Towards Collaborative Testing of Workflows in WMVC-Based Web Applications

Marcel Karam
American University of Beirut
Beirut, Lebanon
mk62@aub.edu.lb

Haidar Safa
American University of Beirut
Beirut, Lebanon
hs33@aub.edu.lb

Abstract— At the core of visual testing is the idea that displaying the test failure on the visual artifacts that represent the constituents of a software under test, rather than just describing it textually has the potential of increasing clarity and understanding of the problem amongst a software testing team. This is particularly true and even necessary in a collaborative testing environment which requires greater communication between testers and developers. In this work we introduce a collaborative visual state-based testing methodology for applications developed in Ruby on Rails, a representative of the Workflow and MVC-based web applications development environment. Our methodology permits, in real time, two or more software testers to share, review, and comment on state-based test results of workflows under test. Collaborators can also examine test cases, edit them and initiate testing sessions. In the paper we describe the design of our collaborative testing methodology.

Keywords—Workflow; MVC; Visual collaborative testing; state-based testing

I. INTRODUCTION

Web development frameworks (WDFs) that incorporate both Workflow paradigms and the Model View Controller (MVC) [1] are known as WMVC-WDFs. Such frameworks guide the developer in generating the flow of control between the various components. Many industrial strength WMVC-WDFs such as Struts®[2] Ruby on Rails®[3], and PHP on TRAX® [4] have incorporated the concept of a workflow to define the transitions between the views of the application that are triggered by user interactions when performing a specific task. We refer to the applications that are instantiated using such frameworks as WMVC-based web applications. We also refer to an MVC-based web application developed in Ruby on Rail as R. Developing R is equivalent to creating a set of related workflows [5] where each workflow is made up of a set of models and a set of views that are linked by a set of transitions that define the relationships between those entities. As such, each workflow is associated with a state, and the states of all workflows define the global state of the instantiated web application.

Software testing, in general, has seen very little or no collaboration when it comes to enabling software testers to conveniently exchange information about a specific module under test. This is mainly due to the isolative nature of testing which is focused traditionally on batch processing of test suites and obtaining sizable volumes of textual feedback to indicate the testedness of a program under test; be it web or non-web program. Some work on non-web applications [6, 7, 8, 9] has been done to allow testers to see a visual feedback about a module or program under test. The extent of this visual feedback however, remained locked in a closed circle away from the possibility of real-time collaborative testing.

Our methodology permits real-time visual collaborative state-bases testing for R by allowing two or more software testers to share, review, and comment on test results of a specific workflow in R under test. The test results of a workflow under state-based testing are represented, in the form of reflective colors, directly on the visual representation of the workflows which are generated for R [10]. Color has been used to reflect the testedness of a program under test either by coloring statements or artifacts representing deeper program relationships like branches in the control flow graph that represents a program under test or paths for definition and uses of variables [11, 12, 13]. Coloring artifacts that represent a program constituents based on a specific testing criterion has been proven to help testers find faults. Therefore, our approach will help a team of collaborative testers to examine test cases, edit them and initiate testing sessions, observe testing results, and comment of these results, all in the spirit of engaging many opinions in real time to find faults. Our methodology is accomplished through the incorporation of an architectural layer in the WMVC of an R under test. The layer uses standard web technologies to maintain interoperability across different platforms. More on the implementation of this layer can be found in Section VI.

In this paper, we shall discuss the design and implementation of a collaborative state-based testing methodology for a WMVC-based web application under test. The methodology is provided to increase collaboration amongst software testers to increase the chances of finding faults. The rest of the paper is organized as follows. In Section II we discuss the testing literature, visual representation, management of workflows in WMVC web applications and the lack of related work that pertains to collaboration strategies for testing web applications in general and WMVC web applications in particular. In Section III, we describe the conceptual modeling of a WMVC web application, and how the visual workflows are generated. In Section IV we describe
how state-based testing can be applied to a WMVC-based web application. In Section V, we describe the design and implementation of the architectural layer which makes this visual collaboration and annotation possible in a state-based testing session of R. In Section VI, we describe how the layer is integrated into the WMVC of R. In Section VII, we describe the user interface of our collaborative testing environment and its immediate and potential benefits. In Section VIII, we conclude and give direction to potential future work.

II. RELATED WORK

A. Testing preliminaries

As with any development framework, WMVC-based web applications contain faults. Existing structural testing methodologies are applicable to two types of applications: non-web-based or (traditional) programs; and web-based applications. Traditional programs are developed using a variety of programming languages. These languages can be grouped based on their paradigms. Imperative and Object-Oriented (OO) languages are examples of such paradigms. Structural testing methodologies exist for testing such languages. For example, the work in [12] discusses various structural testing techniques that target imperative languages. The work in [14, 15] provides structural testing methodologies to test OO languages. These techniques and methodologies exploit mainly the internal structure that represents the control-flow of a program under test. The control-flow for a particular programming paradigm is in general unique. We next give a brief overview of structural testing criteria.

There are many structural or white-box testing criteria [12]. These criteria require the coverage of certain elements in the structure of the program. Most structural testing criteria are defined on an abstract graph model $g$ or structure that represents the control-flow structure of a program under test. For example, the structure of imperative programs is represented by $g$, which divides the statements of a program into a set of nodes. The nodes of $g$ are linked by directed edges that represent the control transfer among these nodes. Such control transfers are due mainly to the presence of predicate statements. Some structural testing criteria check for paths feasibility, that is, they require the coverage of certain paths in the $g$ that satisfy a particular condition. For example, the all-edges criterion requires a test suite (or a set of test cases containing input to the program under test) to cover a set of paths $P$ in $g$, such that every edge in the $P$ is covered by at least one path in $P$. Other structural testing criteria rely on the coverage of the data-flow interactions of a program under test. These criteria are known as data-flow testing criteria. One criterion in data-flow testing; the all-dus (dus is short for definition and uses) for example, requires a test suite to cover a set of paths $P$, such that $P$ contains every definition-clear path from each variable definition in a $g$ node containing the statement defining the variable, to each use reached by that definition through a path in the CFG, and each successor of that use. A definition-clear path with respect to a variable is a path in $g$, representing the function under test that contains no redefinition of this variable [12]. Therefore, a test adequacy criterion that is applied to $g$, is a predicate function that takes a program and a set of test cases as input, and produces a truth value that indicates if the program is sufficiently tested. A test suite is adequate with respect to a particular criterion if it satisfies the predicate associated with the criterion. In WMVC-based web applications that are instantiated from ROR or PHP on Trax, a framework-specific language is generally used. For example, ROR uses Ruby, and PHP on Trax uses PHP. Ruby is an OO language with procedural support, and as such, the aforementioned structural testing techniques and methodologies can be used to test this framework-specific language. However, the interaction model generated by the workflow introduces a set of new faults that is directly related to the requirements of the WMVC paradigm. More specifically, the separation between the business logic and the presentation layers and the way these layers interact, introduce an additional level of complexity that can in turn introduce additional faults. This set of faults requires additional testing to increase the quality assurance of these applications. Other structural testing techniques that test web applications are also available [16, 17, 18, 19, 20, 21]. These techniques concentrate mainly on testing each page or the interactions of these pages. On a page-level, testing targets the control-flow and/or data-flow of the code in a particular page. On an application-wide level, structural testing (control-flow and data-flow) can be applied to interacting pages of a web application under test. Both page-level and application-level techniques are not adequate to test WMVC web applications, since they ignore testing the workflow that exists among the various components of such applications. In this work, we introduce state-based testing strategy on workflows in WMVC web applications.

Black-box testing or functional testing is also available for both traditional and web applications. With black-box testing, the tester has no access to the implementation details, and the requirements of testing are specified based on identified functionalities (features) of a program under test. Such features are usually specified in the specifications of the program. For example, a test suite is adequate if it fully exercises all of the identified features (such as network connection) of a program under test. A feature is fully exercised if it operates as expected under a particular input. Functional testing is insufficient to assure the reliability of a program under test since it does not check any paths feasibility. Moreover, structural testing has proven to provide an effective mechanism over its functional counterpart in detecting faults [12].

In this work, we consider state-based testing for workflows. State-based testing targets the coverage of state transitions in a workflow. This testing criterion is ideal for the testing workflows in collaborative environment where a group of software testers can observe the reflection of the testing results and comments on them in real time. Our state-bases testing strategy has been adopted from [22, 23, 24]. More on the state-based testing strategy for WMVC-based web application will be found in Section IV.

B. Visual representation preliminaries

Various attempts at managing workflows in frameworks that integrate the workflow concept with the MVC have been taken. The majority of that work focused on the high level representation or business workflows [25, 26]. Some focused on creating formal notation to represent workflows and their
visual representation to represent the workflows [27], while others focused on resolving exception handling in workflows [28]. In a previous work [10], we showed how visual representation of workflows can be generated from an R in the hope of providing developers with a visual tool to manage workflows and check for inconsistencies.

As for visual collaborative testing strategies, we are unaware of any work that addresses the concept of visually monitoring and collaborating when applying state-based testing or any testing for that matter for WMVC-based web applications.

III. MODELING AND VISUALIZATION OF WMVC WEB APPLICATIONS

A. Modeling the WMVC Web application

Developing a WMVC-based web application like R in ruby on Rail requires the creation of a set of workflows \( W = \{w_1, w_2, \ldots, w_c\} \) where \( w_i \in W \) is a workflow in \( R \) is made up of a set of models \( M = \{m_1, m_2, \ldots, m_m\} \) and a set of views \( V = \{v_1, v_2, \ldots, v_\ell\} \) that are linked together based on a set of transition rules. The \( c_n \) or control in \( W \) always resides at either a model or view within a particular \( w_i \) in \( W \). The \( c_n \) could move within \( w_i \) or redirect to another workflow \( w_j \) based on a certain transition rule. The transition rule is based on an event. Each workflow \( w_i \) in \( W \) can then be represented with a Local State Machine or LSM\(_{(wi)}\) \( = (N, E, n_{wi}) \) where \( N \) is the set of local nodes representing the models and views within \( w_i \), node \( n_{wi} \) represents the entry node to \( w_i \) and \( E \) is the set of local transitions that link nodes in \( N \) based on a local transition. \( E \) also contains a dummy local transition that links \( n_{wi} \) to the first node \( n_1 \in N \). Figure 1 shows LSM\(_{(w1)}\) and LSM\(_{(w2)}\).

An inter-workflow transition is a transition between nodes in the same workflow, and is defined as follows: \( [n_1, n_2, l_e] \), such that \( n_1, n_2 \in N \) and \( l_e \) is the local event that is generated by \( n_1 \) and that moves the control from \( n_1 \) to \( n_2 \). As depicted in Figure 1 the transition between \( m_1 \) and \( v_1 \) in \( w_1 \) is an inter-workflow transition. The dummy transition \( [n_{w1}, n_1, d_e] \) automatically links node \( n_{w1} \) to node \( n_1 \in N \), such that \( d_e \) acts as a dummy event. A transition between a node in \( w_1 \) and a node in \( w_2 \) is an inter-workflow transition. For example, as depicted in Figure 1, the transition between \( v_1 \) and \( n_{w2} \) is an inter-workflow transition. The collection of all workflows and both their intra and inter workflow transitions comprise the Global State Machine (GSM) in \( R \). More formally, GSM = \( \{LSM_{(w1)}, LSM_{(w2)}, \ldots, LSM_{(ww)}\} \), where each global node LSM\(_{(wi)}\) in the GSM represents a workflow \( w_i \in W \). LSM\(_{(wi)}\) is a sub-graph that corresponds to the LSM of \( w \in W \). These sub-graphs or global nodes are linked by global transitions that cause the redirection from a local node in a particular LSM\(_{(wi)}\) to the entry node \( n_{wi} \) of \( (LSM_{(wi)}) \). We represent the global transition as follows \( [n, n_{wi}, g_e] \) where \( n \) is a local node in LSM\(_{(wi)}\), \( n_{wi} \) is the entry node of LSM\(_{(wi)}\), and \( g_e \) is the global event that is generated by \( n \) and that moves the control from \( n \) to LSM\(_{(wi)}\). We assume that \( R \) has a Home or index page that acts as the entry to the system through which the user can access the various functionalities of the web application. Therefore, any action in any controller can be reached if the control starts at the index page.

B. The Visual generation of Workflows in Ruby on Rail

Figure 2 depicts a web page of a login scenario and its Ruby code. Based on the conceptual modeling presented in subsection A of this section, and the parsing facilities found in Ruby on Rail, extracting workflows from an R under test is fairly easily accomplished. Once the workflows are extracted, the visual representations of these workflows are realized. Figure 3 shows the visual generation of the workflows that are extracted from the code in Figure 2. A detailed description of the implementation of workflow extraction and visual generation can be found in our earlier work [10].

IV. STATE BASED TESTING AN WMVC-BASED WEB APPLICATIONS

As previously mentioned, structural-based testing criteria such as control and dataflow have been applied to many programming languages and programming paradigms. Visual representations of a program’s constituencies like statements, branches and definition-uses have been presented. In our work, we apply state-based testing criteria to an \( R \) under test. Our testing approach investigates the transitions between states in intra and inter workflows, to ensure that at least the states and state transitions have been exercised. In state-based testing however, we are interested to see if a specific action taken by a user or the system can cause a transition between Views and Models in the \( R \); whereas the all-branches investigate if a value or a predicate statement causes the execution of the code to move from one node to another in the control-flow graph that represents a function or program under test. The testing results

---

*Fig. 1. A state machine with two workflows.*

*Fig. 2. The log in web page with its code.*

*Fig. 3. The inter and intra-workflow transitions in the login scenario.*
from the state-based testing criterion, when applied to \( R \), are reflected directly on the visual representation of the workflows of \( R \) which are generated in ‘shareable’ web pages. Both the visual representation of workflows and the reflection of testing results are shared by a team of software testers participating in a collaborative testing session using. This is possible due to the Collaborative Testing and Annotation Module (CTAM). More on the CMAT can be found in Section V. We next describe how state-based testing is applied to a WMVC-based web application in ROR.

A. State-based Testing for WMVC-based web application

Structural state-based test adequacy criteria are defined on the GSM and each LSM that corresponds to a workflow \( w \in W \). Examples of state-based control-flow test adequacy criteria is the all-transitions test adequacy criterion which requires the selection of test data that exercises every local transition in every LSM and every global transition in the GSM. Test \( t \) exercises a local transition \([n_1, n_2, l_i]\) in \( \text{LSM}_{w}[1] \) of \( w \in W \) in a ROR application \( R \) if \( t \) causes the execution of the action corresponding to \( w \) and this execution traverses the local edge \([n_1, n_2]\) in \( \text{LSM}_{w}[1] \) via the local event \( l_i \) where \( n_2 \) is the next local node at which the control will reside after \( n_1 \) generates \( l_i \). Similarly, a test \( t \) exercises a global transition \([n_1, n_w, g_e]\) in GSM of \( R \) if \( t \) causes the execution of the workflow containing \( n_1 \) and this execution traverses the global edge \([n_1, n_w]\) in GSM via the global event \( g_e \) where \( n_w \) is the entry node of \( \text{LSM}_{w}[1] \) at which the control will be redirected after \( n_1 \) generates \( g_e \). Let \( L \) be the set containing each \( \text{LSM}_{w}[1] \) of each workflow \( w \in W \). A test suite \( T \) is all-transitions adequate for \( R \) if for each \( \text{LSM}_{w}[1] \) in \( L \), and each local transition \([n_1, n_2, l_i]\) in \( \text{LSM}_{w}[1] \), there exists at least one test case \( t \) in \( T \) that covers \([n_1, n_2, l_i]\), and for each global transition \([n_1, n_w, g_e]\) in GSM of \( R \) there exists at least one test case \( t \) in \( T \) that covers this global transition.

B. Visual Communication of the testing results

After applying the set of test cases on \( R \), according to the state-based testing criterion that was previously described, we compute the trust of each transition in LSM and GSM and determine its corresponding color. The trust factor is related to (1) the number of test cases that traversed a transition with correct result, (2) the number of test cases that traversed a transition with incorrect result, (3) traversed transitions. The color of nodes and transitions can be green/greenish, yellow/ yellowish or completely red; in accordance with the trust factor. The greener a transition is the more trustworthy it is and the more yellow a transition is the more suspicious it becomes, and if a transition is completely red, it means there is a good cause to believe that a thorough investigation should be conducted on the code that caused the transition to exist in the workflow. The visual generation of workflows is made possible by using the GraphR library [29] to generate postscript files displaying the GSM of \( R \). The GraphR library provides a Ruby interface for dot [30] which is a visualization tool that draws directed and undirected graphs. An important feature of DOT is that it allows clusters (sub-graphs) to be drawn within their own distinct rectangle within a larger graph and to be connected together. GraphR creates a dot graph printer (dgp) object that takes as input arrays of the form \([a, b, c]\) to represent the directed edges of the graph to be drawn, such that \( a \) and \( b \) are the nodes (states) of the directed edge \((a, b)\) (state transition), and \( c \) is the label associated with this edge (transition). Using this dgp, we can access the nodes and the edges of the graph and set their attributes. The dgp can be also written to a file of a particular format such as Postscript, SVG, or GIF to view the graph that it represents. The GSM is represented as set of clusters connected together, where each cluster represents \( \text{LSM}_{w} \) corresponding to an action \( A \). Clicking the cluster of \( \text{LSM}_{w} \), displays its corresponding code in \( R \). Each global transition in the GSM, local transition in each LSM is colored according to its participation in testing (trust factor) as previously explained. These colors help developers having little or no knowledge about the testing theory to detect suspicious locations that may contain faults. More specifically, by examining the local and the global transitions of the state machines (GSM and LSMs) in the postscript file, the developer gets an insight of the suspicious actions where the faults could be located. These actions have suspicious colored local transitions, suspicious colored global transitions directed out of their nodes, and/or suspicious colored global transitions directed toward them. In state-based testing, testers can detect what transitions (global or local) in the GSM or LSMs were not executed at all after running the test suite, since such transitions will never be colored. By examining the states linked by such transitions, the collaborative testers can detect, for example, if there is a reference to a non-existent state. Moreover, by examining the suspicious colored transitions, whether local or global, the testers may detect other types of workflow-based faults such as incorrect state transition or cyclic state transition.

V. COLLABORATIVE TESTING AND ANNOTATION

Our Collaborative testing environment is supported through an architectural layer that is embedded in the WMVC layers of \( R \). the layer is referred to here as CTAM or Collaborative Testing and Annotation Module. CTAM is composed, as depicted in Figure 4, of four main components: Web Annotation Library (WAL); Communication Manager (CM); Test Suites Repository (TSR); and User Action Monitor (UAM). Each component plays a significant role in providing the collaborative testing support for software testers. We next describe the design and implementation of each component.

A. Web Annotation Library

The WAL provides the technical realization of WAs in CTAM. It employs temporary and shared WAs in the form of visual annotation that can be positioned at any desired location on the content of the web page in which, both the visual representation of workflows and the results of the state-based testing are being displayed. This of course will only be visible to participating testers in a collaborative session. A visual annotation can be in the form of text notes, drawings, images, and graphical shapes. WAL is implemented in JavaScript, and is accessible to the set of visual workflows representing the intra and inter workflows of an \( R \) under state-based testing. Each visual workflow can be thought of as a view \( v \) in \( V \) the
set of all visual workflows in \( R \). The WAL supports the following services: Annotation Creation/Presentation; Annotation Storage; and Synchronous Sharing.

The DOM tree representing the contents of a view \( v_i \) is a hierarchical collection of nodes such that each node represents some part of \( v_i \) like text or image. When a software tester uses the Annotation Creation Service to create a WA in \( v_i \), the data representing the WA is kept in a hidden volatile WA container called stickiesContainer. The stickiesContainer is a composite DOM node in \( v_i \). We use the appendChild() method of the DOM open API functions to first append the stickiesContainer to the <BODY> element of the DOM tree representation of \( v_i \), then append each sticky note to the stickiesContainer. For example, the (before and after the sticky note is attached) DOM tree representation which contains one button and one sticky note is depicted in Figure 5. The pink-like colored nodes with solid arrows represent the DOM tree before inserting a new sticky and the yellow nodes with the dotted arrows are the part of the DOM tree representation after the sticky insertion.

Another annotation facility that we have implemented is a free drawing tool that allows for free drawing anywhere inside the view window. This tool acts like a pencil that can be used for drawing using the mouse capturing coordinates. Our implementation of the free drawing annotation facility can be described as layering a transparent slide over a view in the form of a <DIV> element on top of all other elements in that view. We have devised an implementation trick to allow for freehand drawing on the transparent layer or canvas-like layer. This trick is manifested in the drawing of very small dash-like lines that are first obtained when “onMouseMove” events are initiated, and later concatenated on “onMouseUP” event appearing to have created a solid free hand drawing. This implementation technique makes use of the Vector Markup Language (VML), a scripting language used for rendering web-based graphical representations, to render and manipulate the display and presentation of the free hand drawing annotations.

The JavaScript class implementation of the free hand drawing facility is used to construct these connected lines and manipulate them according to the testing collaborator’s mouse movements. This class generates VML scripts or well-defined XML-representations specific graphical objects that are rendered in the browser. Analogous to our sticky notes implementation approach, the VML scripts collection are element nodes that are inserted in the DOM tree of the annotated view using a container-like DIV element (LinesContainer) that is used to hold the information of free drawings.

2) The Annotation Storage Component

This component allows collaborative testers to make these annotations visible. This is accomplished by first reading the data from the containers (LinesContainer and StickiesContainer) and then parsing them into proprietary XML elements that are stored in XML files on the server. The Annotation Storage Component uses a well defined XML storage representation. Each annotation type has a set of

- text: A text value of the sticky.
- x: The horizontal coordinate of the location of the sticky in a view.
- y: The vertical coordinate of the location of the sticky in a view.

Fig. 5. DOM tree example (before and after) of a view containing one button and a sticky note.
properties and their associated values. The XML schema of the free hand drawing and that of the sticky note differ only in the content of the <desc> node. We next show how the XML-based storage representation of WAs on the server side can be shared by a designed user.

3) The Synchronous Sharing Service

In CTMA WAs are used to communicate real-time testing feedback to end-users participating in the testing collaborative session. WAs in CTMA are characterized by contextual and temporal features. Their contextual features or specific coordinates in a view are technically realized using the Annotation Creation Component. Their temporal features are synchronous; providing software testers with the visual feedback on the annotated view without the need to refresh that view. This synchronous annotation service is implemented using AJAX (Asynchronous JavaScript And XML) which is an integration and extension of existing web technologies, mainly HTML, Cascading Style Sheets (CSS), and JavaScript to manipulate the DOM. The JavaScript language introduced a new set of built-in functions used in combination with some of the language tricks to request and receive data from the web server. XML is not necessarily utilized in AJAX; however, it is sometimes used to transport XML encoded data from the server to the client. We used AJAX to display synchronous interactive web annotations for end-users. With AJAX web applications become more responsive and interactive. The sharing of WAs between participating testers is accomplished using call-back functions that probe the XML-based storage of WAs on the server in search of new annotations. Once new WA data is found, it is retrieved and WAL uses the annotation information stored to render these annotations instantaneously. The XML storage of WAs allows for their portability and interoperability across different web browsers. In summary, WAL allows for graphical, contextual, shared synchronous WAs.

B. The Communication Manager

The Communication Manager (CM) is responsible for establishing and handling the communication channels between collaborating testers. It initiates an instantaneous text-based messaging window between the participating testers based on a request made by a tester party. The communication channel prevents data (WAs) propagation to unauthorized pairs. In other words, annotations are only displayed to software testers that are part of the same communication channels. As part of establishing a commutation channel, the CM also includes a service for testing collaborators’ identification which is implemented as user session. That is, when a software testing group initiates a collaborative session, the communication channel, among other things, captures and records the users’ session ids. This Id, as seen in Figure 7, is placed in the sticky note to differentiate between testing participators.

C. Test Suite Repository

The test suites repository contains all the test cases used by a group of software testers participating in a collaborative testing session. These test suites will be available to all testers when they are collaborating to allow them to experiment with the results of a testing case or suite. These test cases have been formatted according to the work in [31]. A quick example of testing script is:

- <enter,login,guest_1>
- <enter,password, 45rtded>
- <click, login,>

The above example shows a sample script for the login example in Figure 2.

D. User Action Monitor

Collaborating software testers are more likely to want to review testing actions. In this direction, we designed the UAM to monitor and record a collaborator’s actions while performing a testing task during a collaborative testing session. The data provided from the monitoring and recording phase allows any testing participator to review actions taken.

VI. INTEGRATING CTMA IN WMVC WEB APPLICATIONS

The CTMA layer and its components are integrated in any R under test. The integration process requires the extension of the framework, as depicted in Figure 6, is accomplished by incorporating the CTMA into R.

In our integration design each component in the CTMA layer interacts with a designated layer in the MVWF-based framework. This design feature makes it easy to deploy, and manage the CTAM layer within the framework. We next describe how to deploy each component in the layer. At the conceptual level, for each v_i in V', we deploy the WAL to provide collaborating testers with annotation services to create, and view annotations, respectively. At the technical level, we add an “include statement” in the form of a <SCRIPT src = "wa_lib.js"> tag to each v_i in the View layer. When a web page is composed of more than one view, we need to include or deploy the WAL in only one view mainly the one that renders the <HEAD> element. To support VML, the WAL adds a namespace attribute (<html xmlns:v="urn:schemas-microsoft-com:vml">) to the <HTML> tag of the views in the View Layer. The WA storage communicates with the View layer to propagate WAs created by support agents for a particular user on a particular view.

The UAM communicates with the Controller layer to capture the collaborator’s action during a testing session. This requires of course that the UAM communicates with the View Layer. The CM communicates with Controller layer to retrieve...
session identification. The TSR interacts with the Model layer as it is mainly designed to store test cases in the form of scripts.

Our CTAM collaborative testing environment does not need to modify the conventional web infrastructure. Furthermore, its deployment is unobtrusive. To comply with standard web technologies we employed state of the art client-side scripting technologies e.g. advanced JAVASCRIPT, DOM level 1, DOM2, AJAX, DHTML, and VML.

A. Deployment Requirements

Our implementation approach is tailored to deliver platform-independent, lightweight, and interoperable real-time online support tool. For this purpose, we draw on standard web technologies that are supported by most adopted browsers without requiring native code or specific plug-ins for end-users. In fact, it is operable on any DHTML-compliant (JavaScript-enabled) browser. Our primary concern was focused on allowing end-users working on any common client machine to make use of the visual instructions without requiring client customization or installation.

B. Important Design Features

In our design of CTAM, we tried to achieve a list of objectives that would, in our opinion, contribute to a successful online collaborative testing approach.

Human factor: The human factor is significant in the process of efficient collaborative delivery. Naturally testers would rather receive collaboration in a one on one scenario; however, the distance makes can create challenges that are only resolved by either an exhaustive testing output script or via online teleconferencing. Yet these challenges are likely to remain, and therefore having this collaborative environment can help tremendously in sharing ideas and thoughts on for example why something has gone wrong and how to uncover faults.

VII. USER INTERFACE

Figure 7 shows the interface of our collaborative testing environment. The interface is divided into the left hand side, and the right hand side. The left hand side contains 2 tabs: Local View for displaying individual workflows; and the other, Global View for displaying global workflows for any R under state-based testing. In the Local View tab, there is also a facility to show the corresponding code of a workflow to provide the testers with the correlation between the visual representation of the workflow and the code. To do that, each transition in the right hand side view is implemented as a hyperlink that takes the user to the related code once clicked. The right hand side, as seen in Figure 7, contains, in addition to the visual workflows, facilities to edit test cases (or the scripts), and execute a test case through the test button with drop down option. These facilities will be available to the group participating in a testing session. A sample collaborative testing session is depicted in Figure 7 which shows the visual workflows from Figure 3 of the “login.rb” and “update.rb” (not shown in this paper due to space constraints).

When the tester initiates a testing session using the state-based testing criterion, the user can click the button “test cases” and enter a set of test cases. These test cases are in the form of a script that is applied to the workflows to ensure that once
traversed, state, or state transitions, be they at the intra or inter level will be colored according to a coloring/trust factor that was previously mentioned in Section IV. These colors related to the visual workflow elements will help testing collaborators view and get an insight into what could be a particular problem. The testers participating in a collaborative testing session can then start to use the annotation services to share some facts about a testing session and its outcome. For example, in Figure 7, this particular collaborative testing session yielded three transitions (colored in red) in the first visual workflow. There are two software testers participating in this session with IDs 1 and 2 respectively. Figure 7 depicts how the tester with session ID 2 is commenting on the red color of the transition and asking if tester with Session ID 1 had the right combination of actions in the test case.

VIII. CONCLUSIONS AND FUTURE WORK

In this work, we provided a visual collaborative testing methodology that allows testers to see a testing result, comment on it and share those comments all in real time. The visual collaborative testing methodology was applied to WMVC-based web applications in the context of the programming environment Ruby on Rails which integrates the concept of workflows in its MVC development environment. The tool we showed in Figure 7 also allows collaborators to see and edit test cases which are designed to be in the form of scripts that will replace the user interaction with the application when state-based testing the web application that is instantiated from ROR. We have not yet performed an empirical study to see the effectiveness of both state-based testing and the annotation sharing in providing a team of testers clues to find faults; we suspect however, albeit with the appropriate training, this tool will play a pivotal role in helping a team of testers, analyze locate and catch faults. This is our plan for the future work.

REFERENCES


