Target Audience: Researchers and developers who want to modify and extend Artemis. Looking for usage documentation? Try to run `artemis --help` and read the Artemis paper.

Artemis
The algorithm below shows the major steps involved in the Artemis testing procedure. The finer details and capabilities are omitted from the algorithm itself, we refer to the detailed description of each step later in this document.

Note, this section is incomplete as we are evolving the testing procedure to accommodate new features.

```python
def main(URL_initial, forms_initial):
    # Step 0: Global initialization
    worklist = Worklist()
    browser = Browser()
    statistics = Statistics()
    input_strategy = InputStrategy()

    for configuration in worklist:
        # Step 1: Page load
        browser.reset_counters()
        browser.load_page(configuration.url)

        # Step 2: Post-load processing
        browser.fill_forms(forms_initial)

        # Step 3: Input sequence execution
        for input in configuration.inputs:
            browser.fill_and_mark_forms(input.forms)
            browser.trigger_event(input.event)

        # Step 4: Post-input processing
        statistics.update(browser.counters)

        # Step 5: Iteration
        configurations_new = input_strategy.generate(configuration, browser.counters, statistics)
        worklist.add(configurations_new)
```
1.1 Step 0: Global initialization

1.2 Step 1: Page load

1.3 Step 2: Post-load processing
   - Form inputs provided on the command line are written to the page.

1.4 Step 3: Input sequence execution
   - Non-blank form inputs (represented by FormInput) are written to the page.
   - Form inputs (represented by FormInput) are marked as dynamic form inputs such that the symbolic infrastructure can associate the form inputs back to Artemis.

1.5 Step 4: Postprocessing

1.6 Step 5: Iteration

Substantial Addons:
Artemis supports the usage of AIL descriptions when testing web applications. An AIL description is a specification of the client-server communication conducted using Ajax. These descriptions allow Artemis to test the client-side without a concrete instance of the server-side.

This functionality is described in more detail in *Server Interface Descriptions for Automated Testing of JavaScript Web Applications*, Casper S. Jensen, Anders Møller and Zhendong Su. ESEC/FSE 2013.

### 2.1 Using AIL Descriptions

AIL descriptions is used by Artemis through the `ailproxy.js` HTTP proxy. This proxy accepts an AIL description file and acts as a mock server listening on localhost port 8080.

To use this proxy Artemis must be run using the `-t` argument.

### 2.2 Generating AIL Descriptions

AIL descriptions can be tedious to write by hand. This can be mitigated by using the AIL learning algorithm provided in the `contrib` folder. The AIL learning algorithm uses a dump (raw) of concrete client-server traffic to generate an AIL description.

The `raw` traffic can be collected by using the `HTTPdump proxy`.

The learning algorithm is provided as a python script in the `contrib/ajaxinterface` folder.
This section documents various modifications made to WebKit as part of our instrumentation.

Note, we require WebKit and specifically the JavaScript Core interpreter to be compiled in a non-JIT mode and in 64bit-mode (read: we do not support JIT compiling and the 32bit compatible version of WebKit).

### 3.1 Tracking Values Symbolically

WebKit has been extended with *symbolic values and semantics* mirroring the concrete values and semantics respectively. Symbolic values are injected at predetermined sources (currently only the value property on input element DOM objects).

As an example, accessing the value property on DOM node D returns a concrete string, denoted C, marked as a symbolic value, denoted S, originating from D. Any concrete operation operation on C will be matched with a symbolic operation on S. Thus, if the length property is accessed on C it will return a concrete value C2 representing the concrete length of the string, and C2 will be marked with a symbolic value S2 representing the symbolic length of the symbolic string S.

We say that there exist a number of mutators in WebKit taking a number of inputs I_0 \ldots I_n and outputting an output value O. Artemis instruments WebKit such that in all mutators, the output value O is marked with a proper symbolic value taking into account the concrete semantics of the mutator and the concrete and symbolic values of the inputs.

The following diagram gives an overview of the different concrete values, mutators and symbolic values and how they relate in the implementation.
Note, there exist three levels of concrete values in WebKit: JSValues, object interfaces and internal objects. The JSValues are used to represent both primitive values and pointers to objects in JavaScript. JSValue is the primary type being passed around in the JavaScript interpreter. If JSValue points to an object, then it has a pointer to an object interface. This interface acts as a proxy to internal objects, usually conducting type conversion while delegating business logic and storage of values to the internal objects. The object interfaces are automatically generated in order to allow different JavaScript interpreter implementations to interface with the same internal objects.

We have identified two primary mutators, the JavaScript interpreter and native functions. In general, the JavaScript interpreter only manipulates the primitive values stored in JSValue, while the native functions operate on everything from JSValue to the internal objects.

- Symbolic values are attached to all primitive values stored in JSValue and the interpreter has been instrumented to maintain the symbolic values for all operations on JSValues.

- A subset of native functions operating on JSValues have been instrumented.

- A subset of interface objects track symbolic values for their concrete properties (JSString and the value property on input elements).

- A subset of native functions operating on interface/internal objects have been instrumented (JSString and input elements).

A note on strings: Symbolic strings are marked symbolic both in the JSValue pointing the JSString object and in the JSString object itself. The JSValue will make sure to propagate its symbolic value to the JSString. The JSString is immutable so it will never change its own symbolic value, thus keeping the two consistent. The symbolic value needs to be represented in the JSString since the native functions operating on JSString never gets a reference to the enclosing JSValue (and these functions derive new symbolic values based on the current string, e.g. string length).

### 3.1.1 Symbolically Enhanced JavaScript Values

JavaScript values in WebKit are NaN encoded

NaN encoding is used in WebKit to represent JavaScript values, denoted JSValue. Each JSValue is a 64bit value, either representing a `double`, `integer`, or a `pointer` to a JSCell. A detailed description of this encoding can be found in JSValue.h, here and here.
We want to enhance each concrete JavaScript value with a pointer to a symbolic representation of the same value. Furthermore, we want this to be fairly efficient, and only decrease efficiency if a symbolic value is present.

If a concrete values is not associated with a symbolic value we use the current NaN encoding for performance.

We can’t represent both a concrete value and a symbolic value within 64bit at the same time. In this case, we tag the value to be symbolic and store a pointer to an object in turn pointing to the concrete and the symbolic value.

Specifically, we change the following patterns:

| Pointer | 0000:PPPP:PPPP:PPPP 
| / 0001:****:****:**** |
| Double | ... |
| Integer | FFFF:0000:IIII:IIII |

into:

| Pointer | 0000:0PPP:PPPP:PPPP 
| / 0001:****:****:**** |
| Double | ... |
| Symbolic: |
| Object | FFFF:3PPP:PPPP:PPPP |
| True | FFFF:7PPP:PPPP:PPPP |
| False | FFFF:5PPP:PPPP:PPPP |
| Null | FFFF:1PPP:PPPP:PPPP |
| Double | FFFF:9PPP:PPPP:PPPP |
| Integer | FFFF:DPPP:PPPP:PPPP |
| Integer | FFFF:C000:IIII:IIII |

Notice that 64bit pointers only take up 44bit, leaving the top 20bit unused.

We extend the value objects representation of 32 bit integers. Here, 4bit of the previously unused area for integers are used to tag a specific symbolic type, and the remaining bits are used for storing a pointer to a combined concrete and symbolic value.

Notice, that the bit patterns for the different symbolic values are as follows:

<table>
<thead>
<tr>
<th>a b c d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
</tr>
<tr>
<td>Null</td>
</tr>
<tr>
<td>True</td>
</tr>
<tr>
<td>False</td>
</tr>
<tr>
<td>S. Int</td>
</tr>
<tr>
<td>S. Double</td>
</tr>
</tbody>
</table>
| Int | 1 1 0 0 | (not symbolic)

The (a) bit indicates if the value is numeric or not, the (d) bit indicates if the value is symbolic or not (in order to differentiate normal concrete integers).

### 3.1.2 Special Casing Symbolic Strings

Strings are represented by a JSValue (object type) who points to a JSString. A string is made symbolic by marking both the JSValue and JSString as symbolic. It is not enough to only mark the JSValue as symbolic, because a number of internal library functions (which we need to instrument for correct symbolic handling) only operate on the JSString
object, and can’t access the JSValue pointing to it. We fix this by propagating the symbolic information from the JSValue to the JSString.

This can cause problems if two distinct JSValue objects point to the same JSString for optimization purposes. Some special handling exist to avoid this case.

### 3.1.3 Special Casing Symbolic Objects

We do not support symbolic objects in general. However, we do mark specific objects as symbolic in order to implement special handling of specific instances of objects.

- We make the result returned by regexp operations (who return arrays or null) symbolic. The symbolic value from these operations is treated as a special null or non-null symbolic value, in order to reason about the outcome of a regexp match.

### 3.1.4 Special Casing Indirect Symbolic Values

- We mark objects as indirect symbolic if they are accessed using a value lookup using a symbolic index. This is used as a flag in order to implement symbolic value properties on option elements within a select element soundly. See issue #82, access pattern 3.

### 3.2 Symbolic Handling of Native JavaScript Functions and DOM

As an easy reference, we use [http://www.w3schools.com/jsref/](http://www.w3schools.com/jsref/) as an easy-to-read reference of the API of native JavaScript-, browser-, and DOM objects. We want to support all parts of the API which can read, modify, or create symbolic values - either by emitting constraints or emitting a warning indicating incomplete symbolic handling.

#### 3.2.1 Symbolic Support

String.{charAt, concat, match, replace, search, toString, valueOf, length, substr, substring, toLocaleLowerCase, toLocaleUpperCase, toLowerCase, toUpperCase},

**Note:** String.replace(S2, S3) only supported if String is symbolic. Warnings are emitted if S1 are not symbolic but S2 or S3 are symbolic.

**Note:** String.{substr, substring} emit warnings if their indexes (start index, end index or length) are symbolic.

RegExp.{exec, test},

**Note:** RegExp.exec only support non-global regular expressions. If the regular expression contains the global flag, then only the first match using exec is supported. Warnings are emitted for subsequent matches.
**Note:** `RegExp.exec, test` and `String.match, replace, search` using regular expressions only support the positive case (in which a match exist). The constraints emitted are not always satisfiable if the solution expects the negative case (in which no match exist).

`parseInt`, `Math.floor, ceil, max, min`.

**Note:** JavaScript represents all numeric values as doubles, while we represent numeric values as integers. Thus, `Math.floor, ceil` returns the input symbolic value unmodified. This introduces some degree of imprecision in our solutions.

`Array.indexOf`

**Note:** Only support for symbolic objects.

`Input.value, valueAsNumber, valueAsDate, checked`, `Select.value, selectedIndex`, `OptionGroup.selectedIndex, Option.value`

**Note:** See the next section for a detailed explanation of symbolic form inputs, and the exact support for different properties for each input type.

**Note:** All other properties on the `Input[(Checkbox, Radio, Text)], Select, OptionGroup and Option` objects are not supported and do not emit warnings.

`Event.target`

**Note:** `Event.target` acts as a symbolic source of symbolic objects.

`Element.tagName, getAttribute`, `HTMLElement.id, className, title, lang`

### 3.2.2 Usage Warnings

`Math.abs, acos, asin, atan, atan2, cos, exp, log, pow, random, round, sin, sqrt, tan`.

`String.charCodeAt, indexIf, lastIndexOf, localeCompare, slice, split, substring, trim, trimLeft, trimRight, anchor, big, blink, bold, fixed, fontcolor, fontsize, italics, link, small, strike, sub, sup, fromCharCode`.

`RegExp.constructor, compile), decodeURI, decodeURIComponent, encodeURI, encodeURIComponent, eval, isFinite, isNaN, parseFloat, escape, unescape`

**Note:** `RegExp.constructor, compile)(A1, A2)` emit warnings if A1 or A2 are symbolic. Thus, we only support concrete regular expressions.

`Element.*, HTMLElement.*, Node.*`

3.2. Symbolic Handling of Native JavaScript Functions and DOM
3.2.3 No Symbolic Support and No Usage Warnings

Array.*, Boolean.*, Date.*, Number.*, RegExp.{global, ignoreCase, lastIndex, multiline, source, toString}, String.<index lookup>

Window.*, Navigator.*, Screen.*, History.*, Location.*
document.*, Attribute.*, Events.*, Event.*


Input Checkbox.*, Input Radio.*, Input Text.*, Select.*, OptionGroup.*, Option.*

3.3 Form Input Support

Artemis supports concolic testing over values injected into DOM Input, Select, and Option elements.

By default, the above elements are concrete until a symbolic trigger is activated. Specifically, any of the above elements act as a symbolic source when their .symbolictrigger property is read from within the JavaScript runtime.

The following table lists the symbolic behavior of the .value, .valueAsNumber, .valueAsDate, .selectedOption, .stepDown, and .stepUp properties on the DOM Input element as a function of the DOM input element’s type. Furthermore, it also lists all relevant attributes on the DOM nodes affecting the valid symbolic input.

The current implementation does not take any attributes into consideration.
### 3.3. Form Input Support

A number of input types add additional constraints on the formatting of valid values.

1. Only one input[type=radio] element within a group may be checked.

<table>
<thead>
<tr>
<th>type</th>
<th>value</th>
<th>valueAsNumber</th>
<th>valueAsDate</th>
<th>checked</th>
<th>selectedOption</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>symbolic</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>WARNING</td>
<td>n/a maxlength, pattern, readonly, required</td>
</tr>
<tr>
<td>search</td>
<td>WARNING</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>WARNING</td>
<td>n/a maxlength, pattern, readonly, required</td>
</tr>
<tr>
<td>tel</td>
<td>WARNING</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>WARNING</td>
<td>n/a maxlength, pattern, readonly, required</td>
</tr>
<tr>
<td>url</td>
<td>WARNING</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>WARNING</td>
<td>n/a maxlength, readonly, required</td>
</tr>
<tr>
<td>email</td>
<td>WARNING</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a maxlength, readonly, required</td>
</tr>
<tr>
<td>password</td>
<td>symbolic</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a maxlength, pattern, readonly, required</td>
</tr>
<tr>
<td>date-time</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a max, min, readonly, required, step</td>
</tr>
<tr>
<td>date</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a max, min, readonly, required, step</td>
</tr>
<tr>
<td>month</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a max, min, readonly, required, step</td>
</tr>
<tr>
<td>week</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a max, min, readonly, required, step</td>
</tr>
<tr>
<td>time</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a max, min, readonly, required, step</td>
</tr>
<tr>
<td>datetime-local</td>
<td>WARNING</td>
<td>n/a</td>
<td>n/a</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a max, min, readonly, required, step</td>
</tr>
<tr>
<td>number</td>
<td>WARNING</td>
<td>n/a</td>
<td>n/a</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a max, min, readonly, required, step</td>
</tr>
<tr>
<td>range</td>
<td>WARNING</td>
<td>n/a</td>
<td>n/a</td>
<td>WARNING</td>
<td>WARNING</td>
<td>n/a max, min, step</td>
</tr>
<tr>
<td>color</td>
<td>WARNING</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>checkbox</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>symbolic</td>
<td>n/a</td>
<td>n/a required</td>
</tr>
<tr>
<td>radio</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>symbolic</td>
<td>n/a</td>
<td>n/a required</td>
</tr>
<tr>
<td>file</td>
<td>WARNING</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a required, accept</td>
</tr>
<tr>
<td>submit</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>image</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>reset</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>button</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>hidden</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
This is a new major-mode for Artemis which runs the concolic analysis on standalone JavaScript snippets. (The normal concolic mode is specifically designed for form validation analysis and relies on this setting.)

### 4.1 Overview

A single JavaScript file is loaded by Artemis and executed. The context is a blank web page (because of the architecture of Artemis).

There are three new built-in functions which are used to get inputs for the concolic testing:

<table>
<thead>
<tr>
<th>Function</th>
<th>Concolic input</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>artemisInputString(&quot;x&quot;)</code></td>
<td>Returns a string corresponding to concolic variable $x$.</td>
</tr>
<tr>
<td><code>artemisInputInteger(&quot;y&quot;)</code></td>
<td>Returns an integer corresponding to concolic variable $y$.</td>
</tr>
<tr>
<td><code>artemisInputBoolean(&quot;z&quot;)</code></td>
<td>Returns a boolean corresponding to concolic variable $z$.</td>
</tr>
</tbody>
</table>

These are called by the input JavaScript code to get the inputs for the code being tested. Any branches depending on these input values should be explored by the concolic analysis, by substituting new values during a subsequent iteration.

### 4.2 Example

Here is `simple-conditions.js`:

```javascript
var x = artemisInputString('x');
var y = artemisInputInteger('y');
var z = artemisInputBoolean('z');

if (x == "testme") {
    alert("String "+ x + ": is OK");
```

```javascript
} else {
  alert("String '" + x + "' is not valid.");
}

if (y > 10) {
  alert("Int '" + y + "' is OK");
} else {
  alert("Int '" + y + "' is not valid.");
}

if (z) {
  alert("Bool '" + z + "' is OK");
} else {
  alert("Bool '" + z + "' is not valid.");
}

Artemis is invoked as follows:

```artemis --major-mode concolic-test -i 0 --concolic-test-mode-js artemis-code/tests/ →system/fixtures/concolic-engine/simple-conditions.js -v all```

It explores the code in 8 iterations, and produces the following tree:

![Concolic infrastructure test mode tree diagram]

4.3 Tests

There is a test suite for concolic-test mode, at artemis-code/tests/system/concolic_engine.py.

So far it only tests the mode itself - the symbolic and concolic features are already covered by other test suites.
With \texttt{--major-mode server} Artemis runs an ‘analysis server’ with a JSON interface for controlling it externally and reporting what it finds.

In this mode all other arguments except those prefixed by \texttt{analysis-server-} are ignored, including the URL.

There is a debug view which shows the internal browser, which is shown when the option \texttt{--analysis-server-debug-view} is given.

The concolic advice mode is documented at \textit{Server Mode - Concolic Advice}.

\section{The API}

The server runs on port 8008 by default. This can be changed with the \texttt{--analysis-server-port} option.

Calls to the server are expected to POST a JSON message with the following format:

\begin{verbatim}
{
    "command": "pageload",
    "url": "http://www.example.com"
}
\end{verbatim}

We do not use REST-style URLs to avoid the complications of URL-encoding complex strings like URLs or XPaths.

The \texttt{command} property must always be set, and the rest of the properties depend on which command was used.

There is an echo command which can be used to check the server is running:

\begin{verbatim}
curl -w "\n" --data '{"command":"echo","message":"Hello, World"}' localhost:8008
\end{verbatim}

This should return:

\begin{verbatim}
{
    "message": "Hello, World"
}
\end{verbatim}
Only one command can be sent per request. The server is designed to be blocking; any requests sent while another is still being processed will return an error.

## 5.2 Commands

- **echo** Used for testing. Returns the text provided in the `message` field. The optional `delay` field is the number of seconds to delay before sending back the response (integers 0–30 are valid).

  Send:

  ```json
  {  
    "command": "echo",  
    "message": "Hello, World",  
    "delay": 1  
  }
  ```

  Receive: `{"message": "Hello, World"}`

- **exit** Shuts down the server.

  Send: `{"command": "exit"}`

  Receive: `{"message": "Server is shutting down"}`

- **pageload** Loads a URL in the Artemis browser. The final URL we end up on after redirects etc. is returned. The optional `timeout` parameter is the number of milliseconds to wait before cancelling the load and returning an error (integers 0–3600000 accepted), 0 implies no timeout.

  Send:

  ```json
  {  
    "command": "pageload",  
    "url": "http://www.example.com",  
    "timeout": 5000  
  }
  ```

  Receive:

  ```json
  {  
    "pageload": "done",  
    "url": "http://www.example.com"  
  }
  ```

- **backbutton** Uses the browser history to go back one page.

  It is an error to call this command before there are at least two pages in the history. Due to an implementation issue, “about:blank” is never accessible via this command.

  Send: `{"command": "backbutton"}`

  Receive:

  ```json
  {  
    "backbutton": "done",  
    "url": "http://www.example.com"  
  }
  ```
• handlers Lists the event handlers registered on the current page. The list returned shows XPath expressions identifying the DOM elements with events, and a list of events attached to each. The special cases "document" or "window" may also be given as an identifier for events registered on those objects.

There must already have been a page load command issued.

Send: { "command": "handlers" }

Receive: (e.g for the handlers.html test case)

```json
{
  "handlers": [
    {
      "element": "//a[@id='dom-attr']",
      "events": ["click"]
    },
    {
      "element": "//a[@id='js-attr']",
      "events": ["click"]
    },
    {
      "element": "//a[@id='listener']",
      "events": ["click", "focus"]
    }
  ]
}
```

It is also possible to specify a filter (by XPath) and receive only the handlers registered on matching elements.

Send:

```json
{
  "command": "handlers",
  "filter": "id('listener')"
}
```

Receive: (e.g for the handlers.html test case)

```json
{
  "handlers": [
    {
      "element": "//a[@id='listener']",
      "events": ["click", "focus"]
    }
  ]
}
```

The XPath identifiers returned are Artemis’ internally generated ones and may not match the filter, even if it selects a single element.

• click Clicks on an element specified by XPath.

For now then only type of click is a JavaScript-level click, with no option for a GUI click.

N.B. This is now just a special case of the newer event command.

Send:

```json
{
  "command": "click",
}
```
There is an optional method field, which allows you to choose the type of click performed. Possible values are:

- **simple** (default) Just generates a click event, in the same way as the event command would.
- **simulate-js** Uses JavaScript events to simulate a user click.
- **simulate-gui** Uses GUI events to simulate a click.

N.B. This click is done by clicking the coordinates at the centre of the element. If the element is behind another element or the element bounding box is larger than the clickable/visible area, this command can miss and click the wrong thing.

```json
Send:
{
    "command": "click",
    "element": "id("clickable")",
    "method": "simulate-js"
}
```

Receive: {"click": "done"}

There are also some custom Artemis event types which are not the standard JavaScript events. These are handled separately by Artemis and are not triggered as JavaScript events directly.

So far there is only one implemented: for pressing **Enter** on a form field (e.g. to submit the form).

Send (e.g. on form-submission.html):

```json
Send:
{
    "command": "event",
    "element": "id("input-text")",
    "event": "ARTEMIS-press-enter"
}
```

Receive: {"event": "done"}

**page** Returns information about the current page (the URL, page title, and DOM statistics).

Send: {"command": "page"}

Receive:
The optional “dom” parameter can be set to True to include the entire DOM dump.

Send:

```
{
    "command": "page",
    "dom": true
}
```

Receive:

```
{
    "url": "http://www.example.com",
    "title": "Example Domain",
    "dom": "<html> ... </html>",
    "elements": 12,
    "characters": 1262
}
```

- **element** Returns the string representation of each element (if any) matching a given XPath.

Send: (e.g. for the click.html test page)

```
{
    "command": "element",
    "element": "id("clickable")"
}
```

Receive:

```
{
    "elements": [ "<a href=" id="clickable">Click here to add new buttons to the page.</a>" ]
}
```

There is also an optional property field which will return the string representation of that object property instead.

Send:

```
{
    "command": "element",
    "element": "id("clickable")",
    "property": "nodeName"
}
```

Receive:

```
{
    "elements": [ "A" ]
}
```
• **fieldsread** Returns a list of the form fields which have been read by different events since the last page load.

Send: `{ "command": "fieldsread" }`

Receive: (e.g. from form.html test page)

```json
{
  "fieldsread": [
    {
      "element": "//button[1]",
      "event": "click",
      "reads": [
        {
          "count": 2,
          "field": "//input[@id='first']"
        }
      ]
    },
    {
      "element": "//button[2]",
      "event": "click",
      "reads": [
        {
          "count": 1,
          "field": "//input[@id='second']"
        }
      ]
    },
    {
      "element": "//button[3]",
      "event": "click",
      "reads": [
        {
          "count": 3,
          "field": "//input[@id='first']"
        },
        {
          "count": 3,
          "field": "//input[@id='second']"
        }
      ]
    }
  ]
}
```

Each “event object” contains the event type triggered and target element (XPath as passed in via the `click` command), and a list of the form fields which were read by the handler for that event. Each of these “read objects” contains an XPath to the field and a count of the number of times the field value was read (at a low level in the JavaScript interpreter).

• **forminput** Injects values into form fields and triggers their change handlers. The method of injection can be changed with the optional `method` parameter (see below).

Send:

```json
{
  "command": "forminput",
  "field": "id('input-text')"
}
```
The valid element types for `field` are `input` and `select`.

The `value` property can be set to a string (as above), integer, or bool. Strings are used when injecting into text fields or select boxes. Integers can be used to inject into a select box by index (sets the `selectedIndex` property to the given value). Booleans are used to inject into inputs with type `checkbox` or `radio`.

The allowable combinations of `field` and `value` are:

<table>
<thead>
<tr>
<th>Compute</th>
<th><code>input</code> (not checkbox or radio)</th>
<th><code>input</code> with type checkbox or radio</th>
<th><code>select</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Sets <code>.value</code></td>
<td>Invalid</td>
<td>Sets <code>.value</code></td>
</tr>
<tr>
<td>Int</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Sets <code>.selectedIndex</code></td>
</tr>
<tr>
<td>Bool</td>
<td>Invalid</td>
<td>Sets <code>.checked</code></td>
<td>Invalid</td>
</tr>
</tbody>
</table>

For example, the following commands are all valid on the form-injections.html test case:

```json
{
  "command": "forminput",
  "field": "id('input-text')",
  "value": "Hello, world."
}
```

This one sets the checkbox to ticked:

```json
{
  "command": "forminput",
  "field": "id('input-checkbox')",
  "value": true
}
```

When injecting into a select box, the `value` attribute of the appropriate `option` element must be given, which is not necessarily the text which appears in the UI:

```html
<select id="input-select">
  <option value="first">First Option</option>
  <option value="second">Second Option</option>
  <option value="third">Third Option</option>
</select>
```

This one selects “Third Option” in the UI:

```json
{
  "command": "forminput",
  "field": "id('input-select')",
  "value": "third"
}
```

This one also selects “Third Option”, by using the index:
The form-injections.html example includes a ‘marker’ element so you can confirm the form input worked:

```json
{
  "command": "element",
  "element": "id('status')"
}
```

There is a method field, which allows you to choose the type of injection performed. Possible values are:

- **inject** Inject the value into the .value property (depending on the input type; see above).
- **onchange** (default) Inject the value and trigger the onchange handler for the form field.
- **simulate-js** Uses JavaScript events to simulate a user filling the form field as closely as possible. The support for text inputs is currently much more sophisticated than for checkboxes, radio buttons, and select boxes.

  When **simulate-js** is used, an extra optional property noblur can be set to boolean true to stop the ‘blur’ (de-focus) event being triggered on this element once the injection is complete. This can be useful (for example) to stop auto-complete boxes being hidden when the field is deselected.

- **simulate-gui** Not yet implemented.

Send:

```json
{
  "command": "forminput",
  "field": "id('input-text')",
  "value": "Hello, world.",
  "method": "inject"
}
```

Receive: "{ "forminput": "done" }"

- **xpath** Evaluates an XPath query and returns the result.

  The result may be a String, Number, Boolean or Node-Set. Node-sets are represented as an array of the string representations of the nodes.

  Node-set (all examples on the click.html test case):

```json
{
  "command": "xpath",
  "xpath": "/h1"
}
```

```json
{
  "result": [ 
    "<h1>Clickable elements</h1>" 
  ]
}
```
It is also possible to provide a list of XPaths to evaluate. The result will be a list of the results of each XPath as above:

```json
{
  "command": "xpath",
  "xpath": [
    "//h1",
    "string(//h1)",
    "string-length(string(//h1))",
    "string-length(string(//h1)) > 10"
  ]
}
```

```json
{
  "result": [
    "<h1>Clickable elements</h1>",
    "Clickable elements",
    18,
    true
  ]
}
```

N.B. Non-matching queries are handled as normal in a browser’s XPath evaluation:
An XPath which cannot be evaluated (because it is invalid) will return an error.

- **windowsize** Set the size of the browser window.
  Send:
  ```json
  {
    "command": "windowsize",
    "width": 1024,
    "height": 768
  }
  ```
  Receive: { "windowsize": "done" }

- **concolicadvice** Allows the server to record traces into a concolic execution tree and return advice about new form field values which can lead to new exploration.
  See the [Server Mode - Concolic Advice](#) documentation for details.

- **evaluate-js** Evaluates a JavaScript string on the current page.
  Send:
  ```json
  {
    "command": "evaluatejs",
    "js": "document.getElementById('clickable').click()"
  }
  ```
  Receive: { "evaluatejs": "done" }

- **setsymbolicvalues** Sets the internal symbolic values of variables accessed via `artemisInputBoolean()`, `artemisInputInteger()`, and `artemisInputString()`. This can be used for testing the internal concolic engine of the platform. For normal testing of web pages the `forminput` command should be used instead for concolic testing.
  The `values` parameter is a mapping from variable names (strings) to values, which may be strings, integers or booleans.
  The `reset` parameter is optional, and if set to `true`, the internal symbolic value table will be cleared before setting these replacement values.
  Send:
  ```json
  {
    "command": "setsymbolicvalues",
    "values": {
      "X": "Hello",
      "Y": 123,
      "Z": true
    },
    "reset": true
  }
  ```
  Receive: { "setsymbolicvalues": "done" }

Now a call like the following would update the DOM with the injected symbolic values:
The **coverage** command returns a report of the line coverage from the executed commands. The line coverage is taken since the server was started, and cannot be reset.

The report is a list of reports for each distinct JavaScript source (web page, JS file, etc.). The line-by-line coverage report is human-readable, not in a good machine-readable format. It can be parsed with analyse-coverage.py.

The `linescovered` parameter is a list of line numbers which were covered.

N.B. A line is considered covered if some interpretation was done on that line. So the close-braces of if statements, else statements, blank lines, and so on will never be considered covered.

Send:

```json
{
   "command": "coverage"
}
```

Receive:

```json
{
   "coverage": [
      {
         "url": "...",
         "line": "...",
         "linescovered": [...] // list of line numbers
         "report": "..."
      },
      {... // more reports}
   ]
}
```
The server is able to record traces (sequences of actions) symbolically and provide suggestions for new form field inputs to use with that sequence which will result in new JavaScript being executed.

The main server mode documentation is here: Server Mode.

6.1 Concolic Advice Model

Each trace recorded is associated with a “sequence ID” identifying that particular action sequence. The required calling sequence is:

- **begintrace “MySequence”**
  - There must be no trace already in progress.
- Some actions here, e.g. forminput, click, etc.
- **endtrace “MySequence”**
  - Sequence identifier must match the preceeding begintrace.
- **May record more traces, with any sequence identifiers.**
  - If using the same sequence identifier “MySequence” the same actions must be performed while recording.
- **advice “MySequence”**
  - Must have recorded at least one trace for “MySequence” already
  - Should not be called while recording a new trace.
- **Any actions executed outside of a begintrace/endtrace block will not be recorded, for example actions to reset the state before re-running a new trace for the same sequence.**

N.B. There are some relaxations to this format allowed by specifically requesting them. See the allowduringtrace option for concolicadvice and the implicitendtrace option for begintrace.
6.1.1 Trace matching

When traces are recorded for a certain sequence they are added to a tree of all execution paths for the execution in that trace. The concolic advice makes suggestions for exploring new unseen branches in this tree.

In order to add new traces to the tree, they must have a prefix which matches the top of the tree. For example, if the first trace has an initial action of filling field A, then all subsequent traces (for the same sequence ID) must begin with filling field A. The simplest way to keep all new traces matching is to make sure all traces recorded with the same sequence ID should execute exactly the same actions during the trace recording. The only thing which is safe to change is the values injected into form fields.

In reality the requirement is slightly weaker. The prefix of the trace has to match the prefix of the tree until the trace reaches a new unexplored area, after which there are no restrictions, other than matching with future traces. For example, if selecting “Return” instead of “One-way” from a trip-planning form causes an extra “Return date” field to be shown, it is safe to only fill “Return date” in the traces where “Return” has already been selected and ignore it in the traces where “One-way” is selected. In practice it is difficult to know when this is safe without inspecting the concolic tree so the above rule of always using identical traces is recommended.

6.1.2 Advice returned

The advice returned will contain suggested values for any form fields which were seen in any branch condition in executed JavaScript code during the trace recording.

This means it may not include all the form fields which were filled during the trace - some of them will have no validation and are considered “uninteresting” by our analysis.

It also means that values may be returned for form fields which were not in the original trace. A typical example of this is if the trace contains only a submit button click, in which case it likely causes some form field validation (leading to some contraints and so some value suggestions) even though no field is filled. To exercise these branches the client can either begin a new sequence which includes these fields, or arrange for the fields to have those values set before beginning the [same] trace. In the latter case the analysis will not be able to see any per-field validation for these fields which are set outside the trace, so some conditions may be missed.

6.1.3 Form restrictions

Radio buttons and select boxes (drop-down lists) have implied constraints. If a select box only contains options “A”, “B” and “C” then the concolic analysis will only return one of those strings as a suggested value for that input.

In the particular case of select boxes we have some special support for dynamically changing forms. A common pattern is to have two (or more) select boxes whose values update based on earlier selections. For example if the fields are “Country” and “City” then selecting “UK” in the first will show a list of UK cities to choose from in the second field. Selecting “Denmark” in the first field will give options of Danish cities, and so on. In this case the analysis can see the updated values and will only make suggestions of valid pairs. ({"Country": "UK", "City": "London"}) would be valid but ({"Country": "UK", "City": "Copenhagen"}) would never be returned as a suggestion.

Note that this pattern is only supported if the “Country” and “City” fields are both filled during the trace recording and that no other types of dynamic form modifications are supported by the analysis.

The following tests from server.py and test pages show some examples of these types of forms:

- test_select_restrictions and concolic-select-restrictions.html
- test_radio_restrictions and concolic-radio-restrictions.html
- test_select_restrictions_dynamic and concolic-select-restrictions-dynamic.html
6.2 Commands

- **concolicadvice > begintrace** Begin recording a new trace (for a new or existing sequence). There must not be another trace in-progress.

  Send:

  ```json
  {
    "command": "concolicadvice",
    "action": "begintrace",
    "sequence": "MySequenceID"
  }
  ```

  Receive:

  ```json
  {
    "concolicadvice": "done"
  }
  ```

  The client can now send commands to execute actions (forminput, click, etc.) which will be recorded into the trace and saved in the concolic tree for sequence “MySequenceID”.

  There is an optional boolean parameter implicitendtrace (default false) which allows this command to run even if there is already a trace in-progress. In this case the existing trace is ended (as if endtrace had been called) and the new trace is immediately started.

  **Errors:** If any trace is already in progress (unless implicitendtrace is set).

- **concolicadvice > endtrace** End recording a trace. There must be a trace with the matching sequence ID in-progress.

  Send:

  ```json
  {
    "command": "concolicadvice",
    "action": "endtrace",
    "sequence": "MySequenceID"
  }
  ```

  Receive:

  ```json
  {
    "concolicadvice": "done"
  }
  ```

  **Errors:** If there is no trace in progress; if the in-progress trace used a different sequence ID.

- **concolicadvice > advice** Request advice on form field values. There should not be a trace in-progress.

  The optional amount parameter (default value 1) requests that number of suggested form field assignments from the server. If there are less than this number available, all available advice will be returned. Setting amount to 0 will return all available advice.

  N.B. It is meaningless but allowed to send the amount parameter with other concolicadvice actions as well.

  It is safe to call this command multiple times consecutively.

  Send:
This example is a list of three separate suggested new traces. The first trace fills field `input1` with value “Hello” and field `input2` with value “World”, and so on.

If there is no more advice available for that sequence, then no values are returned:

```
{
  "concolicadvice": "done",
  "sequence": "MySequenceID",
  "values" : []
}
```

N.B. This result is not necessarily final. If there are outstanding traces which have been suggested by
Artemis but not yet executed then these may open up new possible explorations when they are executed.

Types: The type of the suggested value can be either string, int or bool, depending on the field type. They follow the same rules as the forminput command.

For example the response could be:

```json
{
    "concolicadvice": "done",
    "sequence": "MySequenceID",
    "values": [
        {
            "field": "//input[@id='my-text-box']",
            "value": "Hello"
        },
        {
            "field": "//input[@id='my-select-box']",
            "value": "Hello"
        },
        {
            "field": "//input[@id='my-select-box-accessed-by-index']",
            "value": 1
        },
        {
            "field": "//input[@id='my-check-box']",
            "value": true
        },
        {
            "field": "//input[@id='my-radio-button']",
            "value": false
        }
    ]
}
```

There is also an option boolean parameter allowduringtrace (default false) which allows this command to be called while a trace is in-progress. The information gathered by an in-progress trace will not be available until endtrace is called, so calling advice during a trace does not gain anything. This means that if the first trace for “MySequenceID” is in-progress when advice is requested for “MySequenceID” then it will return an error because there is no concolic knowledge of that sequence yet.

Errors: If there has not been any trace recorded with that id; if there is any trace in-progress (unless allowduringtrace is set).

- **concolicadvice > statistics** Retrieves the statistics about a certain concolic tree. This is purely informational, and not intended to be used to drive the analysis.

Send:

```json
{
    "command": "concolicadvice",
    "action": "statistics",
    "sequence": "MySequenceID"
}
```

Receive:

```json
{
    "concolicadvice": "done",
```
"sequence": "MySequenceID",
"values": {
  "Alerts": 1,
  "ConcreteBranchesFullyExplored": 0,
  "ConcreteBranchesTotal": 0,
  "CouldNotSolve": 0,
  "EndFailure": 0,
  "EndSuccess": 0,
  "EndUnknown": 1,
  "InterestingDomModifications": 0,
  "Missed": 0,
  "PageLoads": 0,
  "Queued": 1,
  "SymbolicBranchesFullyExplored": 0,
  "SymbolicBranchesTotal": 0,
  "TracesRecordedInTree": 1,
  "Unexplored": 0,
  "UnexploredSymbolicChild": 1,
  "Unsat": 0
}
}

The keys here are mostly the same as those from the output of the normal concolic runtime. There are some missing, as in this case the stats are only generated from the tree, not the entire concolic runtime. There are also a couple of new keys: `TracesRecordedInTree` (expected to be the same as `DistinctTracesExplored`, although it is calculated from the tree, not the trace merger), and `Queued`, the number of branches which have been suggested but not yet explored.

Extending Artemis:
The repository is divided into four folders:

- **artemis-code/** Source code, tests and scripts making up the core Artemis tool.
- **WebKit/** An instrumented version of WebKit.
- **ailproxy/** A web-proxy for mocking server-side interfaces.
- **docs/** Source files for the documentation you are reading.

### 7.1 Installation and Usage

For installation instructions see the INSTALL file and for usage run the command `artemis --help`.

### 7.2 The Artemis Tool

The Artemis tool is written in C++ using Qt 4.8 and the qmake build system. The main qmake project file is `artemis-code/artemis-code.pro`, which in turn includes `artemis-code/artemis-core.pri` (the latter is also included by the unit-test project).

The central classes are shown in the following diagram:
The main application loop and initialization of the application resides in `Runtime` (`runtime/runtime.h`). The main application loop maintains the `WorkList` (`runtime/worklist/worklist.h`), using an instance of `InputGeneratorStrategy` (`strategies/inputgenerator/inputgeneratorstrategy.h`) to generate new event sequences for the worklist. Event sequences are executed using an instance of `WebKitExecutor` (`runtime/browser/webkitexecutor.h`), responsible for all interaction with the instrumented WebKit library described in the next section. Finally, the `WebKitExecutor` gathers the feedback from WebKit in an `ExecutionResult` (`runtime/browser/executionresult.h`) and in updates to `AppModel` (`model/appmodel.h`), used by other parts of Artemis.

### 7.3 WebKit Instrumentation

We instrument WebKit (checkout anno 2011-12-28) to observe the execution of the tested application, gathering feedback for subsequent iterations.

The WebKit code-base is extended with a JavaScript debugger (`WebKit/Source/WebCore/instrumentation/listenerdebugger.h`) and a number of listening-points boxed in `ifdef ARTEMIS` and `endif` directives. The debugger and listening-points invoke methods on a global instance of `QWebExecutionListener` (`/WebKit/Source/WebKit/qt/Api/qwebexecutionlistener.h`), denoted the execution listener.

The execution listener provides a Qt signal interface used by Artemis for gathering feedback. In general, each method invocation from a listening-point is translated and emitted as a signal. Furthermore, the execution listener translates from the WebKit world into the Qt world, such as translating WebKit’s StringImpl class into Qt’s QString. As a rule, WebKit related types is not allowed in Artemis, and Qt related types is only allowed in the execution listener (and other Qt provided classes making up the WebKit-Qt interface).
This section documents various modifications made to WebKit as part of our instrumentation.

Note, we require WebKit and specifically the JavaScript Core interpreter to be compiled in a non-JIT mode and in 64bit-mode (read: we do not support JIT compiling and the 32bit compatible version of WebKit).

8.1 Tracking Values Symbolically

WebKit has been extended with symbolic values and semantics mirroring the concrete values and semantics respectively. Symbolic values are injected at predetermined sources (currently only the value property on input element DOM objects).

As an example, accessing the value property on DOM node D returns a concrete string, denoted C, marked as a symbolic value, denoted S, originating from D. Any concrete operation operation on C will be matched with a symbolic operation on S. Thus, if the length property is accessed on C it will return a concrete value C2 representing the concrete length of the string, and C2 will be marked with a symbolic value S2 representing the symbolic length of the symbolic string S.

We say that there exist a number of mutators in WebKit taking a number of inputs I_0 \ldots I_n and outputting an output value O. Artemis instruments WebKit such that in all mutators, the output value O is marked with a proper symbolic value taking into account the concrete semantics of the mutator and the concrete and symbolic values of the inputs.

The following diagram gives an overview of the different concrete values, mutators and symbolic values and how they relate in the implementation.
Note, there exist three levels of concrete values in WebKit: JSValues, object interfaces and internal objects. The JSValues are used to represent both primitive values and pointers to objects in JavaScript. JSValue is the primary type being passed around in the JavaScript interpreter. If JSValue points to an object, then it has a pointer to an object interface. This interface acts as a proxy to internal objects, usually conducting type conversion while delegating business logic and storage of values to the internal objects. The object interfaces are automatically generated in order to allow different JavaScript interpreter implementations to interface with the same internal objects.

We have identified two primary mutators, the JavaScript interpreter and native functions. In general, the JavaScript interpreter only manipulates the primitive values stored in JSValue, while the native functions operate on everything from JSValue to the internal objects.

- Symbolic values are attached to all primitive values stored in JSValue and the interpreter has been instrumented to maintain the symbolic values for all operations on JSValues.
- A subset of native functions operating on JSValues have been instrumented.
- A subset of interface objects track symbolic values for their concrete properties (JSString and the value property on input elements).
- A subset of native functions operating on interface/internal objects have been instrumented (JSString and input elements).

A note on strings: Symbolic strings are marked symbolic both in the JSValue pointing the the JSString object and in the JSString object itself. The JSValue will make sure to propagate its symbolic value to the JSString. The JSString is immutable so it will never change its own symbolic value, thus keeping the two consistent. The symbolic value needs to be represented in the JSString since the native functions operating on JSString never gets a reference to the enclosing JSValue (and these functions derive new symbolic values based on the current string, e.g. string length).

### 8.2 Symbolically Enhanced JavaScript Values

JavaScript values in WebKit are NaN encoded. NaN encoding is used in WebKit to represent JavaScript values, denoted JSValue. Each JSValue is a 64-bit value, either representing a *double*, *integer*, or a *pointer* to a JSCell. A detailed description of this encoding can be found in JSValue.h, here and here.
We want to enhance each concrete JavaScript value with a pointer to a symbolic representation of the same value. Furthermore, we want this to be fairly efficient, and only decrease efficiency if a symbolic value is present.

If a concrete values is not associated with a symbolic value we use the current NaN encoding for performance.

We can’t represent both a concrete value and a symbolic value within 64bit at the same time. In this case, we tag the value to be symbolic and store a pointer to an object in turn pointing to the concrete and the symbolic value.

Specifically, we change the following patterns:

```
Pointer { 0000:PPPP:PPPP:PPPP
        / 0001:****:****:****
Double { ...
        \ FFFE:****:****:****
Integer { FFFF:0000:IIII:IIII
```

into:

```
Pointer { 0000:0PPP:PPPP:PPPP
        / 0001:****:****:****
Double { ...
        \ FFFE:****:****:****
Symbolic:
Object { FFFF:3PPP:PPPP:PPPP
True { FFFF:7PPP:PPPP:PPPP
False { FFFF:5PPP:PPPP:PPPP
Null { FFFF:1PPP:PPPP:PPPP
Double { FFFF:9PPP:PPPP:PPPP
Integer { FFFF:DPPP:PPPP:PPPP
Integer { FFFF:C000:IIII:IIII
```

Notice that 64bit pointers only take up 44bit, leaving the top 20bit unused.

We extend the value objects representation of 32 bit integers. Here, 4bit of the previously unused area for integers are used to tag a specific symbolic type, and the remaining bits are used for storing a pointer to a combined concrete and symbolic value.

Notice, that the bit patterns for the different symbolic values are as follows:

```
a b c d  
Object 0 0 1 1
Null 0 0 0 1
True 0 1 1 1
False 0 1 0 1
S. Int 1 1 0 1
S. Double 1 0 0 1
Int 1 1 0 0 (not symbolic)
```

The (a) bit indicates if the value is numeric or not, the (d) bit indicates if the value is symbolic or not (in order to differentiate normal concrete integers).

### 8.3 Special Casing Symbolic Strings

Strings are represented by a JSValue (object type) who points to a JSString. A string is made symbolic by marking both the JSValue and JSString as symbolic. It is not enough to only mark the JSValue as symbolic, because a number of internal library functions (which we need to instrument for correct symbolic handling) only operate on the JSString.
object, and can’t access the JSValue pointing to it. We fix this by propagating the symbolic information from the JSValue to the JSString.

This can cause problems if two distinct JSValue objects point to the same JSString for optimization purposes. Some special handling exist to avoid this case.

### 8.4 Special Casing Symbolic Objects

We do not support symbolic objects in general. However, we do mark specific objects as symbolic in order to implement symbolic handling of specific instances of objects.

- We make the result returned by regexp operations (who return arrays or null) symbolic. The symbolic value from these operations is treated as a special null or non-null symbolic value, in order to reason about the outcome of a regexp match.

### 8.5 Special Casing Indirect Symbolic Values

- We mark objects as indirect symbolic if they are accessed using a value lookup using a symbolic index. This is used as a flag in order to implement symbolic value properties on option elements within a select element soundly. See issue #82, access pattern 3.

### 8.6 Symbolic Handling of Native JavaScript Functions and DOM

As an easy reference, we use http://www.w3schools.com/jsref/ as an easy-to-read reference of the API of native JavaScript-, browser-, and DOM objects. We want to support all parts of the API which can read, modify, or create symbolic values - either by emitting constraints or emitting a warning indicating incomplete symbolic handling.

#### 8.6.1 Symbolic Support

String.{charAt, concat, match, replace, search, toString, valueOf, length, indexOf},

**Note:** String.replace(S2, S3) only supported if String is symbolic. Warnings are emitted if S1 are not symbolic but S2 or S3 are symbolic.

RegExp.{exec, test},

**Note:** RegExp.exec only support non-gobal regular expressions. If the regular expression contains the global flag, then only the first match using exec is supported. Warnings are emitted for subsequent matches.

parseInt,

Math.{floor, ceil, max, min},

**Note:** JavaScript represents all numeric values as doubles, while we represent numeric values as integers. Thus, Math.{floor, ceil} returns the input symbolic value unmodified. This introduces some degree of imprecision.
in our solutions.

Input Checkbox.checked, Input Radio.checked, Input Text.{value, valueAsNumber}, Select.{value, selectedIndex}, OptionGroup.selectedIndex, Option.value

Note: All other properties on the Input {Checkbox, Radio, Text}, Select, OptionGroup and Option objects are not supported and do not emit warnings.

8.6.2 Usage Warnings

Math.{abs, acos, asin, atan, atan2, cos, exp, log, pow, random, round, sin, sqrt, tan},
String.{charCodeAt, lastIndexOf, localeCompare, slice, split, substr, substring, toLocaleLowerCase, toLocaleUpperCase, toLowerCase, trim, trimLeft, trimRight, anchor, big, blink, bold, fixed, fontcolor, fontsize, italics, link, small, strike, sub, sup, fromCharCode},
RegExp.{constructor, compile}, decodeURI, decodeURIComponent, encodeURI, encodeURIComponent, eval, isFinite, isNaN, parseFloat, escape, unescape

Note: RegExp.{constructor, compile}(A1, A2) emit warnings if A1 or A2 are symbolic. Thus, we only support concrete regular expressions.

8.6.3 No Symbolic Support and No Usage Warnings

Array.*, Boolean.*, Date.*, Number.*, RegExp.{global, ignoreCase, lastIndex, multiline, source, toString}, String.<index lookup>,
Window.*, Navigator.*, Screen.*, History.*, Location.*,
Input Checkbox.*, Input Radio.*, Input Text.*, Select.*, OptionGroup.*, Option.*

8.6. Symbolic Handling of Native JavaScript Functions and DOM
9.1 Exporting Symbols in WebKit

By default, all symbols (functions, classes, etc.) are hidden in the compiled WebKit library. Only a subset of symbols (primarily Qt related) are exposed and usable from Artemis.

A list of all exported symbols is stored in WebKit/Source/qtwebkit-export.map. Edit this file to export new symbols.

9.2 Instrumenting the DOM

The DOM API in WebKit is auto generated from a set of IDL files specifying the exact API and its behavior. Artemis introduces instrumentation into this layer partly by modifying the IDL files and partly by modifying the perl script processing the idl files.

The perl script can be found in WebKit/Source/WebCore/bindings/scripts/CodeGeneratorJS.pm.

IDL files can be found in these folders:
  • WebKit/Source/WebCore/html
  • WebKit/Source/WebCore/dom
The QT coding style is used with the following additions:

10.1 Member variables

Member variables (private and public) are prefixed by `m`.

10.2 Slots & Signals

Slots are prefixed by `sl` and signals `sig`.

10.3 Implicit or explicit this pointers

Implicit this pointers are used when possible.

```cpp
void A::f()
{
   this->mCount = 0; // wrong
   mCount = 0; // correct
}
```

10.4 Memory management

All memory management strategies directly supported by QT are accepted in addition to stack allocated objects ala. RAI/L.

For user-defined types smart pointers (as in QSharedPointer) is highly encouraged.
10.5 If statements and brackets

Contrary to the QT coding style, brackets are always used with if statements even if the body of the if statement only consist of a single line. This is to improve robustness.

10.6 Whitespace for pointers and references

```cpp
// correct
int* value;
void A::func(char* c) { ... }

// wrong
int *value;
void A::func(char *c) { ... }
```