Automatic Testing System for Web Application

ABSTRACT:

AJAX-based Web 2.0 applications rely on stateful asynchronous client/server communication, and client-side runtime manipulation of the DOM tree. This not only makes them fundamentally different from traditional web applications, but also more error prone and harder to test. We propose a method for testing AJAX applications automatically, based on a crawler to infer a state-flow graph for all (client-side) user interface states. We identify AJAX-specific faults that can occur in such states (related to, e.g., DOM validity, error messages, discoverability, back-button compatibility) as well as DOM-tree invariants that can serve as oracles to detect such faults. Our approach, called ATUSA, is implemented in a tool offering generic invariant checking components, a plugin mechanism to add application-specific state validators, and generation of a test suite covering the paths obtained during crawling. We describe three case studies, consisting of six subjects, evaluating the type of invariants that can be obtained for AJAX applications as well as the fault revealing capabilities, scalability, required manual effort, and level of automation of our testing approach.

ALGORITHM:

Algorithm 1. Crawling process with pre/postCrawling hooks

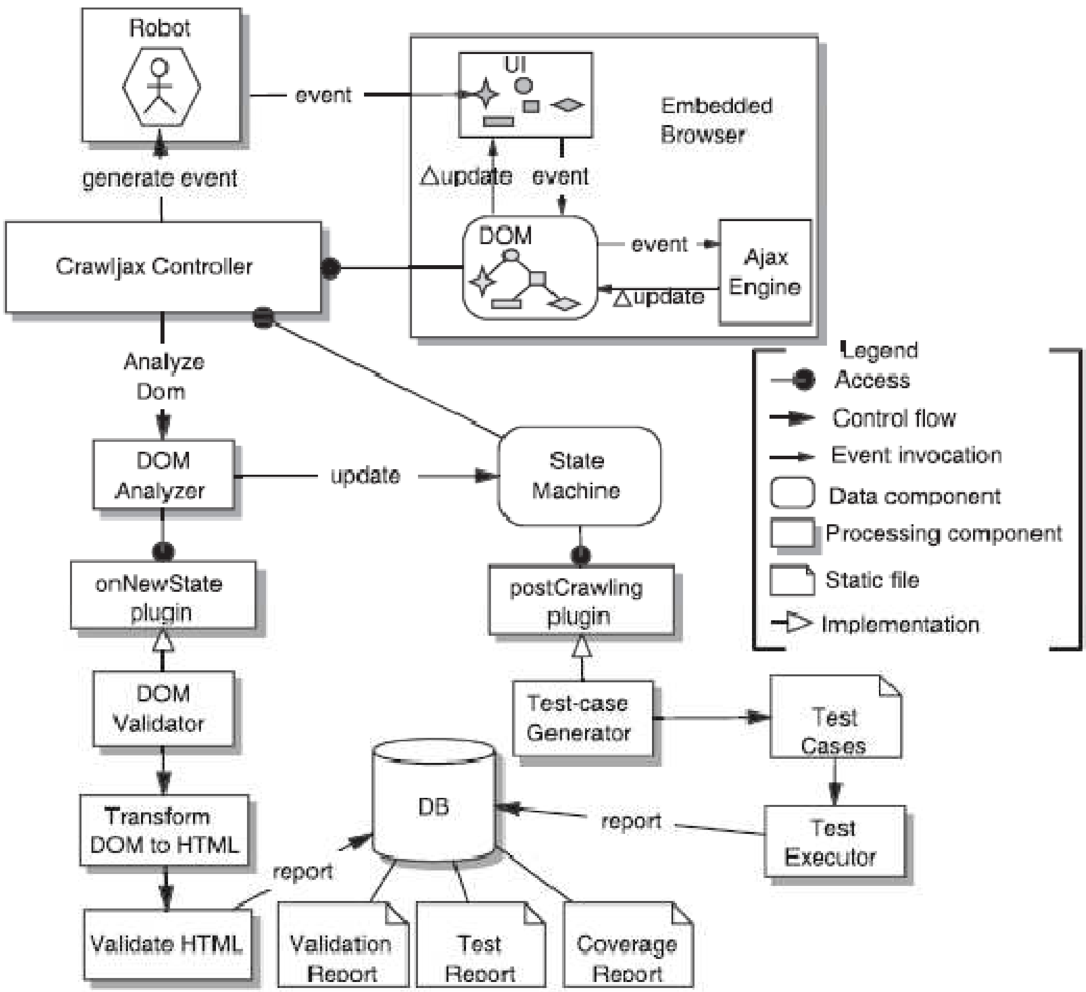
```
procedure START (url, Set tags)
browser initEmbeddedBrowser(url)
robot initRobot()
sm initStateMachine()
preCrawlingPlugins(browser)
crawl(null)
postCrawlingPlugins(sm)
end procedure
procedure CRAWL (State ps)
cs sm.getCurrentState()
_update diff(ps, cs)
f analyseForms(_update)
Set C getCandidateClickables(_update, tags, f)
for c 2 C do
generateEvent(cs, c)
```

```
end for
end procedure
```

Algorithm 2. Firing events and analyzing AJAX states

```
procedure GENERATEEVENT (State cs, Clickable c)
robot.enterFormValues(c)
robot.fireEvent(c)
dom browser.getDom()
if stateChanged(cs.getDom(), dom) then
xe getXpathExpr(c)
ns sm.addState(dom)
sm.addEdge(cs, ns, Event(c, xe))
sm.changeToState(ns)
runOnNewStatePlugins(ns)
testInvariants(ns)
if stateAllowedToBeCrawled(ns) then
crawl(cs)
end if
```

```
sm.changeToState(cs)
if browser.history.canGoBack then
browser.history.goBack()
else
{We have to back-track by going to the initial
state.}.
browser.reload()
List E sm.getPathTo(cs)
for e 2 E do
```


Robot
event
UI
Embedded Browser
generate event
Δupdate
event
DOM
event
Ajax Engine
Crawljax Controller
Δupdate
Analyze Dom
Legend
Access
Control flow
Event invocation
Data component
Processing component
Static file
Implementation
DOM Analyzer
update
State Machine
onNewState plugin
postCrawling plugin
DOM Validator
Test-case Generator
Test Cases
DB
report
Test Executor
Transform DOM to HTML
report
Validate HTML
Validation Report
Test Report
Coverage Report

EXISTING SYSTEM:

In order to improve the dependability of AJAX applications, static analysis or testing techniques could be deployed. Unfortunately, static analysis techniques are not able to reveal many of the dynamic dependencies present in today's web applications. Furthermore, traditional web testing techniques are based on the classical page request/ response model, not taking into account client side functionality. Recent tools such as Selenium1 offer a capture-and-replay style of testing for modern web applications. While such tools are capable of executing AJAX test cases, they still demand a substantial amount of manual effort from the tester

PROPOSED SYSTEM:

The goal of this paper is to support automated testing of AJAX applications. To that end, we propose an approach in which we automatically derive a model of the user interface states of an AJAX application. We obtain this model by "crawling" an AJAX application, automatically clicking buttons and other UI-elements, thus exercising the clientside UI functionality. In order to recognize failures in these executions, we propose the use of invariants: properties of either the client-side DOM tree or the derived state machine that should hold for any execution. These invariants can be generic (e.g., after any client-side change the DOM should remain W3C-compliant valid HTML) or application-specific (e.g., the home-button in any state should lead back to the starting state).

We offer an implementation of the proposed approach in an open source, plugin-based tool architecture. It consists of a crawling infrastructure called CRAWLJAX,2 as well as a series of testing-specific extensions referred to as ATUSA. We have applied these tools to a series ofAJAX applications.Wereport on our experiences in this paper, evaluating the proposed approach in terms of fault-finding capabilities, scalability, automation level, and the usefulness of invariants.

MODULES:

- ✓ The State-Flow Graph
- ✓ Inferring the State Machine
- ✓ Detecting Clickables
- ✓ Creating and Comparing States

- ✓ Navigating the States
- ✓ TESTING AJAX STATES THROUGH INVARIANTS

MODULES DESCRIPTION:

The State-Flow Graph

The crawler we propose is a tool that can exercise client-side code and identify elements10 that change the state within the browser's dynamically built DOM. From these state changes, we infer a state-flow graph, which captures the states of the user interface and the possible event-based transitions between them.

Inferring the State Machine

The state machine (line 4 Algorithm 1) is created incrementally. Initially, it only contains the root state and new states are created and added as the application is crawled and state changes are analyzed (lines 7-8 Algorithm 2). The following components participate in the construction of the graph:. CRAWLJAX uses an embedded browser interface (with different implementations: IE, Firefox, and Chrome) supporting all technologies required by modern dynamic web applications; . a robot is used to simulate user input (e.g., click, hover, text input) on the embedded browser; . the finite state machine is a data component maintaining the state-flow graph, as well as a pointer to the current state; . the controller has access to the browser's DOM and analyzes and detects state changes. It also controls the robot's actions and is responsible for updating the state machine when relevant changes occur on the DOM tree.

Detecting Clickables

To illustrate the difficulties involved in crawling AJAX. This is a highly simplified example, showing how an onclick event listener can be attached to a DIV element at runtime through JAVASCRIPT. Traditional crawlers as used by search engines simply ignore all such clickables. Finding these clickables at runtime is a nontrivial task for any modern crawler. To tackle this challenge, CRAWLJAX implements an algorithm in which a set of candidate elements (line 13 Algorithm 1) are exposed to an event type (e.g., click, mouseover) (line 3 Algorithm 2). In an automatic mode, the crawler examines all elements of the type A, DIV, INPUT, and IMG since these elements are often used to attach event listeners. If the user wishes to define their own criteria for selection, this list can be extended or adapted. The candidate clickables can be labeled as such based on their HTML tag element name and attribute constraints. For instance, all elements with a tag SPAN having an attribute class="menuitem" can be set to be considered as candidate clickable. For each detected candidate element on the DOM tree, the crawler fires an event on the element in the browser to analyze the effect. A candidate clickable becomes an actual clickable if the event fired on the element causes a DOM change in the browser.

Creating and Comparing States

After firing an event on a candidate clickable, the algorithm inspects the resulting DOM tree to see if the event results in a modified state (line 5 Algorithm 2). If a similar state is part of the state flow graph already, merely an edge is created, identifying the type of click and the location clicked. If the next state is not part of the graph already, a new state is created and added first .

The level of abstraction achieved in the resulting stateflow graph is largely determined by the algorithm used to compare DOM trees (which reflect the states in the flow graph). A generic and effective way is to use a simple string edit distance algorithm such as Levenshtein. This has the advantage that it does not require application-specific knowledge and that the algorithm can be fine-tuned by means of a similarity threshold (between 0 and 1). Alternatively, we propose the use of a series of “comparators” that each can compare specific aspects of two DOM trees. Each comparator can eliminate specific parts of the DOM tree, such as (irrelevant) attributes, time stamps, or styling issues. The resulting simplified DOM tree is subsequently pipelined to the next comparator

Navigating the States

Upon completion of the recursive call, the browser should be put back into the previous state. A dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the “Back” function of the browser is usually insufficient. To deal with this AJAX crawling problem, we save information about the elements and the order in which their execution results in reaching a given state. We can then reload the application and follow and execute the elements from the initial state to the desired state. CRAWLJAX adopts XPath to identify the clickable elements. After a reload or state change, DOM elements, can easily be deleted, changed, or replaced. As a consequence, the XPath expression used for navigation can become invalid. To tackle this problem, our approach uses a mechanism called Element Resolver (line 23 Algorithm 2), which examines the clickable elements before they are used to make state transitions.

This examination is needed to make sure we have access to the correct element. To detect the intended element persistently, we use various (saved) properties of the element such as their attributes and text value. Using a combination of these properties, our element resolver searches the DOM for a match, which gives us some degree of reliability in case clickables are removed or changed. Note that despite our element resolving mechanism, because of side effects of server-side state there is no guarantee that we find the same element on the DOM-tree and can reach the exact same state.

TESTING AJAX STATES THROUGH INVARIANTS

With access to different dynamic web states we can check the user interface against different constraints. We propose to express those as invariants, which we can use as an oracle to automatically conduct sanity checks in any state. Although the notion of invariants has predominantly been applied to programming languages for software evolution and verification, we believe that invariants can also be adopted for testing modern web applications to specify and constrain DOM elements' properties, their relations, and occurrences. In this work, we distinguish between generic and application-specific invariants on the DOM-tree, between DOM-tree states, and on the runtime JAVASCRIPT variables. Each invariant is based on a fault model, representing AJAX-specific faults that are likely to occur and which can be captured through the given invariant.

SYSTEM CONFIGURATION:-

HARDWARE REQUIREMENTS:-

- ✓ Processor -Pentium –III
- ✓ Speed - 1.1 Ghz
- ✓ RAM - 256 MB(min)
- ✓ Hard Disk - 20 GB
- ✓ Floppy Drive - 1.44 MB
- ✓ Key Board - Standard Windows Keyboard
- ✓ Mouse - Two or Three Button Mouse
- ✓ Monitor - SVGA

SOFTWARE REQUIREMENTS:-

- Operating System : Windows95/98/2000/XP
- Application Server : Tomcat5.0/6.X
- Front End : Java, JSP
- Script : JavaScript.
- Server side Script : Java Server Pages.