodes with NPC-type changes.
3) In the cases involving multiple changes to a single test case of mixed NPC and PAC types, the first PAC change closest to the beginning of the test path *dominates* the subsequent changes (whether PAC or NPC).
4) When the change involves conversion of a LWP node to a cluster node:
   a) Replace the LWP node with a newly created cluster node with corresponding Single Entry (n_{DI}), and Single Exit (n_{DO}) nodes.
   b) Create partial paths and aggregate them with the test paths that tour the newly created partial paths (See section IV-B).
5) When the change involves the conversion of a cluster node to an LWP node:
   a) For all the paths *pt* that are members of the abstract test case ($pt \in AT$), replace the LWP to be node-changed (LWP_{nc}) with the new cluster node (CL_i). Assuming that the external edges of CL_i are represented by $e_{x_1},...,e_{x_m}$ then
   b) For all the paths that belong to t_a that tours e_{x_i} (i=1,...,m), replace partial path $e_{x_i},...,e_{x_j}$ with LWP_{nc} *before* path aggregation.
   or let *pt* be the paths that visit CL_i and replace the predicate on the outgoing edge with pt_{nc} and annotate the node with *nc* (see Fig. 4). The follwing is a formalization of the two aforementioned steps: Let $e_{x_1},...,e_{x_m}$ be the external edges of CL_i.

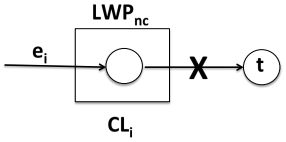

Fig. 4: Changing a Cluster Node to a LWP Node

Then, $\forall pt \in AT$: *pt* tours e_{x_i} (i=1,...m)
replace partial path $e_{x_i},...,e_{x_j}$ with LWP_{nc}

VII. Cost and Trade Offs

In the context of this paper, cost is the mechanism for comparing several different regression testing approaches in order to select the most favorable approach (reduced cost). Additionally, cost refers to required resources. These resources may include analysis, set up, generation/regeneration, execution, etc. Our work is more focused in several respects: 1) it concentrates on the costs associated with obsolete test cases (the identification, execution, and regeneration), and the use of a patching approach for the further cost reduction of obsolete test cases, 2) it considers only black-box testing/regression testing approaches, and 3) it only considers typical web applications (web applications such as Ajax with specially-crafted user interfaces are not considered). A comprehensive cost model combining all factors are beyond the scope of this paper and is left as a future work.

Similar to [9], [18], [19], we start by constructing cost models for selective retesting using classification, patching, retest-all, and brute force regression testing approaches followed by selecting the parameters that influence the cost for a given approach subject to certain assumptions. Next, we analyze (compare) the costs incurred in patching versus retest-all, and brute force approaches. Arithmetically, the comparison between two different regression testing approaches amounts to subtracting the cost of one approach from the other approach, simplifying - when identical parameters exist, and checking whether inequality holds. Trade offs are considered in the post cost-analysis phase of the process in order to determine the most favorable cost.

A. FSMWeb Regression Testing Cost Model

To construct a cost model for regression testing approaches, we start by categorizing the costs into three groups for *Selective retesting* using *classification*, *Retest-all*, and *Brute-Force*. The high level cost-related parameters associated with *Selective Regression Testing* are:

1) *Classification* of the existing test cases ($class_T$),
2) *Execution* of the existing test cases ($exec_T$),
3) *Generation* of new test cases ($gen_{T''}$),
4) *Execution* of new test cases ($exec_{T''}$).

The high level cost-related parameters associated with *Retest-all Regression Testing* are:

1) *Generation* of all test cases (*gen*),
2) *Execution* of all the test cases (*exec*).

The high level cost-related parameters associated with *Brute-Force Regression Testing* are:

1) *Execution* of existing test cases ($exec_T$)
2) *Validation* of existing test cases (val_T),
3) Generation (potentially) of brand new test cases ($gen_{T'''}$).

At high level, table III summarizes the costs for each category of regression testing. C_{SR-C}, C_{RA}, and C_{BF} represent costs for selective retesting using classification, retest-all, and brute force, respectively.

TABLE III: Different Cost Categories and Associated Parameters in Regression Testing

Types of Regression Testing	**Costs**
Selective retesting using classification	$C_{SR-C} = class_T + exec_T + gen_{T''} + exec_{T''}$
Retest-all	C_{RA} = *gen* + *exec*
Brute-Force	$C_{BF} = exec_T + val_T + gen_{T'''} + exec_{T'''}$

B. Cost Computation using a Classification and Patching Approaches

We compute the cost of selective retesting using classification approach (C_{SR_C}) with respect to obsolete test cases. First, the test cases (T) are classified into 3 distinct (non-overlapping) set of test cases of, obsolete (O_{SR_C}), retestable (R_{SR_C}), and reusable (U_{SR_C}). Second, by executing the retestable test cases, a portion of test cases pass and become reusable (U_{R-SR_C} - implying they are reusable and must be added to the pool of the reusable test cases) and the other portion of test cases fails and become obsolete (O_{R-SR_C} - implying they are obsolete and must be added to the pool of the obsolete test cases). Third, by analyzing the newly formed pool of the obsolete test cases ($O_{R-SR\_C} + O_{SR\_C}$), we determine whether the obsolescence is due to elimination of application functionality(ies). If the functionality(ies) still exist, then, replacement test cases are generated in order to maintain the existing functionality(ies) (O_{rep-SR_C}). Otherwise, the old functionality(ies) no longer exist and there is no need for generation of test cases (skip test case generation - $O_{skip-SR\_C}$). Last, the generated replacement test cases are added to pool of the reusable test cases ($U_{SR\_C} + U_{R-SR\_C}$) and further combined with new test cases (O_{new-SR_C}) for new functionality(ies).

The only difference between patching the test cases and the classification approaches is in the analysis of the combined obsolete test cases ($O_{R-SR\_C} + O_{SR\_C}$). In the patching approach, instead of generating test cases for replacement purposes, the existing (obsolete) test cases are analyzed and portion of them are patched (repaired). Figure 5 is the depiction of the foregoing elaboration and it shows the patching work using dash-line format.

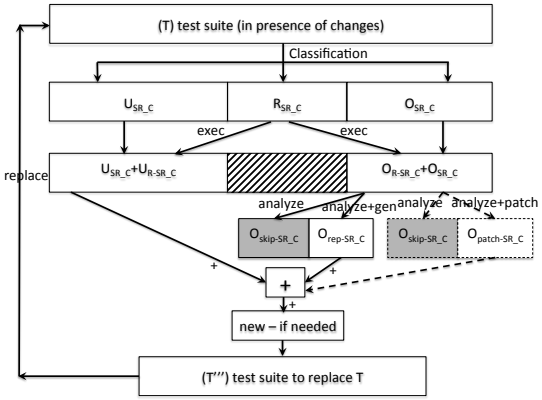

Fig. 5: Regression Testing using Classification and/or Patching

$$C_{SR\_C} = C_{class} + (C_{exec} * R_{SR\_C}) + C_{an} * (O_{R-SR\_C} + O_{SR\_C}) + (C_{gen} * O_{rep-SR\_C}) + C_{new\_SR\_C} \quad (3)$$

$$C_{patch} = C_{class} + (C_{exec} * R_{SR\_C}) + C_{an} * (O_{R-SR\_C} + O_{SR\_C}) + (C_{patch} * O_{rep-SR\_C}) + C_{new\_SR\_C} \quad (4)$$

$C_{class}, C_{exec}, C_{an}, C_{gen}$, and C_{patch} represent costs of test case classification, execution, analysis, generation, and patching, respectively. The comparison of the cost of selective regression testing using classification (equation 3) and cost of regression testing using patching (equation 4) reveals that the theoretical difference between the two approaches is in the 4th terms. This point emphasizes that the patching approach requires some of the steps in the classification approach.

C. Cost Computation using a Retest-all Approach

The computation of the cost for the retest-all approach follows the same methodology as outlined in the section VII-B. First, execute the test cases (T) resulting to some test cases passing and becoming reusable (U_{RA}), and some other test cases failing and becoming obsolete (O_{RA}). second, analyze the obsolete test cases (O_{RA}) to determine whether the obsolescence is due to elimination of application functionality(ies). If the functionality(ies) still exist, then, replacement test cases need to be generated maintaining the existing functionality(ies) (O_{rep-RA}). Otherwise, the old functionality(ies) no longer exist and there is no need for generation of test cases (skip test case generation - $O_{skip-RA}$). Last, the newly generated (replacement) test cases need to be combined with the reusable test cases (U_{RA}).

$$C_{RA} = (C_{exec} * T) + (C_{an} * O_{RA}) + (C_{gen} * O_{rep-RA}) \quad (5)$$

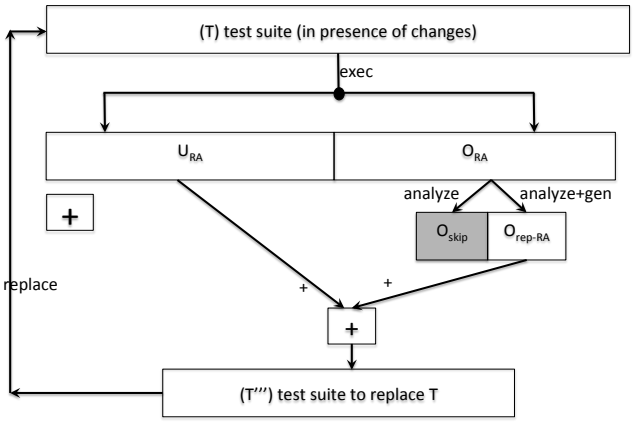

Fig. 6: Regression Testing using Retest-all

One of the differences between equations 5, and equation 3 and/or equation 4 is that equation 5 does not have a cost representation for new test cases. This is due to the way retest-all approach operates in which the cost of generation of the new test cases is *absorbed* in the cost of analysis and generation for the replacement of the test cases.

D. Cost Computation using a Brute Force Approach

In the brute force regression testing approach, each and every test case needs to be executed (*exec*) and validated (*val*). If the validation fails, then a new test case must be generated (*gen*), re-executed, and re-validated. Depending on test case length and number of changes present in a particular test case a test case may go through one or many *generate-execute-validate* cycles until the test case pass. We designate, *K*, as a required number of cycles for a given test case before it pass. For practical purposes, *K* could be a configurable parameter with a maximum value over which the brute force may be deemed as an unsuitable regression testing approach for a given web application. Figure 7 depicts the brute force regression testing approach

$$C_{BF} = \sum_{i=1}^{n}[(C_{exec}+C_{val})+\sum_{j=0}^{K}(C_{gen}+C_{exec}+C_{val})_j]_i \tag{6}$$

, where n represent number of test cases, and i and j are looping indices through the summation operations.

E. Cost Comparisons

For industrial-strength web applications with large numbers of test cases, and in the presence of major changes (i.e. changes in use case, design, etc.), the brute force or retest-all testing paradigms may seem more plausible than the selective retesting approach. However, our assumption is that the changes to the web applications being tested are minor (i.e. local changes to a particular test case does not affect the other test cases). The underlying assumption in the following cost comparisons is

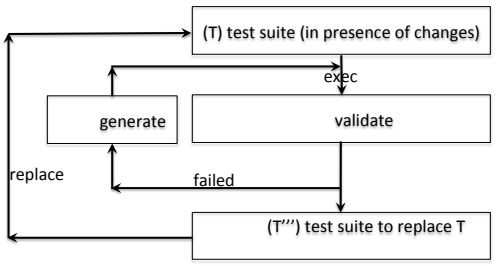

Fig. 7: Regression Testing using Brute Force

that the cost of test case generation dominates the other costs.

In the following sections, we use the cost models developed in the preceding sections to perform a pairwise cost comparisons. The pairwise comparison of four cost models involves six different comparisons. However, for brevity purposes we show only the three comparisons for which costs of the regression testing approaches are progressively decreasing.

1) BruteForce and Retest-all Cost Comparison: In order to compare the cost of a brute force approach against a retest-all approach, we check the validity of the inequality 7:

$$C_{BF} - C_{RA} > 0 \tag{7}$$

Substituting the equations 6 and 5 in inequality 7, we obtain the inequality 8

$$\begin{aligned}&\sum_{i=1}^{n}[(C_{exec}+C_{val})+\sum_{j=0}^{K}(C_{gen}+C_{exec}+C_{val})_j]_i-\\&[(C_{exec}*T)+(C_{an}*O_{RA})+\\&(C_{gen}*O_{rep-RA})]>0\end{aligned} \tag{8}$$

The brute force approach costs more than retest-all approach since brute force approach operations involves all the test cases versus retest-all operations that involves a subset of test cases ($n > O_{rep-RA}$) and on average there are more iterations through *generate-execute-validate* sequence ($K > 1$). Therefore, inequality 8 holds.

2) Retest-all and Classification Cost Comparison: We follow the same approach for costs comparison as section VII-E1:

$$C_{RA} - C_{SR\_C} > 0 \tag{9}$$

We substitute the equations 3 and 5 in inequality 9 and simplify to obtain the inequality 10

$$\begin{aligned}&[(C_{exec}*T)+(C_{an}*O_{RA})+\\&(C_{gen}*O_{rep-RA})]-\\&[C_{class}+(C_{exec}*R_{SR\_C})+\\&C_{an}*(O_{R-SR\_C}+O_{SR\_C})+\\&(C_{gen}*O_{rep-SR\_C})+C_{new\_SR\_C}]>0\end{aligned} \tag{10}$$

As mentioned in "*Cost Comparison*" sub-section (VII-E), the cost of replacing (by generating test cases) the obsolete test cases (maintaining the existing functionalities) is

greater than the combined cost of replacing the test cases in selective retesting and generation of new test cases ($[C_{gen} * O_{rep-RA}] > [(C_{gen} * O_{rep-SR\_C}) + C_{new\_SR\_C}]$) for minor changes ($[C_{gen} * O_{rep-RA}]$ dominates the other terms). This implies that the inequality 9 (and subsequently inequality 10) holds.

3) Patching and Classification Cost Comparison: In the sections VII-E1, and VII-E2, we assumed that the dominating cost factor is the cost of test generation for replacing (maintaining the existing functionalities) obsolete test cases. This is because test generation in FSMWeb involves models (M_l), coverage criteria ($crit_l$), and the aggregation of test subpaths (agg_l) in different hierarchical levels (see equation 2). A change to a test case may be path affecting (PAC) implying that the patching and regeneration cost is nearly the same ($C_{patch} \approxeq C_{gen}$). However, for non-path affecting changes (NPC) the cost of patching would be less in comparison to full test case regeneration. The assumption is that the number of node or edge changes in non-path affecting change (NPC) is lower than the number of nodes and edges for a given test path (corresponding to a test case). This leads to the conclusion that based on the assumptions (in section VII-E), $C_{gen} > C_{patch}$. Based on this analysis, the conclusion is that the brute force, and the retest-all regression testing approaches yield higher costs among the four approaches under comparisons when there are minor changes. This result is also intuitive since brute force, and retest-all involves processing of all the test cases and both approaches incur the overhead cost that could be avoided with analysis (e.g. classification). Further, the cost of patching approach is less than the cost of selective retesting using classification approach given that changes are NPC for test paths of considerable number of nodes and edges.

VIII. An Experiment Results

This section discusses the results of a run of a scaled-down Medical Emergency Response (MER) web application with minor versioning. MER is comprised of three use cases (subsystem clusters) of: 1) On-site Care, 2) Emergency Department, and 3) Application Administration. An authorized user must login (through a LWP) before having access to MER's features. Using the test generation in section IV, we decomposed MER into nodes and internal edges and produced 24 test cases (test paths) of which 2 test cases were redundant. We made 5 minor changes to existing requirements (versioning MER) and added a new LWP (previously non-existing). Employing *classification* using selective regression testing approach, 8 test cases became obsolete, and 14 test cases were identified as reusable. There were no retestable test cases since we assumed the changes were all of NPC types. Further, 2 new test cases were produced due to the new change. Using *patching*, we repaired (patched) 3 test cases in contrast to regeneration of 8 test cases for replacing the obsolete test cases. In this experiment, patching proved to be the favorable (least costly) regression testing approach.

IX. Conclusion and Future Work

The goal of this paper was to perform cost analysis and trade offs in comparing cost of four different regression testing approaches. However, some missing works had to be done before proceeding with the cost analysis work (see section I). However, the prerequisite in achieving this goal entailed the formalization the FSMWeb test model followed by the test generation process. Next, was to create cost models for different categories of regression testing approaches. It was determined that the most favorable approach was patching by considering it's corresponding cost subject to certain assumptions and trade offs. Overall, performing an accurate comparisons among the different regression testing approaches and applying an appropriate trade off analysis is a difficult task. The existence of many types of costs and the inter-relation between cost and value is further compounded by many additional validity constraints swaying the future research toward multi-objective regression test optimization [34]. In the context of this paper, it is due to the fact that web application changes are not controllable, at least from the perspective of the testing system or methodology. Dynamic dependencies among the web application changes may have a compounding impact on test cases when performing regression testing. With this said, the patching approach shows promise when there are few and/or *minor* changes with respect to number of test cases in the test suite.

The future work will extend the FSMWeb test model to include failure and error recovery features in order to represent a real-life web application.

X. Acknowledgement

The author wishes to thank Dr. Anneliese A. Andrews for insights (Figure 4), and suggestions. Similarly, the author is grateful to Dr. Kenneth M. Hopkinson for encouraging comments, insights, and suggestions in embellishing this article.

The author is also grateful to the anonymous referees for their valuable comments and suggestions to improve the presentation of this paper.

References

[1] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing Web Applications by Modeling with FSMs," *Journal of Software and Systems Modeling*, vol. 4, no. 3, pp. 326–345, 2005.

[2] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines–A Survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090 –1123, August 1996.

[3] B. Korel, L. H. Tahat, and B. Vaysburg, "Model Based Regression Test Reduction Using Dependence Analysis," *International Conference on Software Maintenance, 2002*, pp. 214–223, 2002.

[4] A. Andrews, J. Offutt, C. Dyerson, C. J. Mallery, K. Jerath, and R. Alexander, "Scalability Issues with Using FSMWeb to Test Web Applications," *Information and Software Technology*, vol. 52, no. 1, pp. 52–66, January 2009.

[5] A. A. Andrews, S. Azghandi, and O. Pilskalns, "Regression Testing of Web Applications Using FSMWEB," *ACTA Press*, Nov 2010.

[6] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[7] J. Fernandez and J. Gonzalez, *Multi-Hierarchical Representation of Large-Scale Space: Applications to Mobile Robots*. Kluwer Academic Pubishers, 2001.
[8] M. H. G. Rothermel, "Analysing Regression Testing Techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–55, August 1996.
[9] H. K. N. Leung and L. White, "A Cost Model to Compare Regression Test Strategies," *Conference on Software Engineering*, pp. 201–208, October 1991.
[10] M. J. Harrold, "Architecture-based regression testing of evolving systems," , 1998.
[11] G. Rothermel and M. J. Harrold, "A Framework for Evaluating Regression Test Selection Techniques," *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pp. 201–210, 1994.
[12] A. Malishevsky, G. Rothermel, and S. Elbaum, "Modeling the Cost-benefits Tradeoffs for Regression Testing Techniques," in *Software Maintenance, 2002. Proceedings. International Conference on*, 2002, pp. 204–213.
[13] D. Binkley, "Reducing the Cost of Regression Testing by Semantics Guided Test Case Selection," in *Software Maintenance, 1995. Proceedings., International Conference on*, 1995, pp. 251–260.
[14] ——, "Semantics Guided Regression Test Cost Reduction," *Software Engineering, IEEE Transactions on*, vol. 23, no. 8, pp. 498–516, 1997.
[15] ——, "The Application of Program Slicing to Regression Testing," *Information and Software Technology*, vol. 40, no. 11-12, pp. 583–594, December 1998.
[16] J. Sant, A. Souter, and L. Greenwald, "An Exploration of Statistical Models for Automated Test Case Generation," *SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–7, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083256
[17] A. C. D. Neto, R. Subramamyan, M. Vieira, and G. H. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in . New York, NY, USA: ACM, 2007, pp. 31–36.
[18] H. DO and G. Rothermel, "An Empirical Study of Regression Testing Techniques Incorporating Context and Lifetime Factors and Improved Cost-Benefit Models," *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 141–151, November 2006.
[19] H. Do and G. Rothermel, "Using Sensitivity Analysis to Create Simplified Economic Models for Regression Testing," *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pp. 51–62, July 2008.
[20] H. Park, H. Ryu, and J. Baik, "Historical Value-Based Approach for Cost-Cognizant Test Case Prioritization to Improve the Effectiveness of Regression Testing," in *Secure System Integration and Reliability Improvement, 2008. SSIRI '08. Second International Conference on*, 2008, pp. 39–46.
[21] B. Beizer, *Software testing techniques*. Dreamtech Press, 2002.
[22] A. Marchetto, P. Tonella, and F. Ricca, "State-Based Testing of Ajax Web Applications," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, 2008, pp. 121–130.
[23] N. Alshahwan and M. Harman, "Automated Session Data Repair for Web Application Regression Testing," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, 2008, pp. 298–307.
[24] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 4:1–4:36, Nov. 2008.
[25] S. Si Huang, M. Cohen, and A. Memon, "Repairing GUI Test Suites Using a Genetic Algorithm," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, 2010, pp. 245–254.
[26] L. C. Briand, Y. Labiche, and S. He, "Automating regression test selection based on UML designs," *Information and Software Technology*, vol. 51, no. 1, pp. 16–30, 2009.
[27] A. Mesbah, A. van Deursen, and D. Roest, "Regression Testing Ajax Applications: Coping with Dynamism," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, 2010, pp. 127–136.
[28] S. Sprenkle, L. Pollock, and L. Simko, "A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, 2011, pp. 230–239.
[29] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 571–580.
[30] A. Marback, H. Do, and N. Ehresmann, "An Effective Regression Testing Approach for PHP Web Applications," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 221–230.
[31] S. Sampath, R. Bryce, and A. Memon, "A Uniform Representation of Hybrid Criteria for Regression Testing," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2013.
[32] Y. Chen, R. L. Probert, and D. P. Sims, "Specification-based Regression Test Selection with Risk Analysis." IBM Press, 2002, pp. 1–14.
[33] A. Tarhini, Z. Ismail, and N. Mansour, "Regression Testing Web Applications," in *Advanced Computer Theory and Engineering, 2008. ICACTE '08. International Conference on*, December 2008, pp. 902 –906.
[34] M. Harman, "Making the Case for MORTO: Multi Objective Regression Test Optimization," *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference*, pp. 111–114, March 2011.

Seif Azghandi received the B.S. degree in computer engineering from University of Illinois at Chicago, Chicago, IL, in 1989. He received his M.S. degree in computer science from the University of Colorado at Denver, Denver, CO in 2006. He has 20 years of industry experience in software industry and served as a system, and a technical architect on many large-scale software projects. He is currently a post-doctoral researcher at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, OH. His interests are in testing/regression testing, smart grids, cognitive radios, and data mining.