Web App UI Layout Sniffer

First Author **Raymond Peng** pengr@vmware.com
Second Author **Xin Jing Hu** xhu@vmware.com
Product Globalization, VMware, Beijing, 100190, China

Abstract

The problem: the UI layout looks great in the English version of VMware web applications but when the product is localized for our global markets, the UI layout doesn't work.

UI layout bugs found during product localization can be as high as 15–35 percent. Testing, triaging and fixing so many UI layout bugs involves multiple cross-functional teams, and requires extra time and budget. Taking one layout bug as example: QE tests and reports it, assigns it to Dev for fix, and QE then verifies the fix to sign off. If the bug is not fixed correctly, QE re-opens the bug and repeats the whole process. This process is complex and efficiency is painfully low. Consequently, it dramatically slows down product delivery in today's fast-paced and competitive market.

These bugs happen for various reasons but we rarely have time to wait until the testing team finds and reports them in Bugzilla. We must find a better way to identify these bugs.

We believe that UI designers or developers can and should avoid these bugs, and in an automatic smart way as they work from mockup to layout coding.

We propose to help the designer sniff potential layout problems early in the software development cycle to kill the problem from the beginning.

Our idea is to develop and adopt a browser extension to call and refresh UI pages by replacing English UI strings with machine translation (MT) that is close to human translation. UI designers or developers can identify potential layout problems at any time, just as an actual user would see in their production environment.

1. Background

As below sample in Figure 1 shows, estimated an 15%-35 % globalization (g11n) bugs belong to the UI layout category for a typical VMware web application, especially for products or features released for the first time.

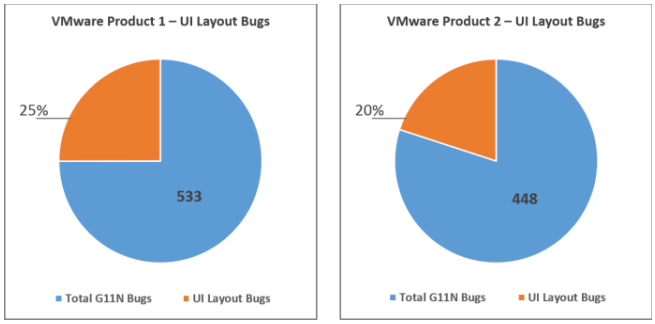

Figure 1. Layout Bug Percentages for Three VMware Products

Testing, reporting, triaging and fixing so many layout issues involves multiple parties and requires a high investment in time and cost, and dramatically slows down time-to-market.

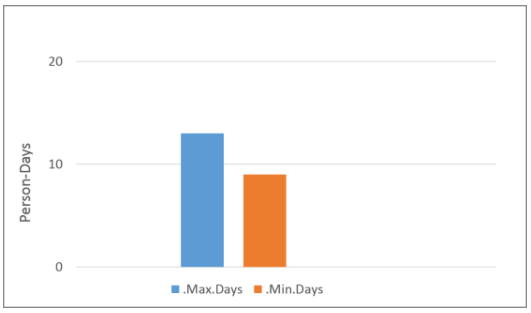

Figure 2. Bug-Fixing Workload for Product 1

For example, 132 UI layout bugs were reported for product 1 and 91 layout bugs for product 2, as illustrated in Figure 1. One engineer can fix 10–15 layout bugs in one working day. To fix the 132 bugs for Product 1 then would need 9–13 person-days, as shown in Figure 2.

Besides the time and cost, another challenge is that some layout bugs can be hard to resolve because the fix does not just involve UI resizing but depends on a localization vendor. Usually there is no time for a re-translation cycle given the schedule pressure.

These problems obviously drag down VMware's ability to deliver better value to global markets quickly.

Our idea, the Web Application UI Layout Sniffer, is designed for VMware web applications, to prevent and solve layout problems at early phases of the software development cycle. With our technology, the expected result is that there will be no layout bugs during i18n and l10n testing.

2. Overview

The web applications mainly render the UI layout within browser in two ways:

Client rendering: The initial request loads the page layout, CSS and JavaScript. All common contents like static pages will be returned from backend server except that some or all of the content isn't included. Instead, the JavaScript makes another request (Ajax), gets a response and generates the appropriate HTML.

Server rendering: The initial request loads the page, layout, CSS, JavaScript and content. For subsequent updates to the page, the client-side rendering repeats the steps used to get the initial content. Namely, JavaScript is used to get some JSON data, and template is used to create the HTML.

In standard processes, QA has to wait and use a pseudo build until all UI freeze to start a series of testing. Once layout bug is found, the whole process has to repeat again. Nevertheless, it is too late and risky to fix global CSS settings for developers at this late stage in product release cycle.

Our idea eliminates this dilemma and streamlines the process from the beginning because we kill the possibility of layout bugs at the design and development stage without needing to wait for a UI freeze to verify and report such bugs. We developed a browser plug-in to support both major types of UI layout rendering. Contents script that can read details of the web pages whenever the browser visits, and analyze contents in text node.

We utilize Machine Translation from a web portal to represent the target language just like

being translated by human translators, so developers can check and verify potential layout issues. The plug-in also supports customized pseudo translation by extending English string length by 20–80 percent longer, as chosen by the user, for a static HTML check. Obviously, using machine translation is more accurate and suitable for detecting layout issues since machine translation results are much closer to the real translations than the pseudo strings. This could increase the bug hit rate and provide much more valuable references for UI designers to adjust the layout related parameters.

Typical testing and user scenarios. UI designers and developers using our browser extension can detect potential layout issues at any time on a daily basis.

3. Advantages

Facing with a real problem world of UI layout bugs, we have following goals in mind:

- **Lightweight:** Most automation tools are hard to install, given the heavy, dependent environment needed, but a browser extension is easy to install, without requiring that extra assets be pre-installed.
- **Easy to use:** No training is needed to use this tool to check potential layout issues. It's a "what you see is what you get" (WYSIWYG) application for web UI designers.
- **MUI supported:** The tool supports a multi-language interface so UI designers from different countries have no language obstacle using it on a daily basis.
- **Fast performance:** Adopting a DOM tree to read each node one time, it stores the text information and uses the xPath to locate the nodes, so users will not feel a response delay.
- **Easy to extend:** For now, the tool supports two types of MT engines, but it can be configured to connect to all kinds of MT engines.

4. Architecture

Figures 3 show the architecture of the Web App Sniffer. It consists of three core components: the DOM Tree Parser, Pseudo Module, and MT Adapter.

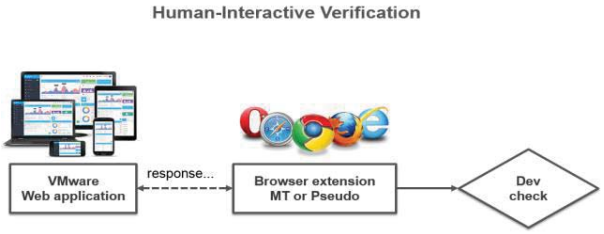

Figure 3. Human-Interactive Verification

4.1. DOM Tree Parser

This component is used to parse the terminal DOM tree and extract the text nodes and natural language processing words. To handle text in a DOM structure is to get intimate with the text

nodes themselves. The application needs to distinguish different kind of nodes. As a W3C standard, the key differences are summarized as follows:

- Text nodes don't have descendant (or child nodes), instead only text content without any HTML or XML markup.
- Almost all contents about a text node are contained in its data (or node value) property, which contains whatever text the node encapsulates.
- Text nodes don't trigger events and can't have any styles applied.

These text nodes will appear in the DOM structure just like element nodes. For example, if we consider this paragraph:

```
<p>
   <a href="/ dashboard ">Go back to dashboard! </a><br/>
   This dashboard reference ID is 12345678.
</p>
```

Its DOM structure is as follows:

```
-> P ELEMENT-> TEXT NODE (data: "n   ")
-> A ELEMENT (href: "/dashboard")
-> TEXT NODE (data: "Go back to dashboard!")
-> BR ELEMENT
-> TEXT NODE (data: "n    This dashboard reference ID is 12345678.n")
// temporary. innerHTML is now:
// "n    <a href="/ dashboard ">Go back to dashboard! </a><br/>This dashboard reference ID is 12345678."
// |_________________________|__________________|_________|__________________________________|
//              |                        |                |                          |
//        ELEMENT NODE             TEXT NODE      ELEMENT NODE             TEXT NODE
```

The parser module supports two types of replacing text sources: using pseudo strings to replace the original text and using machine translation.

Pseudo: The tool supports using pseudo tags, traversing all text nodes according to the node type property, including the nested nodes. At the same time, it does not touch the contents in some special tags as below:

```
var arr = ["STYLE", "IMG", "NOSCRIPT", "SCRIPT"];
```

In special HTML structure, before traversing the nodes, it also needs to traverse the Iframe node.

By current design, in pseudo model the tool will just add pseudo string and pseudo mark simply to expand English string length by 20 – 80 percent. Annotations and semantic are not considered.

Machine translation (MT): The tool supports using MT to pull translation to replace English text. The HTML contains all possible markups that a typical VMware web application has. This means the MT engine translates the whole HTML content including its markup. How-

ever, the MT engine does not support HTML markup well, e.g. translating UI text while keeping HTML markup intact. If we don't do any extra processing of the machine translated HTML files, we will have following errors in the translated HTML:

1) Disrupted tags — If the MT engine doesn't consider HTML structure, they can potentially move the HTML tags randomly, leading to disrupt tags in the MT result.
2) Wrongly placed embedded tags — The example given below illustrate this. It is more serious if content includes links and link targets were swapped or randomly given in the MT output.

```
$ echo 'Hello <b>World</b>' | apertium en-es -f html

<b>Hola</b> Mundo
```

3) Missing embedded tags - Sometimes the MT engine loses embedded tags in the translation process.
4) Split embedded tags - During translation a single word can be translated to more than one word. If the source word has a markup, for example, an <a> tag. Will the MT engine apply the <a> tag wrapping both words or apply to each word?

All of the above issues can impact the accuracy of layout issues detection. To avoid these issues, our application adopts embedded tags mapping, using the concept of fuzzy matches. Essentially the algorithm does a fuzzy match to find the target locations in translated text to apply embedded tag and content given to MT engine is plain text only.

1) For the text to be translated, the tool finds the text of inline embedded tags like bold, italics, links etc. We call it sub-units.
2) The tool passes the full text and sub-units to the MT engine. Use some delimiter so that we can do the array mapping between source items (full text and sub-units) and translated items.
3) The translated full text will have the sub-units somewhere in the text. To locate the sub-unit translation in full text translation, use a fuzzy search algorithm.
4) The fuzzy search algorithm will return the start position of match and length of match. We map the embedded tag from the source HTML to that range.
5) The fuzzy match involves calculating the edit distance between words in translated full text and translated sub-unit. It is not strings being searched, but n-grams[1] with n=number of words in sub-unit. Each word in n-gram will be matched independently.

To understand this, let's try this algorithm in a UI string. Translating the English sentence " Please click Start button to run the command sequence " to French: The plain text version is "Please click Start button to run the command sequence" and the subsequence with annotation is "Start". We give both the full text and subsequence to MT. The full text translation is "S'il vous plaît cliquer sur Démarrer pour exécuter la séquence de commande" and the word Start is translated as "Démarrer". We do a search for "Démarrer" in the full text translation. The search will be successful and the tag will be applied,

[1] An n-gram model is a type of probabilistic language model for predicting the next item in such a sequence in the form of a (n −1) − order Markov model. [2]

resulting <p> S'il vous plaît cliquer sur < span> Démarrer pour exécuter la séquence de commande < /p>. The search performed in this example is a plain text exact search.

4.2. Pseudo Module

This module provides dynamic Pseudo tag with content of a different length, from 20–80 percent, as chosen by the user. One type of pseudo is ASCII characters, the other one is super string with some typical Unicode characters.

```
superString: ["表","ポ","あ","A","中","Ꭾ","é","鷗","停","B","逍","Ü","ß","à","ù","ª","ñ"]
```

4.3. MT Adapter

The MT adapter wraps the MT service, for example if the user configures MT as Bing Translator, then MT adapter will automatically translate each piece of souce contents by Bing Translator engine after source DOM tree was processed by parser.

About call mode, the MT adapter uses Ajax to asynchronously call Machine Translation service. Then depending on the different types of Machine Translation engines, the tool uses DOM parser to do corresponding post-editing on the result from the MT engine.

5. Experiments

We compared results from two different methods : One running with a pseudo tag and the other with machine translation. We utilized an extensive series of languages in the pilot project to compare the hit rate (showed in Table 1, Table 2). For the MT engine, in the first environment we used Microsoft Translator Hub. We also ran on the other MT engine like Google translate. We compare results with different scenarios on VMware Product 3.

5.1. Result and Discussion

Table 1 shows the result of using a pseudo tag to increase the string length by 30% to detect potential layout issues. Table 2 show the result of using MT. Results show that using MT to detect potential layout issues results in a higher hit rate than using a pseudo tag.

Language	Detected By Pseudo	Real	Effective	Hit Rate (%)
German	50	41	34	68
French	58	40	33	56
Russian	63	52	29	46
Spanish	59	53	37	62

Table 1. Hit Rate in Pseudo Model Analysis

Language	Detected By MT	Real	Effective	Hit Rate (%)
German	45	42	41	91
French	43	40	37	86
Russian	50	52	40	80
Spanish	49	53	40	82

Table 2. Hit Rate in MT Model Analysis

Note that :

Hit Rate = Effective number / Detected number

6. Conclusion

In this paper, we discussed the purposes of developing our browser extension, the Web App UI Layout Sniffer. It fundamentally prevents layout bugs in early UI design phases, saves time and testing costs, and reduces layout bug fixing during the i18n and l10n testing process. Ultimately it helps accelerate product time-to-market. This extension adopts Machine Translation to replace an English string so the UI designer can identify potential layout issues from the user prospective. It dramatically streamlines product globalization. Per our testing and pilot, it is effective in detecting UI layout issues for VMware web applications and it can be considered for production use for designer and developers.

References

[1] Fighting Layout Bugs: https://code.google.com/p/fighting-layout-bugs/

[2] n-gram: https://en.wikipedia.org/wiki/N-gram

[3] Pseudo Localization: https://en.wikipedia.org/wiki/Pseudolocalization