Testing AJAX functionality with UniTESK

Yevgeny Gerlits
Lomonosov Moscow State University
Moscow, Russian Federation
Email: gerlits@ispras.ru

Abstract —**AJAX (Asynchronous JavaScript and XML) is a very promising technology for building interactive web applications. At the same time, AJAX significantly complicates the development of the client side of web applications. The paper demonstrates the possibility of utilizing the UniTESK test development technology for testing the client side functionality of AJAX web applications. Using UniTESK, test systems are developed for 8 AJAX web applications. Then the fault revealing capability of the test systems is evaluated in experiments.**

Keywords-AJAX; model based testing; UniTESK, asynchronous interface

I. Introduction

A classic web application is built around the notion of web pages and generally consists of a set of static web pages or server side programs that generate web pages. Such a web application is sufficiently inferior in interactivity to a web application developed with AJAX. The main reason is that the user communicates with the classic web application synchronously, that is he supplies input to the browser, e.g. clicks on a submit button or a link, and then waits until the browser refreshes the page. As opposed to this, web applications developed with AJAX can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.

At the same time, improving interactivity with AJAX sufficiently increases the complexity of the client side development. Using the JavaScript programming language, an AJAX application developer should implement an intermediate level between the browser and web-server which is responsible for handling user actions, managing browser-server dialog, and changing the interface according to web server responses. This task is hard enough to make a couple of faults.

In this paper, we consider the problem of testing the client side functionality of AJAX web applications. We show that qualitative tests can be elaborated using UniTESK [1, 2], an industrial model based test development technology designed in Institute for System Programming of Russian Academy of Sciences.

UniTESK was initially applicable to only systems with synchronous interfaces. After a period of time, an approach [3, 4, 5, and 6] was designed and implemented that extends this technology to asynchronous interfaces. Since then UniTESK has given a good account of oneself in testing several classes of complex applications with asynchronous interfaces such as internet protocols, components of a distributed operating system, and functions of the standard binary interface of Linux. Actually, these successful applications of UniTESK suggested that we apply this technology to AJAX web applications.

This work was supported by the RFBR (grant 09-01-00576-a)

UniTESK offers a test suite architecture consisting of a set of components that are used as building blocks to organize test systems. In the paper, we present a technique for developing these components so that the test system they form aims at revealing faults in the client side of the AJAX web application under test. We do not consider the problem of testing the server side of AJAX applications in this paper.

After presenting the approach to testing systems with asynchronous interfaces proposed by UniTESK and our technique of its use, we conduct several experiments in which we practically apply them. The obtained results show the applicability of UniTESK and the technique for testing the client side functionality of AJAX web applications. At the end of the paper, we present a comparison between our approach and the existing approaches to highlight the key advantages of UniTESK and our technique. We also discuss the main limitations and drawbacks of our approach.

The paper is structured as follows. Section II is devoted to the AJAX technology. We consider the architecture, the behavioral model, and the main features of a typical AJAX application. Section III outlines the UniTESK approach to testing systems with asynchronous interfaces. In section IV, we present our technique for testing the client side of AJAX web applications with UniTESK. We empirically evaluate the applicability of UniTESK and the technique in section V. Section VI compares our approach with the existing techniques. We conclude with a summary of our key contributions, and suggestions for future work in section VII.

II. AJAX

AJAX is an approach to web interaction that combines a set of well known technologies to achieve high interactivity of web applications. In this section, we consider the architecture, the behavior and the main features of a typical AJAX application.

Let us discuss AJAX applications comparing them with web applications that we call “classic”. The architectures of both the classic and AJAX applications are shown in Fig. 1.

A classic web application consists of a set of web pages. Some web pages may be described in static HTML

(Hypertext Markup Language) files; the others may be generated by the server side programs. A web page is displayed to the user, containing lists of links and form elements that allow the user to drill down to further web pages.

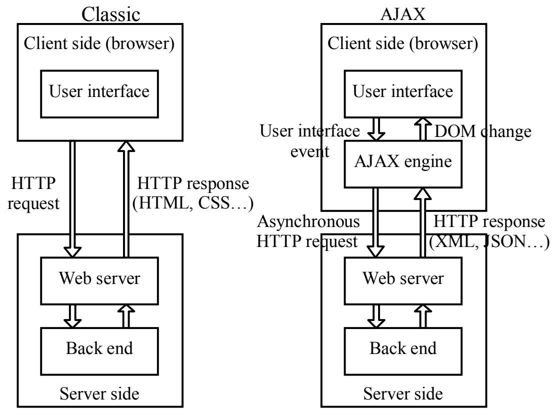

Figure 1. The architectures of classic and AJAX web applications

The main functionality of a classic web application is implemented at the server side. Some animation and additional functionality can be provided using client side programming languages and technologies, but it doesn't change the main behavioral model of the application. This model works as follows: the user supplies input to the browser, e.g. types a URL (Uniform Resource Locator), clicks on a hyperlink, or submits a form; the browser sends the HTTP (Hypertext Transfer Protocol) request for the URL to the web server; the web server responses with a new web page; the browser renders the page and waits for the user's next input.

The key features of classic web applications are as follows:

1. The user interacts with the web application synchronously, i.e. he requests for the next web page only after the response to the previous request has been handled by the browser and the appropriate web page has been displayed.
2. HTTP requests are issued for entire web pages and the entire page gets refreshed as a result of this action.
3. HTTP requests are issued by the browser, and HTTP responses are handled by the browser.
4. HTTP requests occur as a direct consequence of user actions.

As contrasted with a classic web application, the user communicates with an AJAX application asynchronously. The behavioral model proposed by AJAX works as follows:

1. The user performs an action on the web interface, e.g. clicks on a hyperlink, or a button.
2. An appropriate user interface event is fired.
3. The handler of this event, a JavaScript function, is called. It builds an asynchronous HTTP request, sets a callback function that will handle the response, and issues the request to the web server.
4. The web server replies with the data.
5. The callback function is called, it reads the data and changes the client side state that includes the DOM (Document Object Model) state, cookies, and global JavaScript variables.

According to this model, the user is able to go on working with the AJAX web application right after the user interface event handler has been executed, i.e. the user does not has to wait until the client-server dialog has been completed as it is happens in case of a classic web application. Because of the small size of the transferred data, the browser responds very quickly and the user does not feel any delay.

The key features of AJAX web applications are as follows:

1. The user interacts with the AJAX application asynchronously, i.e. he goes on working with the application while asynchronous HTTP requests are issued and responses are handled in the background.
2. The web server does not response with the entire web page, it responses with data that the client side JavaScript uses to dynamically refresh a small part of the currently displayed page.
3. HTTP requests are formed, issued and handled by JavaScript functions.
4. User actions can trigger the execution of JavaScript functions that may change the client side state and perform communication between the client and server, but JavaScript functionality is also able to work independently from user actions. It is usually achieved with special JavaScript functions that use timers to call other JavaScript functions.
5. The JavaScript programming language doesn't support multithreading. The browser uses one thread to handle user actions and execute JavaScript functions, including user interface event handlers and callback functions.
6. Concurrent HTTP requests are possible in some AJAX web applications, i.e. the next HTTP request may be issued before the response to the previous one has been handled.

In the paper, we consider AJAX applications, the client-server dialog of which complies with the behavioral model presented in this section. It doesn't matter which mechanism an AJAX application uses to perform asynchronous client-server communication. Let us note that the use of the XMLHttpRequest [7] object implies a sequence of HTTP responses to a single HTTP request. We take this fact into account.

We also suppose that an AJAX application itself is able to perform client-server communication independently from user actions.

III. Testing Asynchronous Interfaces with UniTESK

UniTESK is a model based test automation technology. It can be used for testing systems with synchronous and asynchronous interfaces. A synchronous interface implies that the subsequent action on the interface may be performed only after the interface has already responded to the previous action. The interface of a software system is considered to be asynchronous if this system can simultaneously interact with several other systems or interactions may be initiated by the system itself. The approaches and test suite architectures for testing systems with synchronous and asynchronous interfaces differ. We will discuss UniTESK implying only the asynchronous case in the remainder of the paper.

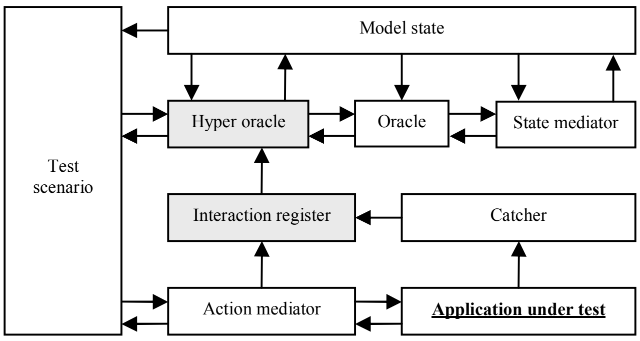

Figure 2. The UniTESK test suite architecture for testing systems with asynchronous interfaces

Each test system developed with UniTESK consists of a set of components. UniTESK defines the number of the components, their responsibilities and relationships. Fig. 2 contains the test suite architecture proposed by UniTESK for testing systems with asynchronous interfaces. Some of the components are already implemented and are used as is independently from the type of the application under test. Their representations have the gray background in the figure 2. The other components should be implemented by the tester and their implementations vary depending from the application under test. UniTESK provides formal descriptions to describe these components, extensions of some of the industrial programming languages to develop them, and software instruments to translate formalisms into the code in the target industrial programming language. The following formal descriptions are provided: *specifications, mediators,* and *scenarios*.

The test system developed with UniTESK supposes that the interface of the application under test consists of atomic operations of two types: *stimuli and reactions*. The test system supplies input to the application by means of applying stimuli to it. The application outputs through reactions that the test system evaluates. Reactions can be of two types: immediate or deferred. *Immediate reaction* is a reaction that is visible from outside immediately after affecting target system. When testing an application with an asynchronous interface, reactions to some stimuli may not be observed immediately because of the internal processes in the application. *Deferred reaction* is a reaction that is visible from outside later some time after a set of affecting target system.

Stimuli and reactions are the notions of UniTESK. The tester has to represent the real application interface through stimuli and reactions, and provide UniTESK with this interface. The following question could be set. Is it always possible to represent an arbitrary asynchronous interface through atomic stimuli and reactions? We haven't heard of a formal proof of it, but we also haven't heard of a contrary instance refuting it.

A formal interface of the application under test consisting of stimuli and reactions is fixed in *specifications*. Requirements to the application behavior are also fixed in *specifications* in the form of pre-conditions and post-conditions of stimuli and reactions, and *invariants of data types*. Specification also contains data structures that model the state of the application under test, i.e. describe the *model state*. The model state reflects the state of the application under test during testing. The requirements in specifications are imposed on the model state changes. *The pre-condition for the stimulus* describes constraints on the state, in which the test system is able to apply the stimulus. Violation of the precondition for the stimulus represents that the test is made incorrectly. The immediate reaction does not have its precondition. The post-condition for the stimulus and post-condition for the immediate reaction are the same things. *The post-condition for the stimulus* defines the requirements to the result of its application, i.e. to the state change and possibly the return value of the application operation the stimulus refers to, e.g. when applying the stimulus leads to the call of a public application operation that returns a value. *The pre-condition for the deferred reaction* describes if appearance of the reaction in the given state is possible. When precondition for the deferred reaction is violated, incompliance between the behavior of the application and its specification is registered. *The post-condition for the deferred reaction* checks compliance of the result obtained when the reaction emerges, to the expected one.

UniTESK defines the structure of specifications. The main goal of this structure is to provide the test completeness metric.

Specifications are translated into the test suite architecture components that take part in the verification of stimuli and reactions: *model state, action oracles* and *state mediators*.

To be able to verify requirements to stimuli and reactions, the test system should somehow link specifications to the application under test. *Action Mediator* component is generated from the formal description called mediator. It performs actions on the application under test, i.e. really applies stimuli. It also registers immediate reactions. The other component, implemented in the target programming language, registers the appearance of the deferred reactions. It is called *catcher*. The component that keeps information about the order of stimuli and reactions is called *interaction register*. The exact order of stimuli and reactions can not always be observed when testing a system with an asynchronous interface; therefore the UniTESK approach to testing systems with asynchronous interfaces was designed to be able to take advantage of the observable partial order of stimuli and reactions. So, interaction register usually keeps information about the detected partial order of stimuli and reactions.

The component of the UniTESK test suite architecture, which is called *test scenario*, is generated from the formal description of the same name and is used to combine operations that test logically related aspects of the application functionality. These operations are called *scenario functions*. Each scenario function applies a set of logically related stimuli to the application under test, supplying values for their parameters. To apply a single stimulus, the scenario function passes its call to test oracle, test oracle passes the stimulus to action mediator, and action mediator finally applies the stimulus.

Stimuli are applied and reactions appear during the execution of the scenario function. The completion of the scenario function indicates that all the stimuli have already been applied and all the reactions have been cached. After the scenario function has been executed, *hyper oracle* begins evaluating the observable behavior of the application under test. Information about the detected order of stimuli and reactions is utilized during the evaluation process as follows. The test system goes over all the possible orders of stimuli and reactions that conform to the partial order detected. For each particular order, each stimulus, and reaction test oracle checks the pre-condition, state mediator synchronizes the model state with the state of the application under test, and again test oracle checks the post-condition. If this procedure discovers at least one order, for which all the constraints on stimuli and reactions are met, the test system claims that the behavior of the application under test is acceptable.

To completely automate the execution of UniTESK tests and automatically generate sequences of test inputs, the developer has to define test scenario automata. A special component of the UniTESK test suite architecture goes over all the states of test scenario automata and calls each scenario function in each accessible state. To define test scenario automata, a function should be implemented that returns the state of test scenario automata after each scenario function call. In theory, the state of test scenario automata is constructed on the base of the model state. In practice, it may be an arbitrary function. This function allows the test system to construct test scenario automata incrementally during testing.

UniTESK imposes the following restriction on the behavior of the application under test: after applying a set of stimuli to the application, it demonstrates a set of reactions during a finite period of time and goes to a state in which no reactions appear spontaneously. Such states are called *stationary*. Stationary states allow the test system to perform the evaluation process and call the next scenario function at the state in which the previous scenario function finished.

In this section, we have only outlined the main characteristics of the approach we use for testing the client side of AJAX web applications. The details can be found in [3, 4, 5, and 6].

IV. TESTING AJAX APPLICATIONS WITH UNITESK

In this section, we present a technique for developing the UniTESK test suite architecture components so that the test system they form aims at revealing faults in the client side functionality of AJAX web applications.

A. The technique

In practice, functional testing of web applications aims at discovering faults of two types: general faults such as dead links and incorrect markup, and business logic faults concerning the behavior of the web application under test. Business logic faults are discovered when the web application under test incorrectly reacts to a logically related set of stimuli. The technique we propose in this section aims at discovering faults concerning the behavior of the client side functionality of AJAX applications.

At the first step, the requirements to the behavior of the client side of the AJAX application under test are extracted. When testing a web application, it is natural that there aren't any well-structured documents describing functional requirements. The probability of getting the requirements to the client side of the AJAX application is even lesser. We do not propose a method for the extraction of the requirements in the paper, because elaboration of such a method requires additional investigations and a separate paper is better to be written on the matter. We only assume here that the result of the requirements extraction procedure is a set of well-structured documents describing the requirements to the client side of the AJAX application under test.

At the second step, the extracted requirements are to be formally fixed in specifications in the form of pre-conditions and post-conditions of stimuli and reactions, and invariants of data types. To be able to formalize the requirements using the software contracts proposed by UniTESK, the tester must represent possible interactions of the client side functionality with its environment as a set of stimuli and reactions.

We believe that an adequate model is shown in Fig. 3. This model conforms to the behavioral model of a typical AJAX web application presented in section II, but it only concerns the client side of the application. An individual action on the application interface represents a stimulus if this action leads to the modification of the client side state or if an asynchronous HTTP request is issued. A user interface event occurs as a result of such an action. The handler of this event is called. It may change the client side state or issue an asynchronous HTTP request. The result of its execution is modeled as a reaction. The new proxy server component of the test system intercepts the request issued by the user interface event handler. It in turn issues the HTTP response. It is modeled as a stimulus. The callback function is called that handles this response. The client side state can be modified as a result of its execution or something else can happen. It is modeled as a reaction.

The client side functionality of the AJAX web application under test may change the client side state or issue an asynchronous HTTP request independently. Such an activity is modeled as a rection.

Having this model, the requirements to the stimuli and reactions can be formalised. Stimuli are specified trivially. A reaction results to the client side state change and possibly an asynchronous HTTP request. So, the postcondition for the reaction should asses the client side state change and the HTTP request in case the request is issued as a result of the reaction.

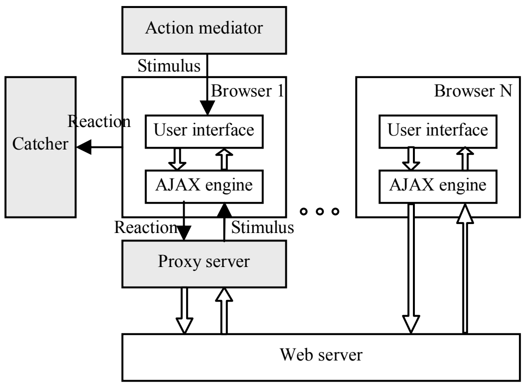

Figure 3. Interactions of the client side of an AJAX web application with its environment

In order that the test system may really verify the behavior of the AJAX application, action mediator, catcher and proxy server test suite architecture components are implemented at the third step.

Action mediator contains functions that programmatically perform actions on the application interface.

Catcher must detect the reactions, and extract and save the client side state changes after them. The single threaded nature of JavaScript helps a lot for the extraction of the client side state changes. If the extraction of the client side state change is accomplished by a JavaScript function, it is guarantied that there aren't another activity that modifies the client side state at the same time.

Proxy server is not a part of the UniTESK test suite architecture. It is a new component specifically desigent to support testing of AJAX applications. Proxy server has two responsibilities:

- intercept asynchronous HTTP requests;
- apply stimuli that model the responses of the target web server.

The use of proxy server allows modeling the real situation of multiple users working with a single web server. The server side state can be changed by the users. Proxy server is able to respond taking the possibility of the server side state changes into account.

The client side state changes and the intercepted HTTP requests are used by the state mediator to synchronize the state of the requirements model with the state of the AJAX application under test during the verification procedure.

At the fourth step, specifications are used to determine the test coverage criteria. The higher is the criteria, the more complecate are test scenarios.

At the fifth step, test scenatios are developed so that the choosen test coverage criteria could be achieved during testing.

Testers often do not take faults concerning multiple asynchronous HTTP requests into account, because of their low probability. A typical example of such a fault can be the following: the second asynchronous HTTP request is issued before the response to the previous one has come; due to network delay, the response to the second request comes before the response to the first one; the callback function that handles the second response removes a DOM element; the response to the first request comes; its callback function crashes trying to access the deleted DOM element. It is obvious, that the proposed technique for modeling stimuli and reactions alows developing scenatio functions aiming at testing multiple asynchronous HTTP requests.

B. Application domain

The approach to testing systems with asynchronous interfaces proposed by UniTESK has two main application conditions:

1. A formal interface consisting of atomic stimuli and reactions may be provided for the real interface of the application under test. This formal interface should adequately model the real application interface.
2. After responding to a set of stimuli, the application under test must go to a stationary state in which no reactions can appear spontaneously.

The technique we have just presented explains how to get a formal interface complying with the first condition.

As concerns to the second condition, we have mentioned in section II of the paper that AJAX web applications may have client side functionality that changes the client side state and communicates with the server independently from user actions and at an unpredictable time. Formally, there are no stationary states in such applications. If such functionality is out of the scope of testing, it usually may be ignored or deactivated by hand. If the test system must take such functionality into account, it has to model stationary states. For instance, the test system may artificially execute a piece of JavaScript during the evaluation process in order that the application under test does not change the client side state or issue an HTTP request.

At the moment, we can not imagine a client side functionality of an AJAX web application that can not be modeled and tested using UniTESK and our technique.

V. EMPIRICAL EVALUATION

In order to evaluate the applicability of the UniTESK test development technology and the technique of its use presented in the paper for testing functionality of the client side of AJAX web applications, we perform a set of experiments.

We collect 8 AJAX design patterns. Each pattern describes how the objects, components, and levels constituting the AJAX web application should interact in order that the application could respond to user actions in a certain way or a certain interactivity effect could be achieved. The patterns primarily describe client sides of AJAX web applications. Implementing them allows us to get AJAX applications that both implemented differently and behave differently.

We implement each pattern in an AJAX web application. So, we have 8 AJAX applications. After that, using the UniTESK technology and our technique, we create a test system for each AJAX web application developed. In order to assess the fault-revealing capability of the test systems we intentionally introduce faults into the source code of the AJAX web applications, perform testing and count the percent of the faults revealed. This section presents the results of our experiments.

A. AJAX design patterns

Here we briefly introduce 8 AJAX design patterns and their implementations for which we develop test systems. Detailed description of the patterns can be found in [8, 9, and 10].

Pattern: Explicit Submission. *Problem:* How can information be submitted to the server? *Solution:* Instead of automatically submitting upon each browser event, require the user to explicitly request it, e.g. submit upon a button click. *AJAX application:* A simple authorization form.

Pattern: Periodic Refresh. *Problem:* How can the application keep users informed of changes occurring on the server? *Solution:* The application periodically issues asynchronous requests to gain new information, e.g. one request every five seconds. *AJAX application:* An application alerts the user as a new comment has been added.

Pattern: Submission Throttling. *Problem:* How can information be submitted to the server? *Solution:* Instead of submitting upon each JavaScript event, retain data in a browser-based buffer and automatically upload it at fixed intervals. *AJAX application:* An application that submits a single field periodically as changes are made.

Pattern: Predictive Fetch. *Problem:* How can you make the AJAX application respond quickly to user activity? *Solution:* Have the application anticipate likely user actions and call the server in preparation. *AJAX application:* An application that preloads the next page of the article.

Pattern: Browser-side Cache. *Problem:* How can you make the AJAX application respond quickly to user activity? *Solution:* Retain server results in a browser-side cache. Whenever the application performs an asynchronous request, it first checks the cache. If the query is held as a key in the cache, the corresponding value is used as the result, and there is no need to access the server. *AJAX application:* A simple calculator that performs calculations on the server and retains the results in a client-side cache.

Pattern: Guesstimate. *Problem:* How can you cut down on calls to the server? *Solution:* Instead of requesting information from the server, use a historical data and make a reasonable guess on the client. *AJAX application:* An approximate calculation of the number of registered users.

Pattern: Pseudo-threading. *Problem:* AJAX web applications are single-threaded. Some of them require complex processing on the client. If the thread of execution is busy performing such processing, users won't be able to perform input. *Solution:* Instead of solving the entire problem at once and returning, a processing function is called once in a while, incrementally processes a bit more of the problem, before yielding. *AJAX application:* Sorting of a big table on the client.

Pattern: Multi-stage Download. *Problem:* How can you optimize downloading performance? *Solution:* Break content download into multiple stages, so that faster and more important content will arrive first. *AJAX application:* An application that downloads additional links after the main content of the article has been downloaded.

B. Experiments

To implement test systems for the AJAX applications introduced in the previous subsection, we exploit both the Java and JavaScript programming languages. The JavaTESK [11] toolkit is used to implement the UniTESK test suite architecture components and run the test suites developed. The Selenium Remote Control [12] testing tool is used to drive the browser, programmatically perform actions on the web interface, and access the resulting DOM states. We exploit Mozilla Firefox as a browser in our experiments. Our technique of the use of UniTESK introduces the proxy server component in the test suite architecture. We implement this component using the Java programming language. It is universal, i.e. implemented once it is included in all the test systems.

We perform five experiments for each AJAX web application and corresponding test system. Thus forty experiments are conducted in the total. Each experiment consists in introducing a single fault into the source code of the application, running the corresponding test system on the application, and analyzing the test results. Table 1 summarizes the results of the experiments performed.

TABLE I. THE RESULTS OF THE EXPERIMENTS

AXAX application for	**Introduced**	**Revealed**	**%**
Explicit Submission	5	5	100%
Periodic Refresh	5	4	80%
Submission Throttling	5	4	80%
Predictive Fetch	5	5	100%
Browser-side cache	5	5	100%
Guesstimate	5	3	60%
Pseudo-threading	5	4	80%
Multi-stage download	5	4	80%
TOTAL	40	34	85%

Here are some examples of the faults introduced: building incorrect HTTP requests in JavaScript functions, removing user interface event handlers, wrong modifications of the DOM, removing an XMLHttpRequest object from the pool of XMLHttpRequest objects, setting timers with wrong time intervals, removing identifiers of HTML elements and etc. All the faults appear at the client side of the AJAX applications.

The test systems reveal 85% (in the mean) of all the errors introduced. We believe it is a good result that confirms the applicability of UniTESK and the technique of its use for testing functionality of AJAX web applications. It is worth noting that the percentage of the faults revealed depends on the quality of the test systems developed.

VI. COMPARISON WITH THE EXISTING APPROACHES

We didn’t manage to discover another approach specifically designed for testing the client side of AJAX applications. In this section, we present an overview of the existing AJAX functional testing approaches. The approaches test an AJAX application as a whole; therefore they are able to reveal faults in both the client side and server side of AJAX applications. We compare them with the approach we propose in the paper, i.e. the UniTESK technology complemented with the technique of its use.

A. Approaches proposed by the scientific community

We succeed in discovering three approaches specifically designed for functional testing of AJAX web applications:

- Invariant Based Testing [13];
- State Based Testing [14] ;
- Search Based Testing [15].

All the approaches use a FSM (Finite State Machine) model of the AJAX web application under test to produce tests; therefore we label them as *FSM based test generation approaches*.

The Invariant Based Testing approach is rather directed to revealing faults in dynamical DOM states such as dead links, incorrect markup, and the absence of widgets, DOM elements, and error messages; than organizing complex test situations in which the test system applies a set of logically related stimuli to the application and verifies the reactions to these stimuli. Accomplishing the latter is the primary purpose of the approach we propose in the paper, i.e. the UniTESK technology complemented with the technique of its use. So, the Invariant Based Testing approach and our approach aim at revealing faults of different types; therefore there is no point in their further comparison.

The State Based Testing approach divides test creation into two stages. At the first stage, the FSM model of the AJAX application under test is constructed on the base of a set of preliminarily recorded real execution traces of the application. The states of the FSM are abstracted from the real DOM states. The transitions are the JavaScript method invocations triggered by user events or server responses and modifying the DOM. At the second stage, tests are generated on the base of the traversal of the FSM extracted at the first stage. The test generation is accomplished so that the generated tests are able to automatically reveal faults leading to the modification of a correct sequence of states in the FSM model of the application.

Because the FSM model is constructed on the base of the real behavior of the application, the approach is expected to show its best in regression testing. The authors strengthen the approach by providing the ability to express general requirements to the behavior of the application in the form of pre-conditions and post-conditions. This feature of the approach makes it possible to apply it for functional testing. An advantage of the software contracts proposed by UniTESK is that they additionally provide test coverage criteria. The State Based Testing approach deals with concurrent asynchronous HTTP requests, but it only warns whether there may be a problem. As opposed to this, our approach reveals faults concerning multiple asynchronous HTTP requests. The authors of State Based Testing claim that their approach is a good complement to the classic functional testing.

The Search Based Testing approach is based on the State Based Testing approach. The authors propose a technique that enhances the fault revealing capability of the tests generated. The main features of the approach remain the same.

A common advantage of the State Based Testing and Search Based Testing approaches over our approach is that they are better automated. The approaches are designed for testing only AJAX applications, the authors of the approaches tried to automate them as much as possible. In contrast to the approaches, the UniTESK technology doesn't take the AJAX specific features into account, because it was developed to be applicable for general purpose software. That is why developing some of the UniTESK test suite architecture components is a fairly labor-intensive task. For instance, special functions should be implemented in order that action mediator could programmatically perform actions on web interface elements. Each particular AJAX application requires its own functions because there aren't two AJAX applications that have the same interface. Other functions should be implemented in order that catcher could get DOM states after the reactions.

B. Approaches used in industrial practice

We examined existing test automation tools that support functional testing of web applications. The tools that are positioned as AJAX test automation tools implement the Capture and Playback [16] approach. According to the approach the tester records the user actions; saves them in a script; enhances the recorded script with verification points, where some property or data is verified against an existing baseline; plays back the script and observes the results. The Capture and Playback approach is very useful for regression testing. It is also widely used for functional testing of classic web applications.

In order to support testing of AJAX applications, Capture and Playback testing tools implement either a method for automatically detecting responses to asynchronous HTTP requests or a method for detecting DOM state changes. Such a method allows a Capture and Playback testing tool to determine whether the application has already responded to the user action during the playback stage. The Capture and Playback approach supporting AJAX is implemented in IBM Rational Functional Tester [17], SWEA [18], and many other test automation tools. The Capture and Playback approach doesn't aim at creating complex test sequences like the approach we propose in this paper. Using it leads to the generation of a big amount of test scripts. A script usually verifies a sequence of possible user actions. Week modularity is a common disadvantage of such scripts. As opposed to this, the test suite architecture is one of the most competitive advantages of UniTESK.

The most flexible of the existing AJAX functional testing techniques is to use a combination of a unit testing framework and a software library which makes it possible to programmatically perform actions on the application interface and then access the resulting DOM state. An example of such a technique is the JUnit [19] unit testing framework complemented with the Selenium Remote Control testing tool. By analogy with the Capture and Playback testing tools, AJAX support is limited to designing and implementing either a method for detecting responses to asynchronous requests or a method for detecting DOM state changes. Let us note that this technique is flexible because it provides minimal support for test automation. In fact tests are handmade, but can be executed automatically.

VII. CONCLUSION

In this paper, we demonstrate the applicability of the UniTESK test development technology for testing the client side functionality of AJAX web applications. We outline the approach to testing systems with asynchronous interfaces proposed by UniTESK, present the technique for modeling and testing AJAX applications with UniTESK, practically evaluate UniTESK and our technique, and compare our approach with the existing approaches.

Though UniTESK can be used to develop test systems for AJAX web applications, UniTESK is not an AJAX-specific testing technique. Developing tests for AJAX with UniTESK is a very labor-intensive task. The future work may consist in enhancing the automation level of the approach we propose in the paper.

In this paper, we ourselves develop AJAX applications. Then we apply UniTESK to them. In our future work, we should apply UniTESK to a couple of applications really working in Internet.

Our approach can only be used for testing the client side functionality of AJAX web applications. On the one hand, the approach is directed to the client side faults that are typical and specific for AJAX web applications. On the other hand, we do not test the server side at all. Future investigations may consist in designing an AJAX testing technique that will take both the client side and the server side faults into account.

REFERENCES

[1] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko, "UniTesK test suite architecture," Proc. FME 2002, LNCS 2391, Springer-Verlag, 2002, pp. 77-88.

[2] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko, "UniTesK: Model Based Testing in Industrial Practice," Proc. the 1st European Conference on Model-Driven Software Engineering (ECMDSE), Nuremberg, Germany, Dec. 11-12, 2003, pp. 55-63.

[3] V. Kuliamin, A. Petrenko, N. Pakoulin, I. Bourdonov, and A. Kossatchev, "Integration of Functional and Timed Testing of Real-time and Concurrent Systems,". Proc. of PSI 2003, LNCS 2890, Springer-Verlag, 2003, pp. 450–461.

[4] A. Khoroshilov., "Specification and Testing Systems with Asynchronous Interfaces," Preprint of the Inst. for System Programming, Russ. Acad. Sci., Moscow, 2006.

[5] V. Kuliamin , A. Petrenko, and N. Pakoulin, "Practical Approach to Specification and Conformance Testing of Distributed Network Applications," Proc. ISAS'2005, Berlin, Germany , April 25-26, 2005, pp. 60-73.

[6] N. Pakulin and A. Khoroshilov, "Development of formal models and conformance testing for systems with asynchronous interfaces and telecommunications protocols," Programming and Computer Software, vol. 33, number 6, Nov. 2007, pp. 316-335, doi: 10.1134/S0361768807060035.

[7] XMLHttpRequest object specification: http://www.w3.org/TR/XMLHttpRequest/

[8] M. Mahemoff, Ajax design patterns. Sebastopol, CA: O'Reilly Media, Inc, 2006

[9] Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett, Proffessional Ajax 2nd edition. Indianapolis, Indiana: Wiley Publishing, Inc., 2007

[10] Wiki on AJAX containing a comprehensive collection of AJAX design patterns:

http://ajaxpatterns.org/

[11] JavaTESK toolkit for testing Java applications with UniTESK:

http://www.unitesk.com/

[12] Selenium Remote Control web application functional testing tool: http://seleniumhq.org/projects/remote-control/

[13] Ali Mesbah and Arie van Deursen, "Invariant-based automatic testing of Ajax user interfaces," Proc. the 31st International Conference on Software Engineering (ICSE'09), IEEE Computer Society, 2009, pp. 210-220.

[14] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of Ajax web applications," Proc. 1st IEEE Int. Conference on Sw. Testing Verification and Validation (ICST'08), IEEE Computer Society, 2008, pp. 121-130.

[15] A. Marchetto, P. Tonella, "Search-based testing of Ajax web applications," Proc. the 2009 1st International Symposium on Search Based Software Engineering, May 13-15, 2009, pp. 3 – 12.

[16] G. Meszaros, "Agile regression testing using record and playback," Proc. the conference on Object Oriented Programming Systems Languages and Applications, 2003, pp. 353-360.

[17] Rational Functional Tester web application functional testing tool:

http://www-01.ibm.com/software/awdtools/tester/functional/

[18] SWEA web application functional testing tool:

http://webiussoft.com/

[19] JUnit unit testing framework:

http://www.junit.org/