Interaction between web browsers and script engines

Xiaoyu Zhuang

Institutionen för informationsteknologi
Department of Information Technology
Abstract

Interaction between web browser and the script engine

Xiaoyu Zhuang

Web browser plays an important part of internet experience and JavaScript is the most popular programming language as a client side script to build an active and advance end user experience. The script engine which executes JavaScript needs to interact with web browser to get access to its DOM elements and other host objects. Browser from host side needs to initialize the script engine and dispatch script source code to the engine side.

This thesis studies the interaction between the script engine and its host browser. The shell where the engine address to make calls towards outside is called hosting layer. This report mainly discussed what operations could appear in this layer and designed testing cases to validate if the browser is robust and reliable regarding hosting operations.
## Contents

1. Introduction ......................................................................................................................... 7  
   1.1 Tasks of the thesis ......................................................................................................... 7  
   1.2JavaScript and its engine ............................................................................................... 9  
   1.3 Browser and Hosting .................................................................................................... 10  
   1.4 Hosting layer feature .................................................................................................... 11  
2. Testing Approach .................................................................................................................. 13  
   2.1 Software cycle ............................................................................................................... 13  
      2.1.1 Why testing ............................................................................................................. 13  
      2.1.2 Life cycle of software development ....................................................................... 13  
   2.2 Testing Angles ............................................................................................................... 15  
      2.2.1 Functional tests ...................................................................................................... 15  
      2.2.2 Boundary tests ........................................................................................................ 15  
      2.2.3 Performance tests .................................................................................................. 16  
      2.2.4 Security tests ......................................................................................................... 16  
      2.2.5 Stress tests ............................................................................................................. 16  
   2.3 Testing Method .............................................................................................................. 17  
      2.3.1 Black Box ............................................................................................................... 17  
      2.3.2 White Box .............................................................................................................. 17  
3. Functional Tests .................................................................................................................... 19  
   3.1 COM .............................................................................................................................. 19  
      3.1.1 Key interface – IDispatch ...................................................................................... 19  
      3.1.2 Variant marshal ....................................................................................................... 20  
   3.2 DOM .............................................................................................................................. 24  
      3.2.1 Name lookup .......................................................................................................... 24  
      3.2.2 Event Handler ......................................................................................................... 25  
      Inline .............................................................................................................................. 26  
      Function pointer ............................................................................................................. 26  
      Script block .................................................................................................................... 26  
      Naming convention ......................................................................................................... 27  
      Automagic ....................................................................................................................... 28  
   3.3 Other Process – GetObject ............................................................................................ 34
1. Introduction

With the rapid development of computer network, Internet becomes an important tool of everyday life. As one of the fundamental applications used on internet, web browser is used by millions of customers browsing World Wide Web on daily basis. This requires a browser with high quality, good performance and solid reliability. Unlike web1.0 which contains only static html pages and links, today’s web focuses on dynamic page content and interaction between pages and end users which makes websites more like an application. To improve the user interaction experience, scripts are introduced to web pages and running on client machines.

A script engine runs scripts directly on interpreting; it has access to variable components on web browser, remote host and local components. The script engine is not self-executable; it requires a host application to create the engine, to pass host objects, and receiving function calls. This thesis investigates the interaction between the script engine and its host and tests this hosting layer. In this case, the host is Internet Explorer 9, the engine is Jscript engine.

1.1 Tasks of the thesis

Because JavaScript engine is the key component for web browsers’ performance and functionality, almost all major browsers have their own version of JavaScript engine and enrich its supporting features on each release. Internet Explorer – developed by Microsoft, enhances JavaScript engine by each release as well. Internet Explorer 9 released JavaScript engine version 9.

The purpose of this thesis is to get a deeper understanding of the communications between the script engine and Internet Explorer through testing the software – what functional features around hosting are provided and how to build a solid communication layer. All the features included in hosting area need to be addressed and discussed. Although the case studied here is Internet Explorer, the same test suite could be expanded to other web browser as well.

Testing is conducted through both end-to-end validation and customized host. End-to-end validation is to verify each feature by directly running scripts inside Internet Explorer. This approach is simple and straightforward. Most of the features discussed in later sections can be covered by
end-to-end tests; especially those scripts calling into DOM objects which only valid inside Internet Explorer.

Customized host is a more direct approach. It builds up a wrapper around JavaScript engine and mimics interactions between the engine and its host. This customized host calls directly into the engine interfaces and may hit features/codes that could not found by end-to-end testing.

Customized host performs as an analyzing tool as well. Some features involve both implementation on the script engine side and the host side. If engine implemented a feature and host presents it to end user, or host passes an object to engine and engine does the real math, a failure in between would be hard to isolate. Customized host could monitor directly through API level and draw conclusion quicker than debugging.

The rest of the report includes:

Section 1.2 – 1.4 is a briefly introduction on JavaScript engine and its relationship to host

Section 2 introduces software development cycle, knowledge on how to test software in general and what basic testing angles need to shot before releasing software to customer.

Section 3 – 5 involves the actual testing work around hosting areas.

Section 3 includes a feature break down on hosting area - what is considered as hosting area feature and testing approach for each feature.

Section 4 provides the idea of mutator. It is used to expand testing coverage by leveraging existing test code. This is a powerful add-on that helps hitting different code paths using the same testing logic.

Section 5 identifies how a customized host is built up and hosts multiple script engines based on public interfaces between the host and the script engine, and how it is going to help on hosting feature tests.

Section 6 gives an overall summary on hosting tests – the good part and where improvement needed.

Section 7 draws what is left to next release of JavaScript engine, new feature included and more.
1.2 JavaScript and its engine

JavaScript is an interpreted programming language with object-oriented (OO) capabilities. It is most commonly used in web browsers, and, in that context, the general purpose core is extended with objects that allow scripts to interact with the user, control the web browser, and alter the document content that appears within the web browser window. [1] Even though there are plenty of scripts supported by different browser and platform, e.g. Perl, VBScript, JavaScript is the only one that is supported by all major web browsers which make it the number one programming language used in web developing. JavaScript code is loaded and executed in client side browser; it has a key impact on the overall performance of a web browser.

Similar to other script languages, JavaScript does not need to compile, nor does it produce an executable application. It relies on host to interpret text based source file and execute while interpreting. The JavaScript DLL component hosted under an application is a live instance of a JavaScript engine. It reads JavaScript source file and executes accordingly.

In Microsoft Windows products, JavaScript is implemented as a COM DLL element that hosted by various applications. Two major hosts here are Windows Script Host and Internet Explorer. Microsoft has different versions of JavaScript engines. Before IE9 (Internet Explorer version 9), IE uses the same JavaScript engine as WSH (Windows Script Host) -- Jscript.dll. In IE9, supported only on Vista and above, the browser loads a different DLL – Jscript9.dll, designed specifically for the browser. WSH, on the other hands, remains loading Jscript.dll – the legacy engine.

There are quite a lot improvements made into Jscript9. Jscript9, as a newer developed engine, implements all new features defined in ECMAScript version 5. It enabled objects with special properties as sealed, configurable, and enumerable. It also improves performance by using compiled code rather than simply interpreting. To make it better backward compatible, Jscript9 engine has standard doc mode as well as IE8 and IE7 doc modes where it kept the same behavior as legacy engine. This feature helps browser functioning correctly where the sites were designed for compatible view only.
1.3 Browser and Hosting

In order to communicate with a script engine, there is a pair of interfaces needs implemented by the engine and the host – IActiveScript and IActiveScriptSite. These interfaces enabled basic functionality that script engine could provide. Any application – includes Internet Explorer, that properly implemented IActiveScriptSite could hold a script engine and uses it to load, parse, execute script source file.

IActiveScriptSite is implemented on the host side that enables the engine to call its host. The engine uses this interface to trigger OnEnterScript and OnLeaveScript events, which inform the host that a script is running. During the executing of scripts, the engine may call into the host through this interface to inform that execution has completed, engine has changed state, or script error thrown but not caught inside the engine.

IActiveScript, implemented on the script engine side, provides necessary function calls to initialize the engine. During initializing, the host calls into the engine through SetScriptSite and passes itself to the engine. This interface also includes an API – AddNamedItem, where the host could add objects directly to engine's global scope. IE uses this function to add Window and other DOM objects into the engine to enable its interaction with DOM. Another important API helps host retrieving IDispatch interface from engine which is a key element in further interactions on invoking methods of the service provider.

The above interfaces support the basic handshakes between the engine and the host, but in order to accomplish more work, especially interactions with
end user, the host needs to invoke engine method responding to user input – attaching event handler. The IDispatch interface mentioned previously is designed for this functionality. IDispatch interface is a common communication interface that used in almost all COM. It provides few methods for initializing, and the most important – invoking method that could set/get property value or invoking function calls.

1.4 Hosting layer feature

The interface mentioned above initialized the first handshake between JavaScript engine and Internet Explorer. After the channel has been set up, the engine could get reference to various objects coming from Internet Explorer and the communications between the engine and foreign objects made up all features in hosting layer.

[2] Feature about hosting area

**COM**: the script engine could call into all kinds of COM objects, native code or managed code through public interfaces that COM objects exposed. Any system registered COM objects could initialized by Internet Explorer through new ActiveXObject, more important here is that COM could return VARIANT to the script engine and the engine needs to marshal it correctly before using it as a JavaScript object.

**Other process**: the engine could create and communicate with other process while running under Internet Explorer. This is a legacy feature and
only application with OLE interface supported could let the engine calling through. Office products all support this interface, so with proper settings from the IE security manager, JavaScript engine could create an Excel process and control its cell element, query and update its value.

**DOM:** This is why JavaScript was introduced to the web browsing at the first place. Hosts grant engines the ability to get and set values, update properties of DOM element, and react to user inputs. From the engine perspective, the host gives the engine two references that injected into global scope – window and document.

**5.8 engine:** As mentioned before, Internet Explorer loads Jscript9.dll as the JavaScript engine by default. All JavaScript goes to this engine. However, there is a special type of JavaScript that is not supported in Jscript9.dll – Jscript.Encode. This is encoded JavaScript that supported only by legacy engine – Jscript.dll. When host parses Jscript.Encode, it initialize legacy Jscript engine to execute encoded scripts. With two Jscript engines hosted under IE, the newer engine could still talks to the legacy engine, gets references of objects initialized from legacy engine, and calls into its method just like talking to COM objects.

**VBScript engine:** IE supports VBScript by default, although this type of script is not widely used on internet world, IE still kept the ability to support VBScript. Similar to the 5.8 engine, JavaScript engine can call into VBScript and vice versa.

**Other JavaScript engine:** Each window hosted in IE loads a different site, thus there is a separate engine created for each of them. These windows are running on different threads but they are still able to talk with each other if there is a reference passed between them by the host– through parent/child relationship. With window.open() calls, IE opens another window and returns the reference to the caller window, then engine could start talking to the engine on child page.

Similar to window.open, one web page could also host several script engines through iframes. Iframes are commonly used for ads display and each iframe uses a different engine. Unlike window.open, all these engines are running on the same thread with the main page engine.
2. Testing Approach

Before jumping into real work that has been done on the script hosting, I will give a brief introduction on concepts of software development cycles and methods used for software testing. It is not only suitable to this particular project, also works in software testing project in general.

2.1 Software cycle

2.1.1 Why testing
In the early age of software engineering, there was no professional tester. The software back then was small application and not used in everyday life. Developers simply test the functions they wrote, make sure it works in several common used scenarios and sign off to release.

Nowadays, software is facing millions of potential customers and usage is not limited to few scenarios. Making a bullet-proved software means it works correctly by providing functions it is designed for; it handles exceptions in a user-friendly way that help users identifying the problem; it is reliable under stressful conditions and not broken in any cases; its functions should use less resource – in the cost of time, memory and CPU.

To accomplish the requirements above, professional testers are introduced to software industry. The developer to tester ratio should be around 1:1 to build a good quality of software. This ratio ensures at least one developed feature could be tested by another one tester.

2.1.2 Life cycle of software development
There are few models in software engineering on developing software. One most commonly used is waterfall model. Breaking software product cycle into several parts and each part starts after the previous one is completed. Shown in the graph below, this model is the nature way for developing a software, even it has drawbacks – not agile enough when a design change is needed, you will still find this model in quite a lot software projects.
A software cycle usually contains few important milestones when planning tasks. Based on the model above, a typical software planning shows as below:

1. Design complete: in early stage, the project manager identifies the requirements of the software and designs user scenarios – who is the targeted customers and how they are going to use the software.
2. Code complete: developers work on the implementation to accomplish design requirements in previous stage.
3. Test complete: after feature implemented and became testable, testing starts and completes all scenarios identified in the test plan.
4. ZBB: zero bug bounce is the milestone when fewer bugs were found than early stage of testing, which means testers have identified all major issues and software starts to become stable.
5. Dog-food: release to internal developers in other team or partner customers. It gets more people using the software and issues found helps reveal testing coverage holes.
6. Beta: public released to customers. In this stage, software should contain all functionality, but may still have bugs need fixing.
7. Release: after collected enough feedback from public customers and fixed major issues, software is finally ready to release.
Okay, here comes the question: when to involve testers? A common view is: the earlier the better. Testers should get involved in design stage to avoid design defects, study user scenarios and question all possible scenarios and error handlings in the spec. While developers are busily working on feature implementation, testers need to prepare test cases based on user scenarios, testing tools and design on how to run the tests. In later stages, tester would verify bug fixes, collect user feedbacks, identify testing holes and expand coverage.

2.2 Testing Angles

The purpose of testing software is to ensure a good quality of software released to customer. Does software tester only need to play as end customer and that’s it? Not exactly.

Tester is a role that needs creations. Testers need to perform as an end user to validate functionality that software requires, need to perform as a beginner user to check any mistype or exception operation returns user friendly tips, need to perform as an attacker to hack software and ensure it is not broken, need to run software under extreme conditions to see if it still works fine.

2.2.1 Functional tests
Functional tests validate software feature from user scenarios. It validates both correct inputs and incorrect inputs. Functional testing requires a deep understanding on the feature, not only on how end user is going to use the feature, but also on the implementing of the feature.

User inputs could contain millions of combinations. There is no way to enumerate them all. The best way is to identify equivalent user input and choose few scenarios from each class. Each equivalent class should go through different code path in implementation. Identify equivalent class helps to narrow down testing areas.

Most of the tests discussed in the report fall into functional test category. That is why section 3 discussed detail features that belong to hosting area along with some implementation designs.

2.2.2 Boundary tests
Boundary testing is sometime considered as part of functional testing. It validates the boundaries of user input and software abilities. Spec needs to address the boundary information on major features, e.g. the largest variable
name could define in JavaScript engine. Spec sometime could not define an exact number on boundaries but will define how software handles input when it exceeds the boundaries, e.g. keep declaring objects in JavaScript engine and it will throw ‘out of memory’ exception. The capacity of the engine memory is different based on hardware but behavior remains the same.

2.2.3 Performance tests
Performance tests measure the resource cost when software is running. It usually uses mainstream hardware and keeps records on time cost, memory consume and CPU usage in a period of time. The data then used to compare with previous release version of the software or software from competitors that provide the same functionality.

Performance of JavaScript engine reflects page responsive time of Internet Explorer, so it is very important to the quality of the engine. Jscript engine 9 puts great efforts in IE9 release to ensure overall performance is compatible with other major web browsers' engines.

2.2.4 Security tests
This is when testers become hackers. Incomplete user input or garbage chars as input will not crash the software. In JavaScript engine, invalid JavaScript files cannot hurt the engine parser. On interface level, a customized host could get interfaces provided by the script engine. Exercise these interfaces with null pointer, mismatched data or special chars would be security tests of hosting. This is another reason to build a customized host.

2.2.5 Stress tests
Stress tests software usability under extreme conditions – when software is running with limited hardware resources, calling into a method for million times, running software for days without termination or terminating the software when it is in initialization. Stress tests help to identify basic hardware requirements and recommended requirements for software and ensure its quality when facing a broader audience.
### 2.3 Testing Method

The below chart shows common steps to test a software:

![Test Plan, Test Case, Manual / Automation Run](chart)


1. **Test plan:** With PM spec finalized, tester starts to draft the test plan. A test plan needs to address what scenarios that end user is going to use the feature, functions provided by this feature, equivalent class of user inputs, limits/boundaries on user input, error messages and other exception handling, security and performance may also considered in test plan if the feature has these concerns. Plan also needs to address testing method – how to conduct the testing, how to verify the result and baselines to identify a bug. The last part would be estimation on cost including developing testing infrastructure, drafting test cases and running all test cases regularly.

2. **Test case:** this is the implementation stage of testing a feature. With all method designed and discussed in the test plan, cases implemented would need to cover angles that discussed in section 2.2.

3. **Test run:** developers keep editing software in the whole cycle; to ensure functions are not changed and no regression brought to the software, test cases implemented need to be run on a regular basis.

#### 2.3.1 Black Box

Black box testing is a straight forward, end-to-end testing. It considers software as a black box – you cannot see its structure inside but only the output. With a given input, validates its corresponding output is correct based on the design spec. This testing method does not require understanding on implementation level, knowing the feature design would be enough.

Black box testing usually requires enumerating user input as it does not know the differences between those inputs. Even it is simple and commonly used, it cannot identify equivalent class of user input in an efficient way and may lose special corner cases. Not recommended in testing but would be the first step to know a feature.

#### 2.3.2 White Box

White box testing is also considered as a code-level testing. Software is considered as a white box where one input leads to certain output. It requires tester to know the actual implementation of the feature and to
target testing based on different code paths. Same code path uses the same logic and does not need repeated testing. Because white box knows the implementation of software, it is more efficient in identifying different equivalent class.

Testing work included in this thesis all tries to address white box testing in certain ways. While identifies the feature design for end-users, a discussion on implementation also get brought up.
3. Functional Tests

All communications between JavaScript engine and the host are considered as hosting area. To identify testing feature and have test cases covering each scenarios, the hosting area has been broken down into the following parts. These features built up the major traffic that between JavaScript engine and the host -- IE.

3.1 COM

COM is used in many aspects of JavaScript engine. It built up the foundation of how JavaScript engine talks to its host. Any traffic through the engine to the host is going through the COM layer. When JavaScript engine wants to pass an object out to other components, objects are marshaled into a COM objects.

With the help of registered ActiveX Object, JavaScript engine can talk to native/managed code through COM interfaces. Any reference passed in or out script engine needs to be properly marshaled to comply with COM standard.

Even COM is the foundation of the hosting layer; it is hard to test directly through IE because object coming from host is usually well-defined without enough variety. E.g. host sending itself to JavaScript engine through IActiveScript interface always passes in an IActiveScriptSite object; the engine executes a script and returns output to the host. The results are always transferred to strings.

ActiveX object comes as a testing approach. It is a good way to test COM related area. It provides a direct way to verify incoming and outgoing traffic between COM and script engine. Inside ActiveX object, any type of VARIANT could pass in as return value or parameters to JavaScript callback functions.

3.1.1 Key interface – IDispatch

There are lots of interfaces implemented in a COM object and JavaScript engine. We are not going to address all of them in this report. The reason I brought up IDispatch, is that it is used by almost all features we are going to discuss later because it is the foundation of further features. When engine wants to retrieve value or execute function from host side, IDispatch becomes the interface to call. On the other hand, host uses IDispatch to dispatch tasks to engine as well. Both sides need to implement IDispatch
interface. To get better understanding on how other feature works, it will be useful to take a closer look at IDispatch.

What is IDispatch? It is an interface that provides another way for components communication. Instead of providing several custom interfaces specific to the service it offers, a COM offer those services through single standard interface – IDispatch [2].

IDispatch contains 4 standard functions for a component to implement:

- HRESULT TypeInfoCount
- HRESULT TypeInfo
- HRESULT GetIDsOfNames
- HRESULT Invoke

When client request a service (usually by calling a function), it calls into GetIDsOfName implemented by service provider side—this function reads the name of the function client wants to call and returns its dispatch ID. This ID then used in the second step – Invoke which simply invokes the function with specified dispatch ID and possible arguments in VARIANT format. The output parameters of invoke include return value from invoked function, exception info if service provider thrown any unhandled exception.

All types of objects coming from the host have implemented IDispatch interface, giving JavaScript engine the ability to retrieve its value, update its status or invoke COM functions. These operations are discussed later in those feature area.

3.1.2 Variant marshal

Any parameter passing between JavaScript engine and COM needs to go through objects marshaling. This is used by both get and set values operations. Objects marshaling are implemented in both directions -- JavaScript objects to VARIANT and VARIANT to JavaScript objects.

The concept of VARIANT comes from COM. VARIANT is a data type in certain programming languages, particularly Visual Basic and C++ when using the Component Object Model. [3] COM component that written in other programming languages also marshaled their objects into VARIANT before passing outside COM. VARIANT is the interface data type that recognized by both parties in COM communication.

As marshaling is needed for both directions and this is the only gate for host and engine to talk. Tests need to run on both directions that enumerate all possible VARIANT types.
**Engine -> COM**
Calling into native code, engine will be able to pass JavaScript objects as parameters to use in native code. These JavaScript objects need to be marshaled into VARIANT before sending to native code. JavaScript is an object-oriented language; every JavaScript element is an object, including functions and properties.

Most common types of JavaScript objects can be successfully passed to COM. e.g. string, number literal, number object, boolean, array. Special objects are also properly handled, e.g. infinity, zero positive, zero negative, null, undefined, NaN. Function objects can be passed to COM as dispatch callback pointer. These areas built up basic scenarios that functional tests need to cover.

Boundary tests are also important during marshaling. Number object in script engine does not have a specific type associates with it. During marshaling, the engine needs to determine which type of VARIANT is suitable for current number object. Number object’s value beyond one VARIANT type is marshaled into another type with a higher boundary.

Objects coming from host, e.g. window, document objects are not JavaScript objects and cannot pass to COM. These scenarios are tested through negative testing.

**COM -> Engine**
When passing a foreign object from COM to JavaScript engine, the object is always represented as one type of VARIANT. To use this object in JavaScript, the engine will marshal VARIANT object into JavaScript object. The COM object could enter the script engine under two ways – as function parameters or return value. Not all COM types are supported in script engine. Engine tried to marshal all objects and map them to corresponding script objects. Most common COM types can be successfully marshaled as script objects, and the script engine in IE9 has improved its ability to receive few more types compare to last version.

The following image shows a complete work flow where the script engine calls into COM side and gets callback. It represents both COM to engine and engine to COM traffic.

The workflow also shows how tests are designed. JavaScript first creates an ActiveXObject – the COM object. The host will evaluate this request, find its class ID through system registry and create the object for the script engine. The host may prohibit object creation if security restriction applies. After getting the COM object returned, JavaScript then calling into COM native function with JavaScript object as parameters. JavaScript gives COM a function as callback; COM could also call the script engine through callback function and pass VARIANT as parameters.

There is no magic on VARIANT->JavaScript marshaling tests. The marshaling process is done inside engine, so all VARIANT types that supported in JavaScript need to go through testing process to ensure functionality. VARAINT represented number value data needs tests on boundaries as well as zero.

Unsupported VARIANT
COM could pass various VARIANT to JavaScript engine, but not all objects can be marshaled into JavaScript objects. As a legacy feature, when a VARIANT type cannot get marshaled and used as a JavaScript object, JavaScript engine will not return fail to COM. Instead, JavaScript engine will marshal it as a default object that does not have any particular object type associates with it. It is just a VARIANT object with JavaScript object shell. Engine gives exception on trying to use this object under all conditions.
However, as a VARIANT from COM, engine can pass this object back to COM. In this way, even the script engine cannot use the object; it can still be the bridges between different COM objects.

Testing of this feature follows three steps: VARIANT passed in; script tries to use it and passes back to COM.

1. For all VARIANT types that not supported by the script engine, COM can pass it to engine without exception. JavaScript engine accepts the object and marshals it correctly.

Most of the VARIANT types not supported in JavaScript are VT_ARRAY with defined types, e.g. VT_ARRAY|VT_I2. Thinking about corner cases, the special ones include passing empty array, array with all elements points to NULL, number array with elements near the boundaries.

Not only the content can be NULL or empty, the type itself can be useful too. A VT_ARRAY|VT_EMPTY is still a valid type in COM. The script engine actually fails early than other types as this variant type cannot contain any content.

2. When try to use the object marshaled from COM, reasonable exception should throw.

There are tons of ways to use an object in JavaScript but they are all similar operations – using an object is to evaluate its value under different conditions. The way to find out as many as evaluation operations is going through ECMA Spec 5, which is the definition spec of JavaScript language. From the spec, operations include math operations, evaluation inside ‘if’ statements, passing into object creator and many other environments. Each unsupported VARIANT type needs to be tested under these operations.

Assigning object to another variant in JavaScript engine does not throw any exception as this operation does not evaluate the object’s value but retrieve its reference. Operations like this are still allowed in the script engine.

3. The object can pass back to COM and the value contains in the object remains unchanged.

Since no operation has been made to JavaScript object, the VARIANT object remains the same value as it was passed in. The round-trip on object marshaling and un-marshaling will match each other. This part can be verified in COM side.
3.2 DOM

The Document Object Model (DOM) is a cross-platform and language-independent convention for representing and interacting with objects in HTML, XHTML and XML documents. [4] JavaScript is the number one programming language used in web page developing mainly due to its ability to control DOM. It has direct access to DOM elements which can dynamically change the DOM object properties, add new element to a web page, retrieve data from server and create customized page based on user profile.

Interaction with DOM is one of most important parts in hosting feature. All DOM object provides the same interface that JavaScript can call to change its properties. And each DOM object type has special feature and different component for the script engine to call. Document object provides name lookup functions for JavaScript engine to search for objects in DOM; window object provides the global scope for JavaScript engine executing code; form object has its own root for property binding; all types of objects support standard events for JavaScript engine to attach event handlers.

Among all these services provided by DOM, there is no need to test all of them. The helper function built in DOM objects are implemented by DOM side, which means it is host function that not part of testing work on the script engine. The responsibility of the engine involves mainly on locating the DOM element and attaches event handlers.

3.2.1 Name lookup

How can JavaScript get the access to DOM? It calls through name loop function. The document object published several functions where developers can get the reference to DOM object by its id, name and types. E.g.

```javascript
document.getElementById("input-button");
```

These APIs are exposed by the host as part of document object; it is the standard way to get reference to DOM. All browsers support these functions. As described above, these functions are bound to document object which implemented by DOM.

A special feature for IE JavaScript engine: it supports getting reference to DOM by using element id name directly. JavaScript engine in IE put extra efforts in locating a reference of DOM and saves developer time in finding out the DOM element. JavaScript engine first searches inside its own scope for the variable reference. If the reference is not found in engine scope, it then ask host to search in DOM or other engines’ scope.
The above code will locate DOM object named logger and uses directly in script engine. This name lookup feature enabled script engine to:

1. Reference to DOM element id can directly get DOM element.
2. Objects under another scope, e.g. embedded in form object, iframe object, are referenced by its parent.child name. e.g. form1.button1
3. With the reference, the script engine can get/set its property value, calling into its functions.
4. If there are two elements with the same name, this reference binds to the last one that loaded in DOM.
5. If there is an existing JavaScript object bind to this name, the reference returns the JavaScript object.

The feature descriptions above are check points for writing test cases.

3.2.2 Event Handler

Events are the beating heart of any JavaScript application. It enables the JavaScript script host react to users’ activities. After registered the event sink (handler function that runs when event fires) to the event sources (object fires the event), the script host can call the appropriate script function to handle events fired by the object.

Web pages use event driven model – DOM fires events during user inputs, mouse clicking or page loading. Each DOM element has a special set of events associate with it for supporting various operations. These events are supported on all browsers and enough for web developers to make well defined web pages providing interaction between users and web pages.

To attach program with events, developers use JavaScript event handlers – after DOM fires the particular event, event handler takes over the running thread and executes corresponding code. JavaScript engine needs to talk to host application to attach this handler to set up the callback.

IE supports five different syntaxes to attaching event handler, including inline, function pointer, script block, naming convention and automagic. Some of them are html standards and supported by other major browser, e.g.

```html
<p id="logger"></p>
<script>
function log(str){
    logger.innerHTML += str;
}
</script>
```
inline and function pointer. Naming convention and automagic are IE-only syntax.

Test cases need to cover all syntax to ensure its functionality. The first 4 syntaxes are commonly used and straightforward on testing. The last one requires more effort in designing cases.

**Inline**

Inline event handler is most commonly used on web pages and easy for code reading. Inline syntax put event handler code directly into HTML tag, these event handlers will run under JavaScript by default. Small html example shows below.

```html
<button id="inline" onclick='alert("I'm clicked")'>Click me</button>
```

Multiple inline event handlers could be defined in same DOM tag. What happens behind the scenes is that IE calls into the script engine with the script code and tells the engine to create an anonymous function (a function with no name under global scope). [5] The drawbacks of inline event handler are code reuse and code management.

**Function pointer**

The good part of inline event handler is easy reading. Developers can easily know each DOM property and its event handlers by reading through html files. However, this will make the code not organized as scripts scatters all over the page.

As mentioned in inline event handler part, the way that JavaScript engine implements this operation is through adding the script code as DOM function property. So another popular way to attach event handler is adding function pointer directly in JavaScript block as a sample shows below.
This operation is supported by all web browsers and it is good for code organizing as it separates static DOM definition code with JavaScript dynamic scripting code.

**Script block**
When defining a script tag, it is also possible to assign it to specific DOM event. This way the DOM event handler is assigned on parsing the script block—when DOM tries to load this script tag. A small example shows below.

```html
<script language=JScript>
button1.onclick= function()
{
    Alert("I’m clicked")
}
</script>
```

An interesting scenario that attaching script block event handler is to use a JavaScript file from different domain.

```html
<script language=JScript for=button1 event="onclick">
    alert("script block handler - JScript")
</script>
```

Script that linked in different domain inside event handler is treated as x-domain code, and cannot get its source code by accessing `button1.onclick.toString()`.

**Naming convention**
There is an IE specific naming convention way to hook up event handler to DOM event. As long as the function name is defined as ObjectId.EventName, the function is attached to that DOM event.

**HTML**

```html
<script language=JScript>
function button1::onclick()
{
    alert("clicked")
}
</script>
```

**Automagic**

In addition to inline and function pointer registration in IE, Jscript supports a built-in method to register event handler through `ObjectId::EventName`. It provides an alternative for IE to hook up objects to their event handler.

**HTML**

```html
<script language=JScript>
function button1::onclick()
{
    alert("clicked")
}
</script>
```

It is quite useful in responding to ActiveX control events since ActiveX control objects do not have standard event handler, and cannot attach event handler through inline or function pointer. In fact, automagic is the most efficient way to attach ActiveX control event handler. This feature needs tests on both DOM objects as well as ActiveX controls. To better understand what/how to test, more explanation of feature shown below.

**Loading and executing:**

Usually the script engine parses and run the script from top to bottom, left to right. So if the event handler is defined below any script code, it is executed after the script code too. E.g. the code below,
The code above dynamically creates a button object and defined its `onclick` event handler and attaches this DOM object to document.

Automagic runs in a different way: its event handler hook up happens at the very beginning of the script engine initialization – before any script ever runs. It is hosted to the top of the scope, the event binding happens on entering the current scope. When the engine got initialized by host, it creates global scope for script code to run. It also creates separate scopes in entering each function. So in order to make automagic syntax works for DOM loading, it needs wrapped inside a function and calling this function after DOM elements finished loading.

Script execution and DOM loading happens on the same thread when loading a page. DOM cannot guarantee JavaScript engine that every element on DOM has been initialized until document fires onload event. The above code put `BindEvent` under onload event handler of document. This will ensure that object – button1 which JavaScript engine needs binding event to, is initialized completely by DOM at the time of binding handler.

**How the binding works:**
When script engine parses this code, it first tries to load the host object which handler attaches to (because both DOM and ActiveX objects are created by host and passed to JavaScript engine, the objects in event handler binding are all host objects). JavaScript engine then query host object for `IBindEventHandler` interface.
If host object implemented IBindEventHandler interface, it will be easy for JavaScript engine to bind the handler. This interface defines how host object is going to trigger the event handler function when an event is fired. JavaScript engine only needs to invoke the BindHandler method on IBindEventHandler interface with event handler function entry point and event name, the rest of attaching code is implemented inside host object.

If the host doesn’t implement the interface IBindEventHandler, the information of event handler (object name, event type, handler function entry point) is saved in an internal structure and the binding is done via event sinking process when the script engine gets in connected mode.

Test cases simply mix-match the following object/event and that covers all scenarios from different equivalent classes. Unlike other tests, event handler is an async call. Test case needs to register event handler to objects, then calling into DOM/ActiveX control to fire corresponding events.

Object and event
After understanding how the event binding works, we can define equivalent class on objects as below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Details</th>
<th>Support IBindEventHandler</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM object</td>
<td>Level1/2/3</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Iframe</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>window</td>
<td>No</td>
</tr>
<tr>
<td>ActiveX Object</td>
<td>Non-IBindEventHandler</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>IBindEventHandler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

DOM
There are not many differences between different levels of DOM objects from JavaScript engine perspective; engine always uses the same event sink process. From host side, objects with different level, especially objects that define scope always support more operations. Tests need to cover those special objects and it will work fine for the rest of DOM types.

- Window object defines global scope of current JavaScript engine;
- Form object defines its own root—another scope in JavaScript engine;
- Document object is the root object for all DOM elements;
- Iframe object defines another JavaScript engine but running on the same thread with current engine.
- Child window object running on a separate thread but can still refer by parent window.
**ActiveX Object**

ActiveX Object is another type of objects that this syntax binds event handler to. Only ActiveX Object can implement IBindEventHandler interface. Both objects with/without IBindEventHandler can successfully attach JavaScript event handler. Therefore, two types of ActiveX control object need to be tried here.

**Events**

The events on DOM objects target two sets: the most commonly used event, object specific supported events.

Commonly used events are those events that most popular in websites e.g. onclick event for most of DOM objects to interact with user click, onsubmit event for form object to submit request, onmouseover event for showing tips, onkeypress event to show inline typing suggestions. Object specific supported events includes more events that only one or two objects supported, e.g. onscroll event for window object, onerror event for window object as the final layer to handle script exceptions, onload event for document object to fire scripts when all DOM element are loaded.

ActiveX Object can be considered simple here because all ActiveX Object events are defined inside ActiveX Object. When the event fired, it calls into JavaScript through IDispatch interface. From engine’s perspective, there is no difference from one event to another, and they are considered as in same equivalent class. However, AcitveX Object is special for that their event handler can accept parameters while regular DOM event handler does not. For DOM objects, even passing parameters in firing event, the parameter is not going to be sent to handler. But, ActiveX Object does supports parameter passing from ActiveX Object to event handler in JavaScript code. The parameter passed into JavaScript engine code is marshaled from VARIANT to JavaScript object. Zero parameter, one and multiple parameters constructed testing for ActiveX controls.

**Build ActiveX Control**

I cannot use existing ActiveX control because I need to monitor and verify data passed to ActiveX control at runtime. Besides it is hard to find an ActiveX control with IBindEventHandler. Most ActiveX control will let engine take care of event sink. Therefore, a test ActiveX Object binary is needed here.

To build an ActiveX Object that supports JavaScript event handler contains 3 parts – register, unregister, fire events. The register and unregister used for binary registration on system. FireEvents function will triggers events inside ActiveX control; it is also exposed to JavaScript engine as a COM visible
function so JavaScript can fire event directly. Other than these parts, ActiveX control needs to provide delegate pointer for JavaScript engine to attach event handler. So once an event fires inside ActiveX Object, it will callback into JavaScript code using delegate pointer.

The following code defines three events that accept different number of parameters. As discussed above, three types of events are included here – with zero/single/multiple parameters.

```
C#
public delegate void ControlEventHandler0();
public delegate void ControlEventHandler1(string str1);
public delegate void ControlEventHandler2(string str1, string str2);
```

Now the ActiveX control understands the delegate definition, only needs to declare them inside object class. These events need accessed from outside, they must declared as public.

```
C#
public event ControlEventHandler0 OnCustomEvent0;
public event ControlEventHandler1 OnCustomEvent1;
public event ControlEventHandler2 OnCustomEvent2;
```

To make these events visible to JavaScript engine, I added a public interface and corresponding dispId for each event, so the engine could get these information through query interface in IDispatch handshakes. After these steps, JavaScript engine will know this ActiveX Object supports three events. After attaching JavaScript event handlers, when any of these events fires, it directly triggers the event handler in JavaScript code.

```
C#
public interface ControlEvents
{
    // the COM DispId.
    [DispId(0x00000001)]
    void OnCustomEvent0(); //This method will be visible from JS
    [DispId(0x00000002)]
    void OnCustomEvent1(string str1);
    [DispId(0x00000003)]
    void OnCustomEvent2(string str1,string str2);
}
```
Usually there is no need to implement `IBindEventHandler`, script engine could still bind event. I implemented `IBindEventHandler` in one of the ActiveX control, just to hit different code path in script engine on binding events.

```csharp
[Guid("63CDBCB0-C1B1-11D0-9336-00A0C90DCAA9")]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
[ComVisible(true)]
public interface IBindEventHandler
{
    void BindHandler(string eventName, IDispatch disp);
}
```

`BindHandler` gets an event name and `IDispatch` pointer from the script engine. `IDispatch` pointer indicates the entry point on JavaScript event handler function. Here the event pointer is assigned to event delegate. When event fires in ActiveX Object, it will reach JavaScript handler code through `disp` ID.

```csharp
[ComVisible(true)]
public void BindHandler(string eventName, IDispatch disp)
{
    switch(eventName)
    {
    case "OnCustomEvent0":
        OnCustomEvent0 = disp;
        break;
    case "OnCustomEvent1":
        OnCustomEvent1 = disp;
        break;
    case "OnCustomEvent2":
        OnCustomEvent2 = disp;
        break;
    default:
        throw new Exception("Non - Support Event");
    }
}
```
Usually this code is more complex as user could add code checks and error handling on the information provided by the script engine. E.g. check if event handler in the script engine side is valid or pre-exist.

With ActiveX Object ready, event handler can attached in JavaScript code:

```javascript
function ActiveXControl::OnCustomEvent0()
{
    /*Event fires in ActiveX Object, catch in JavaScript handler */
}
```

### 3.3 Other Process – GetObject

JavaScript was originally design as a client side script for browser; later on, as it is widely used, it also considered as a powerful script that supports system level operations. It has the ability to create an independent process with interface that supports its calls. GetObject is that interface where JavaScript engine communicates with other process.

GetObject allows JavaScript engine to create another process and return an object reference. The script engine can specify the file path, which opens the file using the default registered program for that file type. Ideally, any process implemented GetObject interface can be created through JavaScript.

JavaScript engine can specify only the progid which will create an instance of the specified program. If the program is already instantiated, a reference to the existing instance will be returned. JavaScript engine can also specify both parameters which will create an instance of the specified program with the specified file opened.

```javascript
var file = GetObject("c:\sampleFile.txt");
```

Object
Office family products are used for testing this feature. Not only they all implemented OLE interface which is needed for GetObject; they also expand
Testing around office family products will cover all end-to-end functional scenarios.

### Feature Interactions

When GetObject is called, JavaScript engine queries IE for permission to invoke ActiveX components. IE checks its security settings and either allows the component to be loaded (if security settings are off), prompts to user for decision (if set to do so for the particular component type, i.e. signed or unsigned, marked safe for scripting), or rejects the request if setting does not permit ActiveX components to be loaded. This process is controlled by IE security manager; it has different settings for each zone based on file resource location.

### Security

Security tests brought up here as this feature has security concerns. As mentioned above, GetObject is controlled by IE security manager due to its security restriction. GetObject could be a path to disk when websites want to open a file directly from browsers. It could also became a nightmare that browser could easily open/edit/close a file, running a component on client side without end users’ knowledge. It is important to cover scenarios under different security settings here.

### 3.4 Legacy Engine

Jscript9.dll is the default engine that IE9 hosted for any JavaScript code, either inside a html page, or linked in a script block as the source file. Web developer cannot specify IE9 to load the legacy engine – Jscript.dll. Although Jscript9.dll engine supports IE7 and IE8 doc modes for backward compatibility, it is just a different mode still running under Jscript9.dll.

Legacy JavaScript engine has a feature that Jscript9.dll has deprecated – Jscript.Encode. For each script block in html page, there is a language property that developer set to specify which script uses here. If this property is not set, browser loads JavaScript engine by default. Other than JavaScript and VBScript in this property, IE supports another type – Jscript.Encode.

Jscript.Encode is encoded JavaScript. Script. It is created by Windows Script Encoder (screnc.exe) -- a Microsoft tool that can be used to encode scripts (i.e. JScript, ASP pages, VBScript). It only encodes scripts, not encrypts.

The following code shows an example on encoded JavaScript. screnc.exe encode all scripts under script block, not includes comments though.
Since Jscript.Encode is not supported on Jscript9 engine, when IE9 encounters a script block with type marked as Jscript.Encode, IE9 loads Jscript engine to decode and execute scripts. IE hosts both Jscript9 and Jscript engine at runtime, and both engines can talk to the other one.

Any communication between Jscript9 and Jscript engine needs go through IE. Even these two engines use the same syntax; they are still treated as two different engines. Operations send to the other engine are through IDispatch interface and follow basic COM principles. Object marshaling and IDispatch interface has been discussed pretty through in COM section, so I'm not going to add any new content here. All basic IDispatch operations include get/set/invoke is supported between Jscript9 and legacy engine.
**Get:** Jscript9 engine could get the values of objects created in legacy engine, and vice versa.

**Set:** similar to Jscript9 engine, set value to objects holds in the other engine is also allowed.

**Invoke:** invoke a function in another engine. Could pass parameter and get return values.

Functional cases go through the above scenarios to modify objects created in Jscript.dll engine. One case is special in legacy engine interaction: If an object constructor is defined in Jscript.Encode and trying to initialize this object in Jscript9 engine, is it a valid operation? This operation is valid in IE8 as IE8 hosts both Jscript and Jscript Encode in the same engine. In IE9, because they are hosted under different engines, extra effort is made to maintain backward compatibility.

### 3.5 VBScript Engine

The default script engine in IE is JavaScript, IE delivers scripts directly to JavaScript engine if no language property set to script block. IE supports another script engine – VBScript. If the language is set to VBScript in property, IE initialize VBScript engine to execute scripts.

Similar to Jscript.Encode, JavaScript can also reference to object in VBScript engine. When calling an object, the JavaScript engine starts property lookup within its own object, then search for window object, after that goes to other script engines.

![Diagram](Image)

[5] JavaScript engine query a VB object from VBScript engine
The functionality that supports object referencing between JavaScript engine and VB script engine follows the same pattern as in Jscript.Encode; they all go through host for any operation. The only difference is that VBScript is not an object oriented language, there is no object definition in VBScript and we cannot initialize a VBScript object in JavaScript.

During investigating VB array data format and implementation, a special data type brought my attention -- **VBArray**. It needs more testing because its structure and syntax.

**First**, the array object in Jscript is not the same structure as in VB or C++. Array in JavaScript is not a chunk of memory where array elements are located side by side. JavaScript array is more like a linked list where element are all referenced by array object as its properties. VBArray is a type of SafeArray, similar to C++ arrays, it is a chuck of memory and its element are referenced by address offset.

[6] Array structure comparison between JavaScript and VBScript

There is an API in Jscript to convert VB array into Jscript array.

```javascript
var x = new VBArray(VBarr)
```

VBArray object defined in JavaScript engine smoothly converts VB array reference to JavaScript array, especially for multiple dimensions array and sparse array. VBScript supports multiple dimensions array, after its converting to JavaScript array; it is merged into one dimension. With large but sparse VBArray case, JavaScript array does not use extra memory to store VBArray element with no value.
**Second**, VBArray use parentheses to index its element and JavaScript syntax defines parentheses as the way to call functions. When we call VBScript array object in JavaScript, JavaScript engine determines this object referenced an object in VBScript engine; then the rest part is passed to VBScript for parsing and evaluation.

The following code may look strange in JavaScript code as it appears to be assigning a value to function calls. It is valid in JavaScript as long as reference VBArr pointed to a VB array object in VBScript engine.

```javascript
VBScript
VBArr(0) = 0;
```

### 3.6 Other Jscript engine

The host can initialize several JavaScript engines, under the same thread or different threads, based on different DOM object type. The current engine can reference to other JavaScript engines through host, the same way as Jscript.Encode and VBScript.

Unlike Jscript.Encode and VBScript, referencing other JavaScript engine is also supported on other browsers and is widely used in World Wide Web. An Iframe (inline frame) loads on HTML page will initialize a new JavaScript engine and running on the same thread as current DOM and main page JavaScript engine. Calling window.open(url) will loads another JavaScript engine in separate thread.

Iframe is commonly used to display a subpage hold inside a parent page. One example would be ads where its source pages came from another domain and could be dynamic – different users could see different contents. Window.open is an even more popular scenario; any links on HTML page could open another new page in the tab. The new window and its opener can reference each other through host.

Web developers will not feel the differences when calling objects from other contexts. They expect using object from another context the same as regular context. Cross context is a large feature in hosting; each feature would need special handling regards cross context calls. The following breakdown is a discussion about how the cross context call is accomplished and the special places of cross context.
3.6.1 Foundation of Cross Context
When IE parses an Iframe or window.open request, both Iframe and window own separate JavaScript engine and these engines are hosted by IE under the same process. Current context – parent frame, always refer to cross context – child frame through host. After host provides the reference to child frame, how it handles parent calls and how parent window responds to child requests are totally based on JavaScript engine implementations.

All cross context calls are accomplished by our universal powerful interface – IDispatch. When parent window wants to create a child frame, it sends the request to host and host initialize child context with parent window IDispatch pointer. After hosts initialized child frame and build up parent-child relationship, it returns the child frame dispatch pointer to parent window, so it can refer to child context. To make sure the functionality works the same as single context, object marshaled from different context should work the same as object created on current context.

When parent window needs to set/get/invoke functions or objects on child context, child window needs to marshal the object and return to parent window. For simple value variant, e.g. primitive number or string, only the value itself is passed back, no reference needed here. Unlike objects created on current context, the current script engine does not have the knowledge of objects’ status from the other context. The object could be deleted on the other context, or the whole child frame engine could be deleted as child frame may get closed or navigated away to other sites. Cross context maintains solid lifetime logic so it will not access to illegal memory.

3.6.2 Create Iframe and window at runtime
To test this cross context feature, I need to know how to create cross context objects dynamically. This JavaScript code needed here is to create iframe and new window at runtime, execute JavaScript on it and return the result back.

Creation
Iframe represents inline frame, it usually embedded inside html page with Iframe tag.

**HTML**

```
<iframe src="source.html" id ="adFrame"></iframe>
```
The source property of Iframe can be easily changed during runtime which allows showing different content while loading the page. After getting Iframe’s window object, window.location.assign can update Iframe source to a new location. The operation is trivial but there might be difficulties during assigning new source to Iframe – if parent page has reference to objects from iframe, they are not accessible after iframe navigates to another page. Detail discussed in section 3.6.4.

```javascript
adFrame.contentWindow.location.assign("newLocation.html");
```

Other than static embedded into html page, Iframe also supports added during runtime – same as other major DOM elements. Not only can JavaScript create a new Iframe at runtime, it can also write to Iframe’s document object and create the whole page completely without having a static page on server. This is useful when parent window trying to show different contents for different users. It is also used here to build up cross context tool – no static file needed for cross context, it could be done at runtime.

```javascript
adFrame = document.createElement('iframe');
document.body.appendChild(adFrame);
writeChildContent(adFrame.contentDocument);
```

Notice that the above code creates an iframe at runtime, appends it to current page so it became an element on page - child frame of current parent frame. Then some code is written to this iframe. After this step, can parent window start to call JavaScript objects and DOM elements? The answer is NO.

**Wait for ready**

One big issue with runtime iframe and even for static iframe, is asynchronous calls. Remember that JavaScript engine runs on the same thread with UI and DOM, and the same thread with iframe too. At any time, only one component could possibly get the execution handler.
In the above example on iframe, parent window JavaScript engine creates an iframe and append that to parent window. Then after execution finished, handler returns to host, DOM starts to create the iframe object along with its engine as requested by parent window JavaScript engine. After iframe and its engine were initialized, parent window ask child frame to load the content as parent engine written. Parent engine can access child objects only after all contents has been loaded, including DOM loaded and scripts executed.

Due to asynchronous nature of iframe code, the following graph shows the correct and safe way to access a child window’s engine. Cross context code that I wrote follows this workflow to avoid any mistakes in asynchronous calls. Even for static iframe, the best way to access child engine is still access it after it fires document.onload event – when the child content is fully loaded.

![Diagram showing correct and safe way to access a child window's engine.]

[7] Safe way to call child iframe engine

### 3.6.3 Object marshaling

Parent window refers to child window object through child window engine reference requires object marshaling as well, because an object from another frame is coming from another engine. Since parent window does not have the knowledge of child context, child window need to marshal object information as well as frame information.

When parent window get the reference to a child window object and wants to make changes or get its value, it queries IDispatch interface and uses it for any information updating. Child window access this through the same way – IDispatch.

Although the IDispatch code has been discussed in previous section, it is important to have tests on each object types because marshaling process is
different from type to type. Functional test needs to cover all type of objects created on child context and access by parent operations.

### 3.6.4 Object lifetime on freed pages

When parent frame gets reference to its child frame, it also gets access to child's object and functions. When parent window access a child's object, it creates a local copy of that object with all properties. Because function execution requires closure and context information, function objects are left on child frame as the engine cannot copy child script context to parent frame.

If child frame is closed or navigated away, the script engine hold on child context also get released. However, parent window can still access to the child's objects accessed before as they are copied locally. When trying to execute any child functions, the engine will throw an exception saying that it cannot execute freed scripts. How browser handles this exceptional scenario is a good way to value its reliability.

**Call structure**

The following workflow illustrates this scenario well.

- Parent frame first creates the child frame, injects code on the child frame and waits till its DOM loaded ready.
- After the child frame loaded successful, parent frame calls a child function or other operation that binds a parent reference to a child function. The function on child window could be an object property, deferred function object, inside an array or even object accessory.
- In the next step, child frame navigates to a different page which will automatically close the previous page engine.
- After navigation completed, parent window will not be able to access the previous function and a JavaScript exception is thrown warning user on this operation, when user trying to use the reference from parent window.
3.7 Query Continue

Query Continue is a legacy feature from IE8; it is a method on `IActiveScriptSiteInterruptPoll` interface. This method polls the script engines on UI thread and allow the host to know whether the script is running. The host can stop the script engine from running scripts. Both legacy JavaScript engine and VBScript engine have implemented this feature. This feature basically enables host to query current state of the script engine and the ability to stop the engine running without reloading it.

Host implement `IActiveScriptSiteInterruptPoll` interface. This interface has only one function -- `QueryContinue`. JavaScript engine needs to call this function while running and ask host if it should continue running script. After host gets the request, it returns the engine `S_FALSE` to notify it stop running.

Sometimes, an engine may be waiting for other calls to return and may not execute any scripts. Host calls into `IActiveScriptStats` to get the statements of engine running. With increased number of statements counts, host knows
the engine is still running and after a while will inform end user that the engine has been running too long which hangs the page, also provides with a way to stop the engine from running. After user made decision on stopping the script engine, host will return S_FALSE to tell the engine to stop.

![Diagram](image)

[9] Query Continue interface

Query Continue is the only way for host to stop the engine running normally. Without Query Continue, hosts including IE9, are unable to tell script engine running state and to stop its running. The only other way to terminate a hang page is to refresh the page, therefore loses all previously inputted information on the page. Query Continue is a ‘nice to have’ feature: no websites are supposed to run into a situation where its script keep running and hangs the page. But if the browser can handle this exception state better, it will be good to maintain its reliability.

3.7.1 Notify users

IE9 implemented Hang Recovery mechanism that triggered when user trying to interact with a hang page. This is a separate feature with query continue. Hang recovery windows shows up as a non-blocking bar at the bottom of IE when page loses responding over 8 seconds and user click on the page or try to interact with the page in other ways.

If the reason causes page stop responding is not the script engine, browser has an option for user to ‘recover webpage’ which basically refresh the current tab and load the page. With Query Continue, if host knows it is the script engine which hangs the page, it notifies user with option ‘stop script’,
this will stop the script engine but changes made by previous code still remains. The engine can still react to new script execution request.

Not all hosts has Hang Recovery implemented. Host without hang recovery will shows a legacy UI. A dialog pops up asking user's decision when limits been hit. This legacy dialog is a blocking alert. The engine stops when popping this dialog for user decision and resume/stop after alert dismissed.

3.7.2 Functional Testing
Engine needs to call Query Continue function periodically to give host a way to stop engine from running. The functional tests are to validate under different long running scripts, host can stop engine while maintain the page correctness. Tests are mainly done by end-to-end tests: running the scripts in IE and causing it hang on purpose. Then check the hang resistant bar to see if IE can stop engine.

Few scenarios discussed below. First scenario is Query Continue triggered inside eval calls. Second scenario is to make a call to built-in native function in loops. Furthermore, browser needs to handle when a nested call happens between engine and host. The last scenario is a nested frame case – the long running script is not on the main page but inside an iframe.

Loop with eval
Eval is a built-in function that available under any JavaScript scope. It does not need to be attached to any objects. It takes a string as an argument and evaluates its value under current scope. The input string of eval call could be one line expression or multiple lines statements. Eval function enables to execute dynamic JavaScript code – code that generated at runtime.
Because eval uses different approach as other JavaScript function calls – the expression it evaluated is created dynamically at runtime, it does not execute the same way as normal JavaScript code. Due to its flexibility in generating code, it is also widely used in websites designs. So it is important to check the host is able to stop engine while it is executed inside eval code.

Stop script executing while keep calling eval inside a loop.

```
JavaScript
while(true)
{
  eval('var x = []');
}
```

Stop script executing while executing inside eval.

```
JavaScript
x = new Object();
eval("for( var i = 0; i< 1500000000; i++)
{
  x.a = {};
}");
```

Loop calls built-in function

Similar to the scenario above, there are other built-in functions. Unlike JavaScript functions, the implementation of these function are inside engine.

For example, the following code forces a garbage collection in JavaScript code. CollectGarbage is a native function implemented by JavaScript engine to enable a garbage collection in case of large object released. The engine need to be able to safely stop when calling into native functions.

```
JavaScript
while(true)
{
  CollectGarbage();
}
```
DOM interaction

The script engine always works together with DOM so DOM interactions build up a significant part of JavaScript usage. Here is to test the engine could be stopped correctly when interacting with DOM elements.

Nested scenario with JavaScript calls into DOM and then callback. This could be done through setTimeout or DOM event handler.

```javascript
window.setTimeout(function(){
    while(true){
        var x = new Array();
    }
}
```

Creating DOM elements and attaching to DOM is another operation that commonly used by script engine. The following code creates a button object and attach/detach it from DOM tree.

```javascript
for(var x = 0; x<1000000; x++)
{
    button1 = document.createElement('button');
    document.body.appendChild(button1);
    document.body.removeChild(button1);
}
```

Long running scripts in iframes

If the long running JavaScript code is from iframe instead of main page engine, host is still able to stop the engine running.

3.7.4 Live site

The above scenarios are just a guess on user patterns. To evaluate how this feature works in real life, I also did a top 300 sites validation. With each site, loads its main page and forces its script loop running. Most of the sites will run into hang state. Only 21 sites have exception handling where it returns to DOM periodically to avoid hang page. With the rest of the websites that
does hang under this environment, all of them calls Query Continue and browser is able to stop the script from running.

This may not be true in real hang site through, because real hangs may happen on simpler or smaller scripts. But this proves that Query Continue allows host to stop script engine while running into hang state without causing any damage to the browser.
4. Mutation Test

During developing cycle of software, after all features been implemented and properly tested, code coverage of existing test cases remains static. Test cases verify all logic defined by feature and go through scenarios carefully. It will be hard to explore coverage by writing more cases. There comes the concept of mutation test – a technique to expand testing coverage by leveraging existing test code.

4.1 What is Mutation Tests

There are features provide the same functionality but using different way to call, e.g. as I described in event handler part, there are 5 different syntaxes to attach event handlers. Even these features has the same functionality, the implementation behind it inside the script engine could be totally different. If there is one case on inline event handler and we know the pattern to mutate it to other event handler format, this one case can be mutated into other four versions and tested against different code path.

Mutation test is a software testing technique that modifies testing source code in a small way. [6] Equivalent mutation is more commonly seen as it mutates code following the same logic. Testing code passed before should pass after the mutation.

The tool used to mutate existing code is called mutator. It is especially powerful on source code testing where simply mutation would hit different implementation from parser to executor.

4.2 Example of mutator

Studying the programming language can help identify syntaxes with the same functionality. It is not hard to find good examples of mutation. Example below shows few demos of equivalent mutations.

Switch If Logic

‘If’ statement is used in all programming languages providing basic logic judgments. Switch its content of if-else is an example of ‘if’ logic mutation.
Original code:                                           Mutated code:

```java
if( isTrue ){
   //Operation A
}
else{
   //Operation B
}
```

```java
if( ! isTrue ){
   //Operation B
}
else{
   //Operation A
}
```

**Name vs. Index property:**

Some programing languages support both name and index calls when accessing an object property. A mutation uses index property method instead of name property method is another type of equivalent mutation.

Original code:

```java
object.foo = 0;
verify(object.foo == 0);
```

Mutated code:

```java
object['foo'] = 0;
verify(object['foo'] == 0);
```

Minor change to code will hit different paths in implementation.

**More complex mutation:**

Small changes to syntax and logic are one type of mutation. There are other ways to make complicated mutations by using a small group of code replacing existing simple expression.

Original code:                                           Mutated code:

```java
x = new Object();
```

```java
try{
   throw new Object();
}
catch(e){
   x = e;
}
```

The example above mutated a simple new object assignment to a try-catch environment. Similarly, any code could use some kind of mutation: using while loop instead of for loop; wrapping an object or value as return value from a function call; moving function object to object property and call through objects.
4.3 How it mutates

After got the idea of what is mutation test and demos of mutation, the next step is to know how it mutates the code and therefore build other mutators to help finding bugs.

The first step of a mutator is to parse scripts. Script source code can be parsed into a list structure where each node could be further parsed as a tree structure. Here is an example shows a small part of JavaScript.

```
JavaScript:
var x = new Array();
x.a = 0;
```

The above code can be parsed as the list below. The operations of two statements are linked in a list. Each operation is further split into arguments. If the argument itself is an operation call, the node is then split into another node with parameters.

[10] Syntax tree

After parsed script code into syntax tree, mutator is just to replace part of tree nodes into other content. E.g. the name-index mutator changes the operation “Get Property” into “Index Property”.

![Syntax tree diagram]
5. Build a JavaScript engine host

Since we discussed all major script engine interfaces in section 3, is it possible to create my own JavaScript engine host and get results by running scripts? This may not be easy for the first time, but creating a customized host would definitely help getting a deeper understanding of interfaces, COM mechanism and provides a tool to explore more on security and reliability aspects where we can directly inject and exercise on API level.

I created an experimental script host under .NET platform to host script engine. This host follows the same interfaces all discussed in section 3. It provides similar functionality to Windows Script Host, runs under command line. Since all script engine which follows ECMA script spec implemented the same set of interfaces, this experimental script host can host not only JavaScript 9 engine, but also legacy JavaScript engine, VBScript engine. So the functionality of this simple host can expanded to compare running results from different engines, IDispatch-based interaction calls between different engines.

The requirements of this script host are:

- A .NET application that implements major hosting interfaces discussed in section 3 includes: IActiveScriptSite, IActiveScriptParser
- Hosts multiple script engines including Jscript9.dll
- Execute same code and compare results between legacy engine and Jscript9 engine.

5.1 Customized script host

All features discussed in section 3 are integrated with IE. Why we need a customized script host? Is there any existing tool that can have the same functionality of the script host?

The first reason would be testing purposes. Not all features can be validated end-to-end. There are features that only exist between script engine and its host. If we uses existing host to analyze results, it is always on script level. We can only test functionalities that IE has provided. IE may has done checks on its own before sending objects to script engine. When script engine talks ONLY to IE, it makes assumptions on objects coming in and trusts on these objects. This would leave engine unreliable when switching to other hosts. These bugs would not pop to surface if we always use IE. For reliability and
security purpose, a customized script host would be important and provides targeted and special attacks.

Another reason would be isolating issues. If a feature requires supports from both engine and host, how can we easily tell if a bug comes from host side or engine side? Feature like query continue, if no ‘stop script’ option showing for a long running scripts, does the issue due to engine not calling query continue or IE failed to show the UI? If we want to know when does query continue get called and how many calls we got, a customized host could easily print these information out.

Scripts are dynamic languages and have easy syntax for developers. Embedding scripts under a .NET application is sometimes needed and Microsoft has Script Object to complete this functionality. A Script Object is a COM object that can receive script source code and execute under specified engine type. It is an old component that release along with Windows but does not support Jscript9.dll which runs the new version of script engine. The only way to host a Jscript9.dll script engine is under IE9 or a customized host.

5.2 Fist Step – get interface reference

The first step is to give the definition of host/script interfaces so host would know how to call script engines. These interface information are included in activscp.idl and ships along with .NET developer SDK. The Interface Definition Language (IDL) is a language for specifying operations (procedures or functions), parameters to these operations, and data types [7]. C++ uses for compiling header files. But this file cannot be used under C# and Microsoft does not have a C# version of interface definition – either by interface code or dll file.

I can convert and wrote a C# version based on the definition from MSDN or uses activscp.idl to get its .NET definitions. The second option sounds better so I leveraged few tools in Visual Studio – midl.exe and tlbimp.exe.

Midl.exe converts interface definition files to type library files (.tlb). A type library (.tlb) is a binary file that stores information about a COM or DCOM object’s properties and methods in a form that is accessible to other applications at runtime. [8] With type library file, I will be able to include interfaces into a COM later in step two which supported in all types of languages. Meanwhile, the activscp.idl file is not used to convert to type.
library file. Before throw it to Midl.exe, a library scope is required to wrap outside of interface definitions.

With interface definitions in type library file, the second step is to convert library to COM. It is done fairly easy through tlbimp.exe which imports tlb file and produces a dll file. Adding this dll file as a reference to .NET project and now the project knows interfaces definitions of script engines.

The interfaces defined in this dll contain all we need to build up the first handshake between script engine and host:

**Host**: IActiveScriptSite, IActiveScriptSiteInterruptionPoll, IActiveScriptSiteWindow

**Engine**: IActiveScript; IActiveScriptEncode, IActiveScriptError, IActiveScriptParse, , IActiveScriptParseProcedure, IActiveScriptParseProcedure2, IActiveScriptParseProcedureOld, IActiveScriptProperty, IActiveScriptStats, IBindEventHandler,

These interfaces contain almost all we need for hosting a script engine and can provides basic functionalities requested for this simple host.

### 5.3 .NET – As simple as few lines

The reason I chose C# as developing language is not only it is an easy writing language and well organized, it also marshaled every data type to COM types automatically. There is no more need to be done except using System.Runtime.InteropServices. The System.Runtime.InteropServices namespace provides a wide variety of members that support COM interop and platform invoke services [9]. With this namespace, there is no need for declaring VARIANT type; expose a public interface of COM or loading a COM component is just an attribute.

My code builds up the host side; it implements interfaces a script host would need to have. The script engine is a COM component to load. Here I import the registered script engine component by referencing its GUID. Wrap this imported COM with a class name and host will knows which engine to reference:
GUID is used to specify which engine to load and System.Runtime.InteropServices will finish the rest of work on locating the DLL and loading it on runtime – enjoy the benefit of .NET.

5.4 Scripts running

To make the script host start running, host side needs to implement IActiveScriptSite and pass itself to the script engine through IActiveScript interface. After this first handshake, host can calls into engine’s function through different interfaces engine supports.

IActiveScriptSite has few functions that engine will call on events happenning, e.g. OnEnterScript defines the operations where script engine get handlers, this function get called on every entering or re-entering script engine; OnLeaveScript similar to OnEnterScript, it get called before script engine give handler back to host; OnScriptError will tell host that an unhandled JavaScript exception thrown and needs host to bubble to upper layers; OnScriptTerminate informs the host that script has completed execution and is going to be torn down.

For now, none of these functions needs real implementation, simply print to console would help us to know which function get called.

After finished IActiveScriptSite on host side, I created a function as the entry point of the application. In this function, host initializes a shell class which represents a script engine; calling into engine’s IActiveScript interface and pass host object to engine through IActiveScript::SetScriptSite.
After setting host on script engine, host can start to run a sample script by using engine's `IActiveScriptParse::ParseScriptText`. This interface is used to parse script text, executing scripts and returns its result.

```csharp
public void Init() {
    Jscript9 engine = new Jscript9();
    IActiveScript ias = engine as IActiveScript;
    ias.SetScriptSite(this);
}
```

The above code executes a simple expression and receives the results back. This function will get changed later to execute more complex scenarios. Both legacy engine and Jscript9 engine can be easily hold under this host. From the result we can see engine calls enter and leave script before and after execution. Leave scripts calls before it returns handler back to host.

```csharp
IActiveScriptParse iap = engine as IActiveScriptParse;
iap.InitNew();
object output = null;
tagEXCEPINFO exe;
iap.ParseScriptText("0+1", null, null, null, 0, 0x00000002 | 0x00000020,
out output, out exe);
Console.WriteLine(engineName + " : " + output.ToString());
```

```
Executing under Jscript9
=========
Enter Script
Leave Script
Jscript9 : 1
```

```
Executing under Legacy
=========
Enter Script
Leave Script
Legacy: 1
```
5.5 Inject host functions to engine

JavaScript does not have an output function to print strings to console. Host is always in charge of providing a logging mechanism for the script engine. If JavaScript is running under Windows Script Host, “WScript.Echo” is the host function for alerts something as output. In web browsers, “alert” provides the same functionality.

In previous demo code, the output is returned to host and print to console instead. If I want to use this host to execute a JavaScript file and print the output at runtime, a C# callback function is required to be injected into the script engines.

Injecting a host object to the engine is provided in IActiveScript::AddNamedItem. Host provides an object name and flags to the engine. When the engine runs into the object with this name and there is no definition in the engine, the engine will return to host for further execution. Most of host objects are injected into script engine through this way.

I want to first give my engine the ability to output to console.

```
C#

IActiveScript ias = this.engine9 as IActiveScript;
    ias.SetScriptSite(this);
    ias.AddNamedItem("HostObject",
SCRIPTITEM_ISVISIBLE|SCRIPTITEM_GLOBALMEMBERS);
```

The two flags are used to indicate that the HostObject injected into script engine is under global scope where any script code can access to it, and all its content – properties, child object, events – are visible to script code for accessing.

The above code only injected the object name to global scope, how can the engine tie it up with functions in C#? It is through IActiveScriptSite::
    GetItemInfo – another function implemented by host.
When engine needs to make a call on host object, it goes through GetItemInfo to get the reference to host object. Within this function, engine provides the host with object name, host need to fill two out parameters – Object reference marshaled as IUnknown and TypeInterface on the object.

```c#
public void GetItemInfo(string pstrName, uint dwReturnMask, [Out, MarshalAs(UnmanagedType.IUnknown)] out object ppiunkItem, IntPtr ppti) {
    switch (pstrName) {
    case "HostObject":
        ppiunkItem = new HostObject();
        Marshal.WriteIntPtr(ppti, Marshal.GetITypeInfoForType(ppiunkItem.GetType()));
        break;
    default:
        ppiunkItem = null;
        ppti = System.IntPtr.Zero;
    }
}
```

Here HostObject is a .NET object, System.Runtime.InteropServices will take care of marshaling managed object to IUnknown object which the engine can call into it. Using this approach, host can inject any code into the script engine global scope -- .NET object or COM, providing more flexibility on host.

### 5.6 Running a script file

Before continue with the rest of the functions, this host needs to be able to run a script file instead of an expression. This is simple done by reading a file path as parameter and pass its content to the engine on ParseScriptText.

If any script file needs output a log, simply calling into the Host function implemented in section 3.5 will give handler back to host functions.
5.7 QueryContinue interfaces

Query Continue feature, discussed in section 3.7, is one of the reason I need to build a customized host. Because Jscript9 re-implemented query continue feature and to make sure it works, I want to know whether a query continue is called, how often this function get called, the number of stat from `IActiveScriptStats::GetStatEx` where host uses to determine if script engine is still running.

Implementing `IActiveScriptSiteInterruptPoll` interface and get QueryContinue function to output a log file would solve the problem. With this function, host can measure how often a query continue call made by engine, given specific script code.

End-to-End validation is done as well on Query Continue feature. But using customized host is much easier to locate where the problem is. If a query continue call has been made through the script, then engine did its job, anything happens after this step will be host’s responsibility. If no query continue called through the time, then probably something goes wrong inside engine -- query continue was not fire on time.
6. Summary

So far IE9 has been released and so does all features related to hosting. Hosting area is one of the complicated areas of an engine. Anything happens inside JavaScript engine, the engine would always have a full knowledge around that and it well-maintained and under control. Objects coming from host into JavaScript engine are another story. When JavaScript engine passes a reference out, it needs to know that requester is a trust user that will not mess up with JavaScript engine resource. When the engine received an object from outside – usually COM object, it needs to make sure if anything goes wrong inside this component, it is not going to affects functionality of the engine itself or break anything.

JavaScript is most commonly used for web programming and it always live with DOM and other hosting objects. Through the testing work done towards the browser, I approved that IE9 makes great efforts in building a robust hosting layer while keeping it in high performance.

All objects passing into JavaScript engine come from host through COM interface. But testing COM interface and object marshaling are not enough. The real world scenarios and objects are more complicated than simply IDispatch object. To ensure functionality works for each type of host object, hosting layer has been further break down into sub features.

During this project, all these sub features and objects discussed in section 3 were thoroughly tested, including COM objects coming to JavaScript engine as an AcitveX Object; DOM object that JavaScript engine can locate its reference, update its property and attach event handlers; Other process reference created by GetObject command; Legacy engine objects through Jscript.Encode; VBScript object that initialized and hosted under same browser; Other JavaScript engine runs on the same or different thread with current engine that build up a parent-child relationship; Query Continue function call that enables host to stop script engine if it runs for too long time.

Other than functional cases written for each feature, a customized host is created as a tool to exercise hosting interface from API level; to isolate issues when a bug comes up; to analyze and compare script running results between different engines.

The testing result shows that IE9 has built up a reliable hosting layer and supports a wide selection of hosting features allowing web developers to use JavaScript as a powerful tool for active web pages.
7. Further Work

After releasing IE9, what happens to the next release? Is IE10 just another browser with some new features that IE9 does not supported?

There are new features added to IE10. Microsoft has released four developer preview version on IE10. It contains TypedArray which allows JavaScript creating array buffer in C++ style with specific types. It enhances the performance when the script engine need to access a large amount of data.

IE10 also contains new HTML5 standard feature – web worker. As JavaScript relies on direct access to DOM, it is required to run on the same thread with UI. This means anything happened inside the script engine needs to be fast and then return the handler back to UI. UI can then response to UI input to keep the page responsive. If a script needs to run for a longer time, it has to yield to UI and switch back using setTimeout. Web worker allows developer to put pure JavaScript, non-UI related code on a different thread and communicate with UI thread through messaging model. This feature allows any long running script to run without stop and yield to UI. Because worker thread is also hosted under IE and communicates with UI thread through message queue, it is also considered as a feature for hosting.

If JavaScript engine continues to be hosted only under IE10, this would make is a simply advanced version of IE9 – that is not true. If you ever tried Windows 8 Developer Preview version, you will know a new concept called Windows Web Application (WWA). This is a great new concept that Microsoft brought to costumers and developers. WWA allows developer to use JavaScript to build a Windows application. JavaScript calls can be projected to native C++ component through WinRT objects, so writing a new Windows application would be much easier than before and native code allows it running efficiently.

More features and more work for hosting. JavaScript grows from a simple client side script which enables basic interaction between user input and web page response, into a dynamic yet powerful and extendable script language with better support in web browser as well as application developing environment.
8. Reference


