Automatic Internationalization for Just In Time Localization of Web-based User Interfaces

LUIS A. LEIVA and VICENT ALABAU, Universitat Politècnica de València

The need to modify an application so that it can support different languages and cultural settings can appear once the application is finished and even in the market. This may introduce serious time delays and an increase in costs. We solve this problem for web-based software through JITL, a *post-hoc* method to automatically internationalize websites and web-based applications, without having to modify the source code. With JITL, users can pull resource strings out of an arbitrary website and perform on-demand localization tasks. Based on this novel capability, JITL enables a complete infrastructure for collecting, storing, sharing, and delivering website translations, which invokes a number of exciting scenarios. Our studies show that JITL leads to significant savings in terms of user effort and, in consequence, money. With JITL, now it is possible to localize what is needed, when it is needed.

Categories and Subject Descriptors: H.5.2 [**User Interfaces**]: Prototyping; H.5.3 [**Group and Organization Interfaces**]: Web-based interaction; D.2.2 [**Design Tools and Techniques**]: User interfaces

General Terms: Human Factors, Languages, Design

Additional Key Words and Phrases: JIT; Internationalization; i18n; Localization; L10n; Translation

1. INTRODUCTION

In order to support a multilingual audience, applications must follow a sequential and often iterative process based on two equally important levels: first internationalization (i18n), then localization (L10n). Internationalization consists in decoupling translatable text out of the application source code, basically by wrapping each message or "resource string" with a translation-capable function. After internationalization, the application is ready to support the requirements of different locales, i.e., specific languages and countries of the target audience. Then, software localization focuses on reflecting the conventions of said target audience, by operating on two sub-levels [Hsieh et al. 2008; Sun 2001; Yeo 2001]:

(1) Language *translation*, including jargon, technical terms, etc.
(2) Aesthetic *adaptation*, including images, colors, branding, etc.

Most companies are well aware of both sub-levels but often opt for the former due to time and budget limitations [Abraham 2009; Esselink 2000; Esselink 2003; Hogan et al. 2004; McKethan and White 2005]. After all, while it is true that aesthetic adap-

This work is part of the Valorization and I+D+i Resources program of VLC/CAMPUS and has been funded by the Spanish MECD as part of the International Excellence Campus program. This work is also supported by the 7th Framework Program of the European Commision (FP7/2007-13) under grant agreements 287576 (CASMACAT) and 600707 (tranScriptorium).
Authors' address: L. A. Leiva (corresponding author) and V. Alabau, PRHLT Research Center, Departament de Sistemes Informàtics i Computació, Universitat Politècnica de València. Camí de Vera, s/n – 46022 València, Spain; corresponding author's email: llt@acm.org.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

tation can enhance the user experience [Reinecke and Bernstein 2011], an application cannot be considered as being localized if it does not speak the language of its users.

In short, an internationalized product needs to be localized to a specific audience. However, localization cannot take place without internationalization. Furthermore, there are currently no proven techniques for automated approaches for software internationalization. Our work is aimed at tackling this challenge.

In this article, we focus on web-based UIs, motivated by the fact that today people use web browsers more than any other class of desktop software on a daily basis. Moreover, multilingual websites and web-based applications, much like any other type of software, are crucial to almost every player in the industry [Hogan et al. 2004; Sun 2001]. Thus, as businesses continue to globalize, localizing web-based UIs becomes more compelling. Last but not least, localization is a unique opportunity of preserving a language [Keniston 1997; Leiva and Alabau 2012], which puts forward its notable importance on culture and society.

1.1. Motivation: Lowering i18n/L10n Costs

Software internationalization often occurs during the earliest life-cycle phases of an application [Hogan et al. 2004; Luong et al. 1995]. However, the need to adapt software to support different languages and cultural settings can appear once the application is finished [Wang et al. 2009] and even in the market [Cardeñosa et al. 2006; Troyer and Casteleyn 2004]. This may introduce serious time delays and an increase in costs, which go far beyond just engineering. For instance, big companies typically invest $2M and 12–18 months of their engineering resources in internationalization and delivery of the first foreign language [SurveyMonkey 2013]. Then, fully localizing a software product for one additional language can add up to $100K, whereas Microsoft estimates its costs are $300K or more per product [Collins 2002]. This can be a tall order for small software companies, but also for big companies to target languages with low strategic value in terms of market share.

We believe that, for web-based software, it is possible to significantly lower these costs, inspired by the following principle. In industrial design, Just In Time (JIT) production [Ohno 1988] is about having the right material, at the right time, at the right place, and in the exact amount. Similarly, our method aims to minimize assets (manpower, effort, time, and, in consequence, money) by only localizing what is required, when it is required. We have named our method 'JIT Localization' (JITL) for obvious reasons.

1.2. The JITL Concept

Before we get into the details of our technology, it is worth mentioning the basics of current software localization workflows, as they provided us with a number of design notions and ideas for improvement.

The localization workflow is usually convoluted and communication-intensive overall [Muntés-Mulero et al. 2012; Patel et al. 2009], as it may involve many thousand people scattered in virtually every country of the world [Huang and Trauth 2007; Keniston 1997]. The traditional localization workflow can be summarized in the following iterative steps, see Figure 1:

(1) Internationalization, in order to extract resource strings from the UI. Software developers first generate a message catalog file by extracting resource strings from the UI. The most popular format for message catalogs in open source software is the Portable Object (PO), which is basically a collection of resource strings together with their eventual translations.

(2) Translation of resource strings. This is typically done by editing the message catalog file offline, although there are online tools to support this task.
(3) Text insertion of the translated resource strings into the UI. The translated message catalog is compiled by the developer and placed into a special directory so that the UI can fetch the localized strings at runtime.
(4) Evidence sampling. The developer takes some screenshots of the localized UI, so that translators can validate the translations in context.
(5) Quality control, based on the aforementioned evidence sampling. This process is highly iterative and usually highly resource-consuming, because any string with a translation error must be re-translated and re-inserted into the UI.
(6) After quality control, a validated message catalog is generated. The developer inserts again all translated resource strings into the UI, as indicated in step (3).
(7) Finally, the UI can be tested with native speakers of the target language.

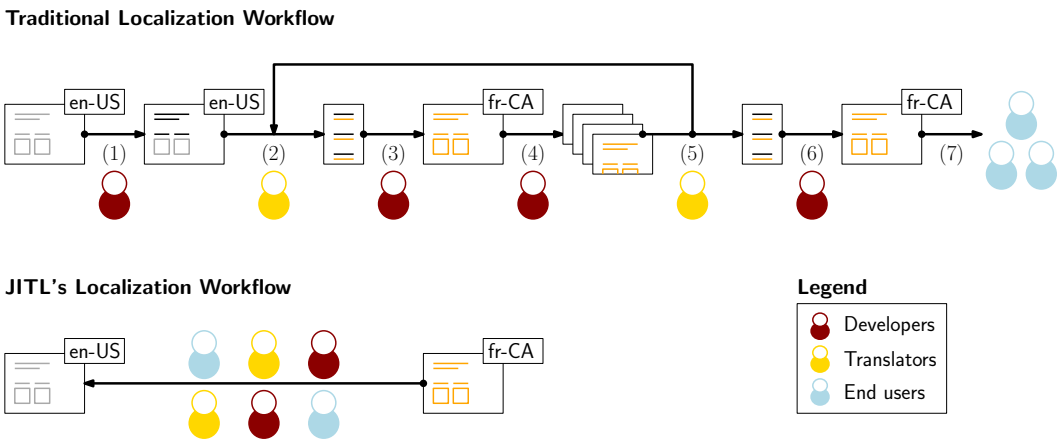


Fig. 1: In current localization workflows, resources are pushed by a company. With JITL, resources are automatically pulled out of the UI.

For most companies, translation is often outsourced to a translation agency. Thus, at least steps (2) and (5) are performed outside the facilities of the company. This typically hinders communication and introduces time delays. Since translations become decoupled from the UI, if an error happens at an early stage, then it is likely to be propagated to later stages, forcing thus a new iteration loop that will probably cost the company both time and money.

In contrast, our approach follows the JIT philosophy, and thus is aimed at localizing what is needed, when it is needed, in order to save time, costs, and user effort. JITL wants to bring together *developers*, *translators*, and *end-users* with the aim of integrating all parties into the whole localization process, placing a special emphasis on the end-users. After all, there is no better representative user sample than the actual users of a website or a web-based application. In sum, all of these target groups can potentially contribute to localizing web-based software, either by submitting initial translations or by providing an alternate translation for a particular widget, so that these are better tailored to a specific locale or country region. The roles of these groups, however, do change in JITL with respect to the traditional localization workflow.

First, *developers* are mostly interested in internationalizing the UI, since it is often their sole responsibility within the software company. They usually perform an important re-engineering effort, to ensure later UI localization, by decoupling all translatable strings from the source code. This is by far the most complex and expensive part of the traditional localization workflow. JITL alleviates this problem since now developers do not need to worry about internationalization, as it is performed automatically on the client side. As a result, this allows them to concentrate on website development. However, if developers are not involved in internationalization anymore, how can strings be traced from UI to code? Actually, JITL manages localization on the client side, so backtracing to code is not necessary. In fact, JITL has no access to the source code on the server side. Thus, if backtracing is a must (e.g., the website owner wants full control over which strings can be localized) then some manual intervention is inevitable. Even so, JITL can be very beneficial to improving both localization quality and coverage, by allowing regular users to contribute.

Second, *end-users* are mostly interested in using the application or the website, as ultimately they will consume its contents. Moreover, end-users are implicitly trained in the specific software, as they often make intensive use of it. Consequently, we argue that end-users are best suited to judge if a particular UI localization is correct or not. For instance, if a particular translation of a menu entry does not reflect what the menu entry actually does, then it is expected that end-users would notice it immediately. Hence, JITL allows end-users to gain some control over the localization process, by being able to translate and correct errors as they go. This way, the texts shown on the UI can fit with the intended use of the website.

Finally, *translators* are mostly interested in localizing the UI, since they are often hired by the software company as external workers. Usually, translators are qualified editors and technically proficient in the context of the software being localized. Therefore, their work is essential (and central) to the traditional localization workflow. Nevertheless, they are not necessarily required to localize a website with JITL; such work is actually surrogated to the end-users. However, at some point the website owner could realize that having the site professionally translated into a particular locale is beneficial; e.g., because visits from that locale have increased substantially and are generating an important revenue for the company. But, how can the owner know that the website is fully localized? And more important, is it correctly localized? In principle, these questions cannot be answered by the end-users, as they are expected to localize the site by following the JIT philosophy, i.e., what is needed, when it is needed. Therefore, we consider translators to be a last resort for achieving a professional localization. Apparently, this will come at a lower cost, since the website would have been partially localized by casual users and volunteers.[1] Thus, instead of having to navigate through all of the pages on the website, JITL allows translators to automatically generate localization files by collecting all the strings on the web pages the users have visited. Furthermore, JITL provides pointers to UI elements so that the visual context is always available to the translator (cf. Section 4.2).

1.3. Contributions

JITL has the critical advantage over current approaches to web-based localization that UIs do not need to be internationalized: resources are automatically pulled out of the UI, instead of being pushed by a company or a developer. This means that it provides end-users with a means to localize any website to their specific locale. This novel capa-

[1] It is important to remark that users' contributions are not immediately applied to the website; they are only visible to the user until she decides to contribute. Then, the website owner decides when should contributions be made visible to the public. See Section 3.10.

bility enables a complete infrastructure for collecting, storing, sharing, and delivering UI translations, which invokes a number of exciting scenarios. In this article, we make the following contributions:

— A detailed description of the technology behind JITL, so that others can build similar systems upon our work.
— A number of applications that illustrate a range of complexity in developing with JITL.
— Evaluation of JITL's automatic internationalization capability, showing that it can save a significant amount of manual work for developers.
— Analysis of JITL's coverage on the top 25 websites according to the Alexa ranking, showing that our method can be applied safely and successfully in complex pages.
— A user study of collaborative in-place UI localization, followed by a number of analyses which suggest that JITL transforms the localization process in a naturally easy task.

1.4. Organization

The remainder of this article is organized as follows. Section 2 discusses related work. Section 3 describes JITL's implementation details. Section 4 provides a number of deployment possibilities, together with a number of already-implemented applications. Section 5 evaluates JITL's capabilities from different perspectives. Section 6 discusses the implications and limitations of JITL. Finally, Section 7 gives a number of concluding remarks and provides opportunities for future work.

A Note for Practitioners. We believe this article will be useful for both practitioners and researchers. However, practitioners are likely to only care about a small portion of the article: the details of the implementation (Section 3) and the deployment possibilities (Section 4). We encourage practitioners to read also subsections 5.1 and 5.2, as they provide evaluation results on actual websites. Practitioners may also benefit from reading Section 6, as that section highlights the merits and demerits of JITL.

2. RELATED WORK

Almost every software development kit provides some kind of internationalization possibilities (e.g., Xcode,[2] Qt linguist,[3] or Facebook Translations[4]), though their practicability is still limited to one development platform. For this reason, we discuss here previous work that provides developers with value-added tools to internationalize UIs, not only web-based, beyond an expected basic support.

2.1. Approaches to Automatic UI Internationalization

To the best of our knowledge, there is no generalized approach to automatically perform web-based UI internationalization, even less using a web browser alone. The closest approaches in spirit to ours are a number of desktop-oriented tools, which means that either an IDE or a specific framework is required for these tools to work. Notable examples in this regard are the following.

TranStrL [Wang et al. 2009] is an Eclipse plugin that takes the source code of a Java application and automatically produces a list of need-to-translate strings. Smartling[5] has an Objective-C library that achieves the same effect by adding minimal modifi-

[2] http://developer.apple.com/internationalization/
[3] http://qt.gitorious.org
[4] http://www.facebook.com/?sk=translations
[5] http://smartling.com

cations to the application's source code. Finally, Globalyzer[6] and World Wide Navi[7] provide a suite of desktop tools to analyze, test, and fix internationalization issues in different programming languages.

Taking a different tack, Wang et al. [2010] extended TranStrL to locate need-to-translate strings in web applications. They used context-free grammars and a lexical parser to differentiate strings that belong to HTML tags and strings that are displayed onscreen. Results were promising, however the need to analyze server-side code may render this approach impractical for translators and end-users, who are unlikely to be able to access and manipulate an application's source code.

2.2. Engineering Translatable Interfaces

Hunt [2013] devised a cost-effective method to localize UI strings inspired by crowdsourcing. The method allows end-users to translate the resource strings of the UI using a dedicated tool, which means that the UI must be internationalized beforehand. A more interesting approach is ScreenMatch [Kovacs 2012], a Java system to assist software translators by showing UI screenshots alongside each localizable string.

Tschernuth et al. [2012] aimed to unify translation for several platforms through CATT, a generic and context-aware navigation tool. CATT is based on an interface definition language that creates an abstract description of the UI, and a meta-model that represents both UI content and structure. This idea could be explored further for web-based software, though eventually a deep refactoring would be required to integrate CATT in an already deployed product.

Context-aware UI translation has been a subject of study for a wide number of desktop-based tools; e.g., Passolo,[8] Catalyst,[9] Multilizer,[10] or RCWintrans.[11] However, these tools force developers to code according to particular guidelines and particular programming languages. More importantly, software localizers are forced to switch and use said tools, which may prevent them from contributing. In contrast, JITL can be integrated off the shelf, without having to modify a single line of code.

Dixon et al. [2010; 2011] were able to build a structured representation of a UI by using pixel-based manipulation procedures. This enables a way to reverse-engineer UIs and interpret their content, which can of course be leveraged to automatically translate them. However, an important limitation of this method is that translated strings are painted onto the original interface, so the font size of the translated text must be adjusted to fit in the available region; see e.g. [Dixon et al. 2011, p.2]. Unfortunately, this can render a UI unusable if there is a significant variation in the number of characters between source and target languages—e.g., sentences in French and German are on average 30% longer than English texts [Esselink 2000].

Moreover, what all these tools display as UI is not the actual UI users will operate, just a static visualization. We believe this is rather limited when it comes to internationalizing web-based UIs, since these are inherently *dynamic* and therefore change often over time. The TOCHI website, for example, is mostly static but actually changes content and structure in subtle ways.

[6] `http://lingoport.com/globalyzer`
[7] `http://kokusaika.jp/en/product/wwnavi.html`
[8] `http://www.passolo.com`
[9] `http://www.alchemysoftware.ie`
[10] `http://www2.multilizer.com`
[11] `http://www.schaudin.com`

2.3. Other Approaches to Internationalize Web-based UIs

Some professional translation tools such as TRADOS,[12] or Google Translator Toolkit[13] do support automatic translation of web pages. Translating web pages is also supported natively by some browsers like Google Chrome[14] or SlimBrowser.[15] All these tools can significantly speed up the localization workflow, despite of their low quality in comparison to human-generated translations [Pérez-Quiñones et al. 2005]. However, these automatic tools are oriented to translate *content* like text documents or static HTML files, and hence they do not support a proper internationalization method. For instance, some UI elements such as placeholders or drop-down lists cannot always be localized. We also must emphasize that these tools do not support the JITL concept, as they are just read-only automatic translation applications. For instance, they do not allow users to edit and save the translated texts, contribute with a custom translation for a given UI element and populate it through the website, or export localization files on the fly.

2.4. Making Websites more Accessible

Researchers have observed that users who want access to inaccessible content must ask the site owners for help [Chilana et al. 2012; Kawanaka et al. 2008; Takagi et al. 2008]. This process is slow and too often the need is mooted before the content becomes accessible. As a result, Takagi et al. [2008] and Kawanaka et al. [2008] introduced a web-based annotation tool aimed to drastically reduce the burden on site owners and to shorten the time to provide accessible content by allowing volunteers to insert custom metadata on webpages. Then, site owners can introduce "accessibility renovations" based on the volunteers' metadata. A very similar approach in this vein is the project Wikify[16] (no longer maintained). It was aimed at applying the collaborative wiki content model to any website, by allowing the community to "exchange ideas, update content, fix errors, parody, and improve the Internet as a whole". Another related approach is the AnswerDash project,[17] a "self-service contextual help for websites and web applications". AnswerDash is a spinoff of LemonAid [Chilana et al. 2012], and provides users with a point-and-click support to both retrieve and raise questions as they browse websites. For instance, a user that wants to buy a laptop can go to an online shop and click on the image search results to ask for the laptop case that best fits with the selected laptop image, without having to leave the online shop or having to wait for the customer service to be available.

With these approaches, users can contribute to arbitrary websites. Edits and annotations are stored on a centralized server, where other users can retrieve them, view them, and contribute more. These approaches, although not related to software internationalization, share our vision of how the Web should actually be; i.e., a place where anyone can truly contribute to enhance and reshape its contents.

3. SYSTEM DESCRIPTION

JITL is a novel approach to internationalize web-based UIs. It allows arbitrary users to translate and modify the texts on any web page, in order to improve its localization without requiring explicit owner's consent. In this section we describe our technology, together with a series of design principles that make JITL possible. We believe that

[12] http://www.trados.com
[13] http://translate.google.com/toolkit/
[14] http://www.google.com/chrome/
[15] http://www.slimbrowser.net/
[16] https://code.google.com/p/wikify/
[17] http://www.answerdash.com/

understanding these principles will be useful to researchers trying to build similar tools. Figure 2 depicts the fundamental steps followed by JITL, once the Document Object Model[18] (DOM) has been loaded and is thus ready for manipulation. The system architecture is shown in Figure 3.

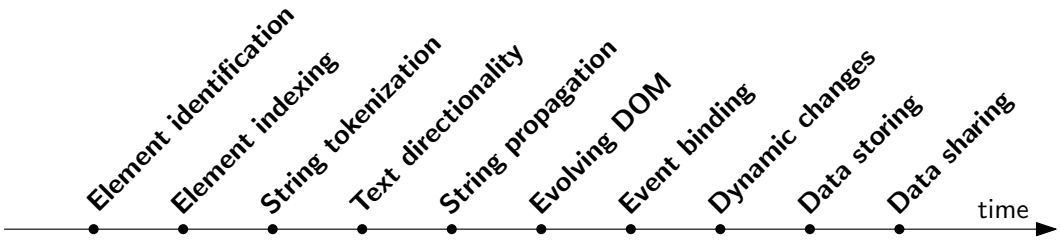


Fig. 2: JITL as explained in the next subsections.

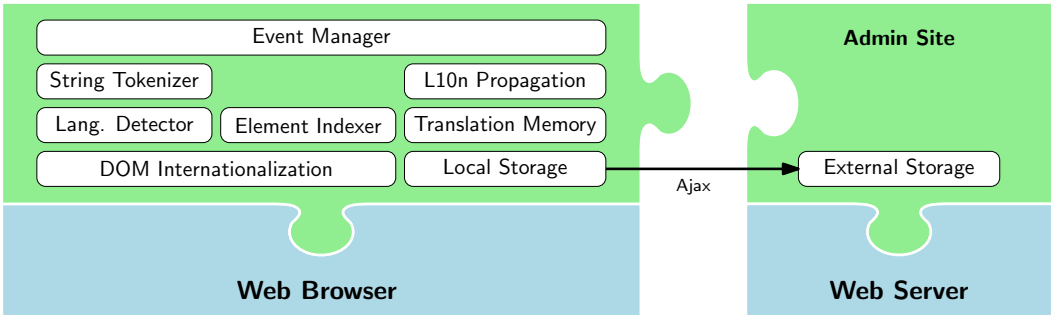


Fig. 3: JITL system architecture. A web server centralizes all user contributions.

3.1. Identification of Localizable Elements

First of all, we should identify every text node in the DOM that can be internationalized and thus localized. However, there are text nodes that must be left untouched. For instance, HTML comments and text inside `<script>`, `<code>`, or `<textarea>` elements should not be localized. Therefore, a per-element node analysis becomes necessary. Currently, there are three methods for achieving this: (1) recursive DOM tree traversal, (2) plain XPath selection, and (3) TreeWalker iterator. Among these, we decided to implement TreeWalker, as it has been shown to outperform all other methods, sometimes with more than one order of magnitude in runtime performance; see [Steiner et al. 2012, fig.2]. TreeWalker is supported by all major browsers, and enables an easy way to filter nodes through the DOM. Concretely, JITL uses TreeWalker to recursively descend through the DOM and process node leaves, which are the nodes that contain text, ensuring that the parent node is eligible for internationalization.

[18]The DOM defines an internal representation of web pages based on a node hierarchy.

3.2. Indexing UI Elements

Additionally, we also need a way to uniquely identify *all* UI elements on a page, in a manner that also lets us correctly identify the same elements on other web pages in the same web site. In web-based software, the DOM can be queried using XPath expressions, which represent the sequence of nodes and their indices from the root node down to a particular node. Therefore, using XPath alone should suffice to point to arbitrary DOM elements. However, given that internationalization must be carried out in a per-domain (instead of a per-page) basis, the same XPath may have different source texts from page to page on the same website. For that reason, we decided to index UI elements by a hash consisting of XPath plus source text. This proved to be a solid approach to begin with, although a number of corner cases remained to be resolved. We discuss them in the next subsections.

3.3. String Tokenization

Resource strings are then extracted from the localizable elements and tokenized to support the so-called "format strings" macros, i.e., using `%d` to indicate a digit (e.g., `"%d items"`) or `%s` to indicate string substitution for those messages that can be identified by means of regular expressions. JITL can automatically detect digits, URLs, emails, hashtags (strings starting with `#`), and mentions (strings starting with `@` as in Twitter or starting with `+` as in Google Plus). Eventually, it is possible to add any other cases as long as they can be expressed by means of a regular expression.

String tokenization facilitates propagating localization information to other UI elements that share the same source text tokenization, avoiding thus having to localize near-duplicate strings (c.f., `"5 items"` and `"12 items"`). Then, the text that is rendered on UI elements comprises untokenized strings, to make localization easier for non-experienced users; i.e., without being aware of this tokenization process.

Sometimes it may be necessary to tell the difference from a templatizable digit and a digit just used in a string (e.g., `"Step 1:"`). While both cases are internally handled in the same way, JITL can actually tell the difference between them both. For instance, if a user translates `"Step 1:"` to `"Paso A:"`, then the translated text would not be tokenized (as it has no digit) and so this translation would not be propagated to other elements with the same source text; e.g., an element with `"Step 2:"` would be left untouched.

We assume that texts consisting on their entirety of format strings macros are not eligible for human localization, as they do not contain user-generated content and so translators would not be able to produce any meaningful localization. For instance, the element `<tt>email@address.com</tt>` would be automatically internationalized as `"%s"` so it should not be considered for human localization.

3.4. Checking Text Directionality

Different page layouts are often required for right-to-left (RTL) languages, as most RTL languages should be right-aligned rather than left-aligned. By inspecting previously translated strings, JITL modifies the text direction accordingly; e.g., if the user is localizing into Arabic, the text orientation of UI elements is set to RTL so that the text flows right-to-left. In addition, the UI is adapted for these RTL languages, essentially mirroring the layout of the default UI language (Figure 4). This is accomplished by adding a `dir="rtl"` attribute to the `<body>` element. More advanced techniques might be devised in future work for handling unexpected corner cases, but so far this approach has proved to work well. The most interesting part here is that JITL can tell RTL and LTR languages apart without prior training or using a language model, just

by means of a single unicode-capable regular expression. Therefore, it is a very scalable approach.

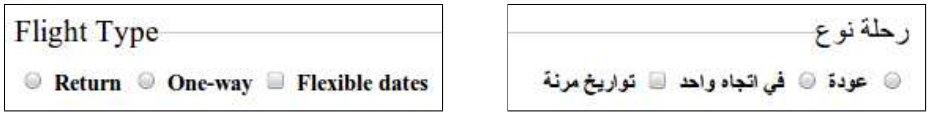


Fig. 4: JITL changes the text direction and UI layout according to the target language the user is localizing into.

3.5. Loading and Propagating Translations

With JITL, localized content can be loaded either from previous browsing sessions or from a centralized server. A recurrent problem we have observed is that the same source text may appear in a different DOM position on the same page and even in another UI element on another page. Also, many websites reuse the same words and phrases on many different pages. To avoid having to translate each string separately, JITL incorporates the following propagation mechanism.

In its most basic instantiation, source-target localization pairs are used to build a translation memory, regardless the DOM information, which can be seen as an internal "dictionary" that allows non-localized elements to be adequately initialized. Moreover, since the translation memory is built as the UI gets localized, subsequent localization updates can be successfully propagated to other UI elements on the website. Thus, the same source text does not have to be translated more than once, unless it should have two different translations. This corner case is illustrated in Figure 11.

3.6. The Evolving DOM Problem

Looking under the hood of many modern websites, it is not uncommon that their DOM changes over time, even if the visual appearance is not altered dramatically. This may be problematic if two (or more) previously localized elements having the *same source* text but *different translations* are now in a different DOM position, because in principle it would be necessary to translate again those "orphan" elements (i.e., previously translated elements that are no longer located in the DOM). However, it is also true that the updates performed to the DOM are not dramatically substantial. For instance, common DOM updating operations are *wrapping* (e.g., a button is surrounded by a new `<div>` element) or *insertions* (e.g., a new paragraph is added to the DOM, either well before the previously localized element or far away from it). Thus, to keep JITL translations in the long term as the website evolves, we have incorporated an efficient recovery mechanism of localized content (Figure 5).

When there is an orphan element, JITL first normalizes its XPath as well as the XPath of all nodes that have the same source text. This normalization process rewrites their XPaths using a more compact, symbolic representation. Next, the Damerau-Levenshtein distance from the orphan XPath to the candidate XPaths is computed, operating over the aforementioned symbolic XPath representations. This allows JITL to successfully perform fuzzy matches, even when the original element (e.g. `<button>`) is replaced by another element (e.g. `<input>`). When the closest node is found, it is assigned the previously localized string and JITL's internal indexing structure is accordingly updated, replacing the old XPath by the new XPath.

Originally localized elements, now orphan	**Symbolic XPath**
`/html/body/p`	`ABC`
`/html/body/a`	`ABE`

Non-localized elements, with same source text but different translations	**Symbolic XPath**
`/html/body/div/a`	`ABDE`
`/html/body/section/p`	`ABFC`

If we rely on the original XPaths, `/html/body/div/a` would be assigned the translation of `/html/body/p` and `/html/body/section/p` will assigned the translation of `/html/body/a`, since:
$d(\texttt{/html/body/p}, \texttt{/html/body/div/a}) < d(\texttt{/html/body/p}, \texttt{/html/body/section/p})$ and
$d(\texttt{/html/body/a}, \texttt{/html/body/div/a}) < d(\texttt{/html/body/a}, \texttt{/html/body/section/p})$.

In contrast, using the Symbolic XPaths yields the correct assignments, i.e., `/html/body/div/a` is assigned the translation of `/html/body/a` and `/html/body/section/p` is assigned the translation of `/html/body/p`, since $d(\texttt{ABC}, \texttt{ABDE}) < d(\texttt{ABC}, \texttt{ABFC})$ and $d(\texttt{ABE}, \texttt{ABDE}) < d(\texttt{ABE}, \texttt{ABFC})$.

Fig. 5: Recovery mechanism for localized content on a web page.

3.7. Binding Interaction Events

Ultimately, event listeners are attached via `document.addEventListener` so that JITL can receive user interaction. This mainly allows the user to localize the UI on demand, but it also enables other applications, such as supervising the internationalization result (see Section 4.1) or amending localization errors as soon as they are spotted. Attaching events only once to the `document` element instead of to every UI element precludes any performance hit. Further, the `addEventListener` method avoids to conflict with existing event handlers on a web page.

Currently JITL listens to mouse hovering (to highlight elements), clicks (to edit the i18n/L10n of an element), and key strokes (to save editing changes). These actions only take place if a special key is pressed (currently the Control key), in order to allow for regular interaction with the UI otherwise. It is worth noting that browsers completely ignore interaction on form fields that have a `disabled` attribute. To solve this problem, disabled elements are automatically wrapped in a proxy node that can detect user interaction, thus adequately forwarding event data.

3.8. Tracking Dynamic DOM changes

Dynamic web applications frequently introduce changes to the DOM, e.g., to inform the user about some error or an update. Also, potentially translatable resources are sometimes not just text, but a regular expression with text and other fields that are filled in at runtime. JITL addresses this job by monitoring the `MutationObserver` events, which make it possible to look for all dynamic modifications in the DOM, including e.g. node subtrees, attributes, and character data. When this occurs, the modified elements are internationalized from scratch and localized, if there is information available to do so. JITL provides fallback events such as `DOMNodeInserted` or `DOMSubtreeModified` for older browsers that do not understand `MutationObserver` events.

3.9. Storing i18n/L10n Data

Of special importance is deciding how localization resources and other user-generated metadata should be stored. Because JITL can target websites that are not under the developer's control—e.g., one of the most interesting uses of JITL is on-demand localization; see Section 4—the data should be stored in principle on the client side. This also avoids hitting a remote server (if any) frequently, and thus only submitting the data to a centralized server when it is really needed; see Section 3.10. Currently, there

are three competing approaches for saving serious amounts of data (i.e., persistently and bigger than cookies) locally in the browser: (1) Web Storage (2) Indexed Database, and (3) Web SQL Database. Among these, we decided to use Web Storage because it has a simple API and is already available in all major browsers.

Conversely, when JITL targets the application developer, it features an additional storage mechanism on the server side, in plain JSON files. This is interesting to save internationalization data whenever a page is requested, which can be used to generate a localization file for offline edition, as currently held by professional translators; see Section 4.

3.10. Sharing Localization Data

JITL makes it possible to build a number of exciting applications, some of them relying on external workers or arbitrary end-users. For these cases, all translated strings and metadata (e.g., user agent string, browser's locale, geolocation) should be submitted to a centralized repository.

Also, it is important to decide how localization data should be submitted to the repository. For example, is it after explicit user consent, or whenever n elements have been localized, or some other criteria? We opted for the former method, since it allows users to review their contributions before deciding when it is best to submit. This way, a full website or web application can be incrementally and collaboratively localized. Of course, ultimately the website's owner must decide whether to make all users' contributions visible or not. A public website that is partially localized may cause companies to lose potential customers.

4. DEPLOYMENT AND APPLICATIONS

JITL has three different instantiations, each one aimed at fulfilling a different purpose on the basis of its potential usage.

For developers. JITL is basically a collection of small JavaScript libraries, and so developers can easily incorporate JITL to their websites so that it can be automatically internationalized or to allow others to contribute to localizing it. A developer just would add a `<script>` element to those pages that should be automatically internationalized. In practice, a website has a header or a footer part that is common to all of the pages on the site. Therefore, the developer can place the `<script>` element in any of these common parts, allowing thus the whole website to be automatically internationalized over time. In this case, localization data are stored on the server side, so they are readily available for offline localization; c.f. Section 4.2. Moreover, by means of a special code, the developer can retrieve localized data from a centralized server. This allows the website to be automatically translated with the strings that have more consensus among the community.

For translators. Software translators are often interested in localizing a company's website only once, who then submit their translations to the company for review. But it could happen that JITL has not been enabled on the website. So, we provide a simple one-click tool to add internationalization and localization functionality to the browser by means of a bookmarklet. Bookmarklets are snippets of JavaScript code that can be bookmarked, i.e., saved as a favorite, and are browser-independent. A translator would install the bookmarklet from the JITL project page,[19] by simply dragging a link to the bookmarks bar of the browser (Figure 6a). Moreover, the translator can generate

[19] `http://personales.upv.es/luileito/jitl/`

a localization file for offline processing. This makes it possible to work with a preferred editor or even other localization tools.

(a) (b)

Fig. 6: Both the bookmarklet (a) and the browser extension (b) inject a single line of JavaScript code to the DOM so that the required libraries are automatically loaded.

For end-users. Arbitrary users may also be interested in localizing some parts of the UI; for instance, because everything but one widget is speaking their native tongue, or because there is some text that bothers them; see e.g. the Twitter's "Who to follow" issue.[20] Then, to support these usage cases it is desirable to 1) automatically allow users to localize the website on demand; and 2) detect that the website should be automatically localized to users if they had previously done so. To achieve this, an end-user would install JITL as a browser extension (Figure 6b). Extensions adds functionality to the browser without diving deeply into native code. The JITL extension enables a dedicated menu entry on the browser by which the user can localize any web-based UI on demand. In order to avoid writing specific code for each browser, we have tapped into Kango,[21] which provides cross-browser extension deployment.

JITL enables a set of applications that until now have been either not supported or only available for other platforms or programming languages. Because of this, we validate and provide insight into our work through a number of scenarios that illustrate a range of complexity in developing applications with JITL. We confirm that the following applications are implemented and work on all major browsers.

4.1. Interactive Internationalization

Due to a design decision, only leaf DOM nodes are automatically internationalized, since leaf nodes are the ones that contain text. As we will show in Section 5.1, this may provide an internationalization result that is not perfectly aligned to what a professional developer would produce. In consequence, JITL can operate in "interactive i18n mode", by allowing developers to intervene in the internationalization process.

Furthermore, in a professional localization scenario, some strings should not be translated, such as company names, catch phrases, and so on. By means of this interactive internationalization application, it is possible to indicate which elements should be localized by simply CTRL+clicking on them, and which elements should not be localized by CTRL+SHIFT+clicking. On the other hand, if a resource string should be made up of two or more leaf nodes, it is possible to indicate so by CTRL+selecting them (Figure 7). This way, the string will be assembled using the text of the closest common ancestor of the selected leaf nodes.

[20] `http://stancarey.wordpress.com/2012/04/05/who-to-follow-is-grammatically-fine/`
[21] `http://kangoextensions.com`

Fig. 7: JITL supports the aggregation of different DOM nodes into the same resource string, provided that those nodes share a common direct ancestor.

On the other hand, since JITL can generate a PO file on the fly, strings that cannot be localized at runtime can be eventually localized offline. This may include messages that appear for a short amount of time (e.g. `"Loading..."`) or elements that display additional text on hover events (e.g., tooltips). An experimental procedure to handle these cases is shown in Figure 12.

Once the desired UI elements have been internationalized, the remaining elements are internally marked with a `jitl-nonlocalizable` data attribute. In the end, a UI internationalized this way is eventually supervised by a human and of high quality. To our knowledge, JITL is the first approach that provides this capability.

4.2. Automatic Generation of Localization Files

After internationalization, JITL already knows which resource strings are available for translation. Therefore, one immediate application is the on-the-fly generation of a localization file. Some well-known formats are PO, TMX, and XLIFF. Currently JITL exports to PO (Portable Object) format, since it is widely used at present, although localization data are sufficiently abstracted in the web storage (as JSON files) and thus it should be easy to export to other localization formats.

In short, PO files are collections of source (msgid) and target (msgstr) entries in plain text.[22] In the traditional localization workflow, the PO file is delivered to the translator, who translates all PO entries. Developers then produce an MO file (Machine Object, binary code) and place it into a special directory so that the application can fetch the appropriate locale for the resource strings. When JITL is installed on the server side (see Section 4), it can also generate PO files, so they would be immediately available to the translator. Conversely, when JITL is invoked as a bookmarklet or as a browser extension, a dedicated menu entry makes it possible to download the PO file.

Moreover, JITL enhances the utility of these automatically generated PO files by augmenting each msgid with a link to the element where such msgid belongs to. The link actually points to the element's page URL, together with a special URL parameter that indicates the element's XPath (Figure 8). By signaling such a special URL parameter, JITL loads a highlighting module so that the translator can visually notice where is exactly located each resource string on the UI. If the resource string appears in more than one element, only the first element is shown, since the purpose of the highlighting module is to provide translators with an example of the element's visual context. Nevertheless, this could be easily adapted to support highlighting multiple elements that share the same msgid. Finally, it is worth mentioning that when dynamic DOM changes are detected, the corresponding source-target entries are automatically appended to the PO file. Hidden elements cannot be highlighted unless the user triggers first the action that reveals those elements.

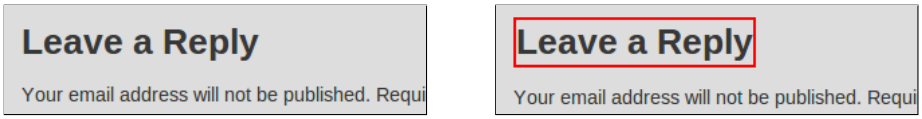
4.3. Real-time Localization of UIs

JITL was designed to perform localization manually, without any machine translation engine. However, we have observed that major search engines are able to translate

[22] `http://www.gnu.org/software/gettext/manual/`

```
#sample: http://localhost/app.php?xvis=%2Fhtml%2Fbody%2Fdiv%2Fdiv%2Fh2
msgid "Leave a Reply"
msgstr ""

#sample: http://localhost/app.php?xvis=%2Fhtml%2Fbody%2Fdiv%2Fdiv%2Fp%5B1%5D%2
msgid "Your email address will not be published."
msgstr ""
```



(a) Regular UI view (b) Highlighted element

Fig. 8: Automatic PO generation augmented with visual context hints. Each resource string is attached a URL that provides translators with an example of the element containing such string.

web content in real time, though with some limitations, c.f. Section 2. Therefore, we decided to integrate this capability into JITL in order to match industry's standards.

Tapping into the automatically extracted text from the DOM, JITL can submit all resource strings to a number of online translation services and present thus the translated content on the original UI in a transparent way for the user. This way, users can browse the site in their native language when it is not natively supported by the site.

Currently JITL can interface with Apertium,[23] an open source machine translation platform, although the communication API is ready to be extended to other translation services, even to custom machine translation engines like Moses.[24] The latter integration is especially interesting since it would allow building systems tailored to translate UI strings, which is a very challenging task because machine translation engines do not deal with the (structural) context of the UI [Muntés-Mulero et al. 2012].

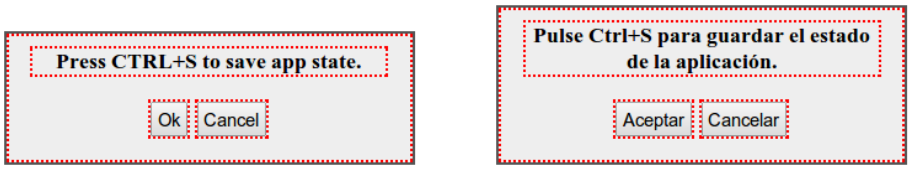
Another convenient feature derived from real-time translation is that, since a web-based UIs is rendered on a browser, translated strings can naturally take up as much space as needed (Figure 9). It is because browsers use a flow-based layout model,[25] in contrast to other UIs that are constraint-based, so that the position and geometry of the UI elements are recalculated whenever the DOM content is modified. This is worth considering, as strings usually change size in translation, so if there is not enough space on the UI, some strings could overlap other controls.

Some web applications use fixed positioning for navigation bars or menus. In these cases, automatic reflow is not possible, but JITL allows the user to see the broken wrapped result immediately and try for a shorter translation. For instance, imagine that the `"Ok"` button in Figure 9 had a fixed width. If a Spanish user translates it to `"Aceptar"`, the string would overlap the `"Cancel"` button; in which case she could use an alternate, shorter translation such as `"Sí"`.

4.4. In-place Localization

This is perhaps a radical departure from the traditional model of software localization, although we believe it may gain notable acceptance in the industry. For instance, Facebook and Google+ allow users to request an *inline* translation of post comments. By clicking on a link, the comment is replaced by an automatic translation. Never-

[23] `http://www.apertium.org/`
[24] `http://www.statmt.org/moses/`
[25] `http://www.mozilla.org/newlayout/doc/`

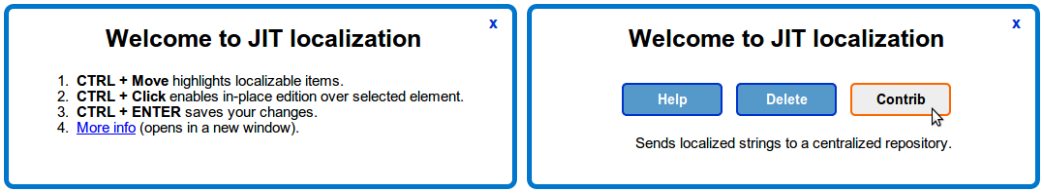


(a) English dialog (b) Spanish translation

Fig. 9: Content reflow after translation, showing in red the redraw regions of the affected elements. Notice that the translation of the `"Ok"` string would overlap the Cancel button if the UI content could not reflow accordingly.

theless, website users are not allowed to correct translations errors or even contribute with another translation, since that could incur in a serious case of impersonation.[26]

In JITL we extend this concept to produce localization data that is properly contextualized, by allowing users to edit or translate texts on the UI at will. We find it especially appropriate to foster contributions from arbitrary users, because of its low entry cost, i.e., there is no need to switch to using a dedicated localization application. For instance, if a user finds an untranslated sentence, a grammatical error, or she thinks of a translation that fits betters her linguistic preferences, she can contribute with a new translation on demand.



(a) (b)

Fig. 10: JITL shows two different menus to the user, one for first-time users (a) and other for returning users (b).

In our current implementation, a welcome screen (Figure 10a) provides the user with the basic instructions to translate the UI in-place. To capture user input, if an element supports the `contentEditable` attribute, JITL allows the user to type right over the element, as shown in Figure 11. For the remainder UI elements, a pop-up dialog is used instead. We should mention that this point-and-click interaction technique is not novel by itself; see Section 2.4. However, it nicely fits in with the JIT philosophy, since it allows to localize only what is needed, when it is needed.

The following particular case is worth of discussion. On dynamic websites, a number of strings appear only for a short amount of time. While it is possible to generate a PO file with all strings that have been shown on the UI for offline translation, we have devised a way to do it online. As shown in Figure 12, the browser extension displays a queue of all seen strings that were removed from the DOM. Each string in the queue is

[26] For instance, if user A writes the text `"I like your site."` and user B rewrites A's text as `"I don't like your site at all."`, then it may cause some trouble for user A, who is the post author but now has been impersonated by user B.

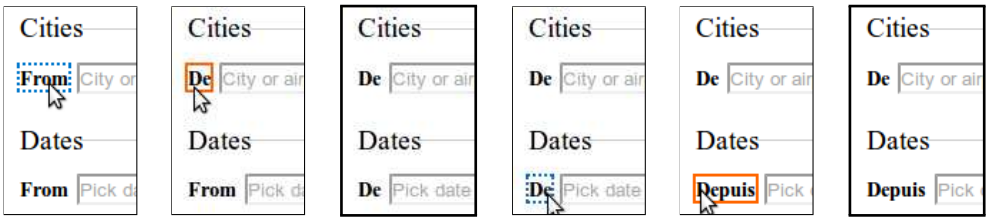


Fig. 11: Localizable UI elements are highlighted when hovered with the mouse cursor (a). Then the user simply clicks on the element to translate it in-place (b). Translations are propagated to other non-localized elements (c). If the same source string should have a different translation (e.g., because of the UI context) the user can proceed (d, e) and the previously localized element will be left unchanged (f).

clickable, in which case a pop-up dialog allows the user to enter the desired translation. By now it is an experimental feature, and may be subject to further changes in future work.

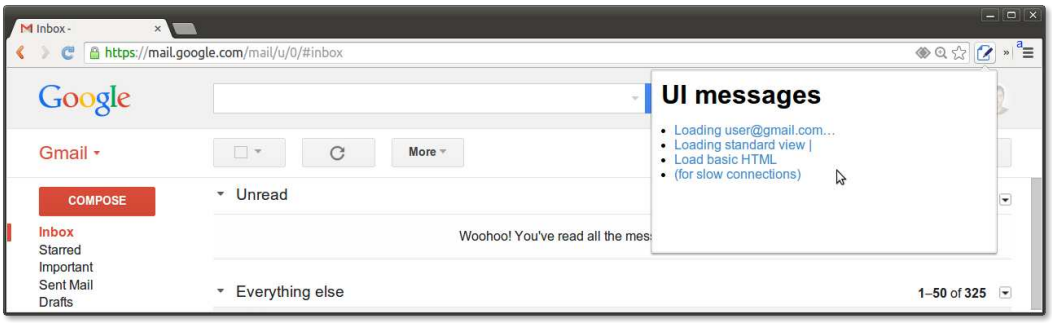
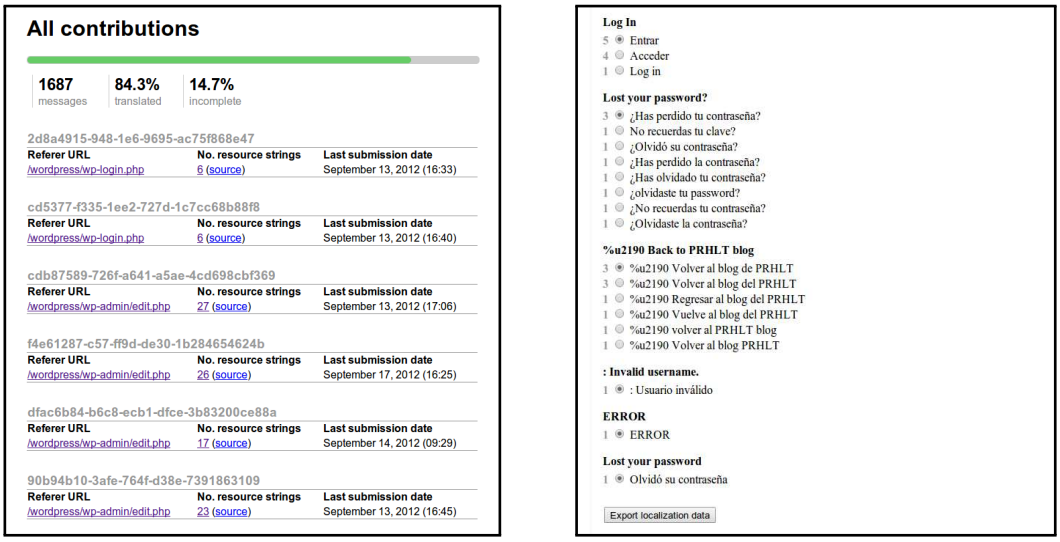


Fig. 12: Envisioned alternate mechanism to process seen but not localized strings. Google and the Google logo are registered trademarks of Google Inc., used with permission.

4.5. Collaborative Localization

JITL allows end-users to contribute with their translations by pressing a 'Contrib' button (Figure 10b). By doing so, all localized items for the currently browsed domain are submitted to a centralized repository (Figure 13a). This repository relies on an infrastructure that should be configured by the owner of the website, or the organization that will incorporate JITL into their localization workflow.

Localized items are then statistically processed to find the most probable agreement level among all submitted strings and to handle variance on translation skills among all contributors. For doing this, we tap into the Fleiss' kappa (κ) [Landis and Koch 1977], which is a well-established measure of agreement rate [Bentivogli et al. 2011; Paul et al. 2012]. Basically, κ discounts the probability of chance agreement, $\Pr(e)$,



(a) (b)

Fig. 13: Localization resources can be submitted to a centralized repository (a) for later postprocessing (b). In our implementation, each contributor is assigned a unique identifier and resources are grouped by domain.

from the observed probability of agreement, $\Pr(a)$:

$$\kappa = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)} \tag{1}$$

The estimation of these probabilities requires the specification of a number of *items* and a number of item *categories*. In this case, such items are the resource strings, whereas the different categories are the translations that can be assigned to each resource string. Unfortunately, it is not possible to anticipate the exact number of *all* possible translations for a given resource string. Hence, only the *observed* translations are considered as categories.[27]

5. EVALUATION

In order to assess whether JITL can achieve its intended goal, we analyzed it from several points of view. This section summarizes three studies performed so far. First, we show the value of JITL as an automatic internationalization tool. Then, we evaluate the impact on runtime performance incurred by JITL. Finally, we perform a controlled in-lab evaluation aimed at assessing how JITL would be used by end-users.

5.1. Automatic Internationalization

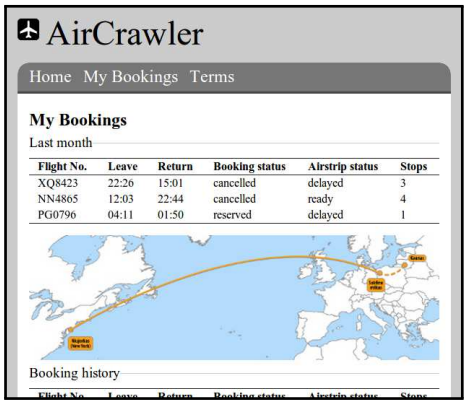
The fundamental feature of JITL resides in the capability of being able to internationalize web-based UIs automatically. Therefore, we felt that an evaluation of this capability was necessary in any case. Concretely, we aimed to answer the following questions:

[27] This is a pessimistic approach, since the number of observed translations is always less or equal than the number of possible translations. Thus, $\Pr(e)$ is overestimated and, in consequence, $\Pr(a)$ is underestimated.

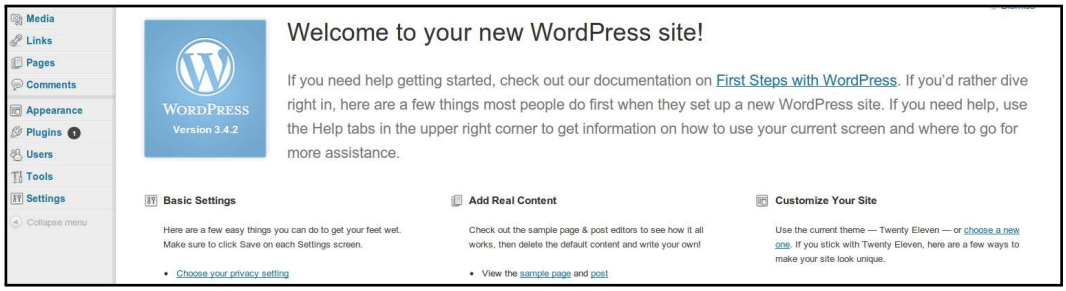
(1) **Time:** How fast is JITL when it comes to internationalizing a website in an automated way? Does it scale well?
(2) **Precision:** Among all translatable strings that were automatically identified, how many of them do match those manually defined?
(3) **Recall:** Among all translatable strings that were manually defined, how many of them were successfully identified by JITL?
(4) **F-measure:** What is the overall accuracy of JITL while retrieving all of the relevant translatable strings?

First, we compared JITL against manually internationalizing a small airline website (Figure 14a), so that we could estimate the overall performance of our method. Then, we repeated the same analysis with Wordpress (Figure 14c), the popular content management system,[28] which is already internationalized and would thus provide us with a unique opportunity to test our method in an actual and a very popular product. All experiments were performed on a GNU/Linux machine, i386 dual core @ 1.3 GHz and 2 GB of RAM.

(a) Airline website, home page.



(b) Airline website, search page.



(c) Wordpress website, dashboard page.

Fig. 14: Screenshots of the analyzed websites. The WordPress name and logo are registered trademarks of the WordPress foundation, used with permission.

[28] `http://www.wordpress.org`

5.1.1. Procedure. Two professional developers that have worked as software localizers were recruited to internationalize the airline website. The website had 5 sections that were dynamically assembled using 10 PHP files. Both developers were told to be as accurate and quick as possible at decoupling all resource strings from the source code, resulting in 32 and 33 minutes each. Then, all identified resource strings were extracted using the `xgettext` command, which is available in all Unix systems. As a result, the corresponding localization files were generated: one PO file per developer, each comprising 92 strings in total.

We noticed a couple of discrepancies after using the `diff` command over both PO files. One developer considered that the site name should be internationalized, because it was inside a visible `<h1>` element. The other developer thought that the copyright text `"© Site 2013 - All rights reserved"` that appeared at the bottom of the website should be internationalized in its entirety, while the other developer thought it should be only the `"All rights reserved"` part. Thus, both developers were told to discuss these discrepancies and reach an agreement. In the end, they decided that the site name should not be internationalized, and that the copyright text should be partially internationalized. Finally, a single PO file was compiled as a result of the manual internationalization, which would be used as ground truth data.

We should mention that Wordpress is already internationalized and even provides an official PO file. Therefore, no manual intervention was performed to the Wordpress site. The official PO file would be used as ground truth data as well.

In order to make results comparable, we decided to follow a common procedure for evaluating JITL on both the airline website and the Wordpress admin site. We included JITL as a `<script>` element and opened all site pages by clicking on all links of the navigation menus (5 pages for the airline website, 37 pages for the Wordpress admin site). Every time a page was requested, we logged the time required to internationalize it and a list of the automatically identified strings. Then, we averaged logging times and compiled a unique list, removing string duplicates.

5.1.2. Results and Discussion. Table I summarizes the performance that JITL achieved. As observed, on average just 11 and 30 ms were required to internationalize each page at the respective websites. It is clear that this is extremely fast in comparison to manual work (c.f., developers spent half an hour internationalizing the airline website), and that it scales really well (c.f., it takes an increment of 20 ms to scale from 92 to 1,687 strings). However, as expected, because JITL performed the internationalization process without human supervision, it may not be perfectly accurate.

Indeed, more resource strings than necessary were automatically extracted from both websites. This is because, in its current state, only leaf DOM nodes are automatically internationalized, i.e., the nodes that contain text. Therefore, if a string in the reference PO file is composed of HTML elements, then JITL would retrieve different strings for each of the leaf DOM nodes. For instance, for the resource string `"Insert your <em>name</em> here"` JITL would generate three different strings: `"Insert your"`, `"name"`, and `"here"`, as these are the leaf DOM nodes of the resulting HTML code. Consequently, the previous example would reduce the recall counts by one item (the missing `"Insert your <em>name</em> here"` string) and the precision counts by three items[29] (the three strings into which the original string would be split). This example is actually considered a bad practice in the localization industry, because it couples markup

[29] Although one may think of clever *ad-hoc* rules to solve this shortcoming, in practice we have observed many corner cases that would invalidate such rules, not adressing the errors but also leading to a different kind of internationalization errors.

structure with resources. However, we observed that it happened in about 10% of the strings in the Wordpress PO file.

Website	No. Strings	Time (ms)	Precision (%)	Recall (%)	F-measure (%)
Airline	92	11 ± 4	75.4	96.7	84.8
Wordpress	1,687	30 ± 17	61.5	67.0	64.0

Table I: Automatic internationalization results for the airline website and Wordpress.

Fortunately, recall shows very promising results. For the airline website, JITL was able to retrieve 96.7% of the strings we were looking for, whereas for Wordpress 67% of the strings were correctly retrieved. It must be reminded that recall accounts for the strings that were internationalized *exactly* as they were manually defined in the reference PO file. Therefore, in the end a lower precision is less important provided that recall is high. This allows developers to better refine the internationalization of the UI (cf. Section 4.1). In principle, it is easier to ignore irrelevant strings than having to manually specify the missing ones. For completeness, we report in Table I the F-measure (the harmonic mean of precision and recall) to illustrate the overall retrieval performance of JITL. It can be observed that automatic internationalization is currently possible.

On another line, JITL also found some strings worth internationalizing that were not included in the reference PO files. This is particularly interesting to detect changes introduced to the UI that may have passed unnoticed. For instance, it may happen that a new widget goes quickly from testing to production on an already internationalized website, the widget being partially internationalized, or not at all.

Overall, these results put forward the fact that JITL can save a significant amount of manual work for developers. For instance, JITL can be used to extract all potentially translatable strings and create a single localization file, then iterate over that file to achieve a high-quality result. Another application to achieve this result consists in operating under an interactive setting, which has been discussed in Section 4.1.

5.2. Tokenization and Interaction

JITL enables in-place localization by injecting JavaScript code on web pages. For this reason, we decided to evaluate the impact of said injected code on page performance. In particular, the goal of this evaluation was to identify whether our methods for tokenization (Section 3.3) and event binding (Section 3.7) would work well on complex websites. For the former case, we were interested in assessing how many strings could be automatically tokenized, as the untokenized ones would potentially increase translation effort. For the latter case, we were interested in knowing how many elements could be localized by end-users, for which event binding should succeed without impacting regular page browsing. We queried the top 25 Alexa[30] global sites to conduct these experiments.

5.2.1. Procedure. We instrumented the JITL bookmarklet in order to gain insights about the goals discussed above. First, we computed the number of strings *before* tokenization, i.e., the original strings as shown on each browsed site. We also computed the number of unique strings, removing duplicates, as they are theoretically the ones that should be translated. Next, we computed the number of unique strings that were automatically tokenized. Then, we computed the number of strings *after* tokenization,

[30] `http://www.alexa.com/topsites`

i.e., the strings that should be eventually translated. With this information, we computed the effort saving that is achieved by string tokenization. In addition, all of the tokenized strings were stored in a flat file database, in JSON format. This way, tokenization results could be manually inspected later.

On the other hand, JITL's event handlers were modified to increment a global counter that accounted for the number of events that were successfully triggered in an automated way. Therefore, after attaching a click event listener to the `document` element, all localizable UI elements were programmatically forced to trigger a mouse click event. This simulates what a user would actually do on each page to localize a particular UI element. Whenever the global counter were increased, it would mean that event binding had succeed for such element. Otherwise, the event would have been prevented by other event handlers attached to that element, e.g., because the developer has specified `event.stopPropagation()`.

We executed the bookmarklet on the 25 Alexa top sites as of Jun 2014, excluding non-English sites since we were interested in manually inspecting both localized and localizable strings afterwards. For all websites, we inspected the home page if it had informative content. Otherwise, we issued the query "TOCHI" in a search form, if available. If no search form was available on the site, we clicked on the navigation menu links until finding a page with informative content.

5.2.2. Results and Discussion. The results of this experiment are shown in Table II. The "No. Strings" column indicates the number of strings found on the websites, *before* tokenization. The "To Translate" column indicates the number of strings that should be translated, *after* tokenization, together with the effort saving introduced by string tokenization. Finally, the "Actionable" column indicates the number of strings that could be localized on demand by the user.

We observed that for online shopping websites such as aliexpress.com, the proportion of tokenized strings is fairly high. This is mainly explained by the large amount of price and date-related strings, which are more prone of being localized. Conversely, we observed that informational websites such as twitter.com have less text that is suitable for tokenization. Overall, 26,208 strings were analyzed, of which 15,485 are unique. In principle, these unique strings are the ones that should be translated. However, 690 of the unique strings can be tokenized, leading to savings of 45.9%; i.e., overall near half of the strings would have to be eventually translated. This suggests that string tokenization actually leads to important effort savings.

However, was our tokenization method missing important tokenization patterns? A manual inspection of the tokenized strings revealed that most of the potentially tokenizable strings were indeed well tokenized. Nevertheless, there were some rare cases where developers seemed to have templatized strings; e.g., we believe that `"Welcome back, John"` is likely to be templatized by `"Welcome back, %s"`. Interestingly, we observed that for most of these cases the templatized string was wrapped in DOM elements such as `<a>` or `<strong>`; e.g. `"Welcome back, <strong>John</strong>"`. Therefore, JITL was able to avoid repeating translations like these, since duplicated strings were placed at different text nodes and were thus translated only once. In any case, we observed that the number of missed tokenization cases was merely anecdotal on the analyzed websites.

Additionally, JITL only parsed digits, not floating point numbers or negative signs. While regular expressions for matching all kind of numbers are well-known, sometimes they could introduce undesirable errors. For instance, the string `3-4` would be parsed as `%d%d`. However, `%d-%d` is preferred, since in a specific locale the - symbol could be replaced by a different symbol. Furthermore, month names, week days, PM, AM, etc. were not tokenized. Although it would seem logical to try to tokenize dates and hour-related strings as a whole, that would prevent users to localize them properly,

Website	No. Strings		Tokenized	To Translate		Actionable
	Total	Unique		Unique	Saving (%)	
activities.aliexpress.com	3,045	1,677	69	350	79.1	3,045
best-of-tumblr.tumblr.com	191	78	9	77	1.3	191
darraghdoyle.blogspot.com	1,182	744	24	628	15.6	1,182
blog.wordpress.com	354	276	14	244	11.6	354
wikipedia.org	92	90	5	90	0.0	92
go.com	189	162	7	160	1.2	189
imgur.com	6,750	2,369	100	1,018	57.0	6,750
newyork.craigslist.org	502	325	19	220	32.3	502
stackoverflow.com	1,427	988	40	514	48.0	1,427
store.apple.com	480	259	24	195	24.7	480
vube.com	1,190	1,116	21	330	70.4	1,190
amazon.com	803	629	36	581	7.6	803
ebay.com	849	664	56	559	15.8	849
imdb.com	347	283	23	240	15.2	347
microsoft.com	204	129	21	121	6.2	204
neobux.com	357	324	29	191	41.0	357
pinterest.com	511	282	5	251	11.0	511
reddit.com	4,817	3,478	81	1,223	64.8	4,817
search.yahoo.com	166	117	11	113	3.4	166
twitter.com	943	394	24	369	6.3	943
adcash.com	121	70	3	70	0.0	121
facebook.com	566	299	12	241	19.4	566
google.com	319	198	29	189	4.5	319
linkedin.com	392	255	12	195	23.5	392
youtube.com	411	279	16	209	25.1	411
Total	26,208	15,485	690	8,378	45.9	26,208
Mean	1,048.3	619.4	27.6	335.1	45.9	1,048.3
Median	480	283	21	240	15.2	480
SD	1,577.1	805.1	24.6	285.1	64.5	1,577.1

Table II: Statistics of tokenizable and localizable strings for the top 25 Alexa global sites in English. All results depict the total number of strings found in each category.

e.g., translating the name of the months separately. Therefore, these strings should not be tokenized so that users can translate them the way they want to. Also, speech marks (e.g. quotes, dashes, hyphens, or brackets) were not tokenized because they are language-dependent.

Finally, the last column in Table II represents the number of strings for which event binding succeed. In comparison to the total number of strings, it can be observed that each and every UI element could be localized with JITL. Therefore, we can say that our method for providing user interaction is quite robust, at least for the websites analyzed so far.

5.3. Collaborative In-place Localization

This experiment was aimed at assessing how JITL would be used by end-users. Thus, we performed a controlled in-lab evaluation. We recruited 10 Spanish speakers aged

26–36 with an advanced English level. All participants were aware of the basis of software localization.

5.3.1. Procedure. Participants were told to translate the Wordpress website (see Section 5.1) into their native language, using our browser extension. Since Wordpress has a large number of sections, participants were told to "localize the elements that a Spanish speaker [with no English knowledge] would need in order to understand how to manage the blog posts." This statement led participants to consistently browse 3 pages at most: the login page, the dashboard, and the 'admin posts' page. At the end of the session, participants submitted their translations to our server and were rewarded with a gift voucher.

5.3.2. Results and Discussion. Results were quite encouraging. In just 5 minutes, 57 different UI elements were collaboratively translated, resulting in 410 localization pairs (i.e., source texts plus their target translation) including duplicated pairs, too. We identified at most 291 potentially localizable elements among all of the pages that participants browsed, including hidden help messages and items in deep menus. Therefore, not everyone localized the exact same number of elements. Nevertheless, given the high-level articulation of the task, it is understandable that many elements were left untranslated as long as they would not be crucial to achieve the task goal.

On average, each user contributed with 41 translations ($SD = 8.6$). Figure 15a shows the histogram of resource strings with different translations. It can be observed that more than half of the UI elements were assigned a couple of different translations, while it was not unusual to have up to 4 translations for each string. We observed that most of the differences among submitted translations were due to the use of synonyms and punctuation symbols; c.f. Figure 13b.

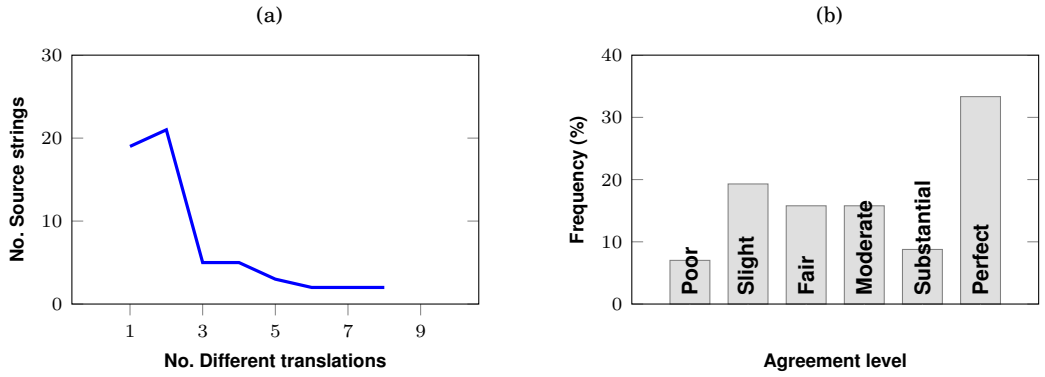


Fig. 15: Distribution of submitted translations per source strings (a) and histogram of user agreements (b).

Figure 15b shows the relative frequency of user agreements, according to Eq. (1) and the following categories [Landis and Koch 1977]: poor, slight, fair, moderate, substantial, or perfect. It is worth noting that for one third of the source strings the user agreement was considered "perfect." This is explained in part because some UI elements were fairly simple to translate (such as navigation links) and thus it was expected that in these cases users would submit very similar translations. Indeed, short strings are a particular feature of UI localization [Muntés-Mulero et al. 2012; Leiva and Alabau 2014], and we believe that JITL is specially interesting to deal with such

strings. Additionally, the remaining translations were uniformly distributed between "slight" and "moderate." These results can be leveraged as a means of setting up a level of confidence for a particular translation. This way, the webmaster or the application owner could decide that, e.g. in case of an overall poor agreement, the corresponding resource string should be sent to a professional translator.

In general, participants reported that they were happy to test JITL. They felt the in-place technique was really easy to use, and they expressed the intention to contribute with translations for their favorite websites. Hence, it seems plausible that further deployment of JITL would be successful.

Note that, as commented in Section 1.2, measuring the time required by end-users to explore *all* pages on a website would not be a natural comparison with respect to the time it would take translators to use conventional localization software. JITL users are expected to localize when they need it, and so they would seldom crawl a complete website in a row. Hence, we did not considered this metric in the current experiment. However, the interested reader is referred to Leiva and Alabau [2014] where the authors looked at this very specific topic.

5.3.3. Visualization Analysis. We visualized the gathered data in order to gain more insights about the localization process followed by our participants. Interestingly, we observed that there was an evident consensus regarding which elements should be localized to achieve the task goal, even regarding the order in which localization should take place. In the following we describe the main results derived from this analysis.

Fig. 16: Visualizing localization results on the Wordpress login page. Circle radii indicate the number of times an element was localized, whereas lines depict the localization order. The WordPress name and logo are registered trademarks of the WordPress foundation, used with permission.

Figure 16 displays the frequency of localized UI elements along with the localization trails followed by all participants on the Wordpress login page. Circle radii represent the number of times an element was localized. As observed, all strings were localized by all participants on that page. Additionally, one participant tried to sign in without filling in the login form, so an error message showed up. That specific message was not shown to the rest of the participants, and thus it was not translated 90% of the time. Nonetheless, according to the JIT philosophy, UI elements are localized only when they are needed. Hence, as observed, frequent errors like this one would be localized soon enough in a real setting.

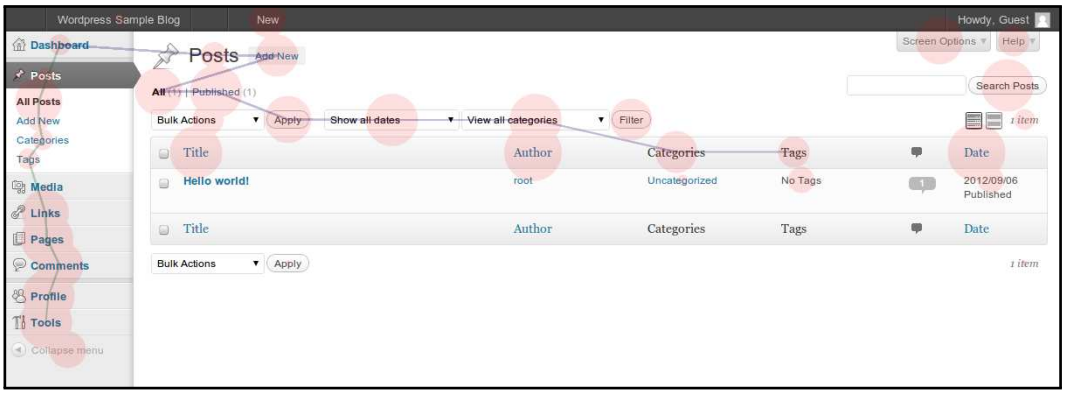


Fig. 17: Visualizing localization results on the Wordpress 'edit posts' page. Circle radii indicate the number of times an element was localized. Hidden localized elements such as dropdown items are not shown in the figure. Lines highlight the first 10 elements localized by 2 users, illustrating two of the strategies that users followed on this page.

After logging in to Wordpress, users were redirected to the dashboard section and quickly navigated away to the 'admin posts' section using the aside menu. Figure 17 shows a similar visualization to Figure 16 where, for illustration purposes, we display the localization path of the first 10 elements that followed two participants. In contrast to the patterns observed on the login page, here both participants chose completely different strategies to start localizing the UI. Although not shown in the figure, strings in drop-down lists and other hidden elements were also localized for that particular page. These results put forward the fact that translations are propagated to similar UI elements having the same source text; see e.g. the elements at the bottom part of Figure 17.

A more detailed analysis revealed that UI elements that trigger some "action" such as buttons, menu links, and table headings were typically localized earlier than less important parts such as dates or text paragraphs. We believe this information is of special relevance to inform about UI element localization preference; see next analysis.

5.3.4. Localization Order Analysis. In light of the previous visualization results, we wondered if there would be any significant correlation regarding the order that users followed while localizing each page. Figure 18 shows the rank correlation coefficient (Spearman's ρ) against the number of localized elements. For each value in the x-axis we considered the order in which the first n elements were localized. The order followed by each participant was set as ground truth and compared to the order followed by the rest of the users. Thus, each value in the y-axis accounts for the average ρ up to n localized elements.

It can be observed that the first 6 elements, which belong to the login page, accounted for a large consensus ($\rho > 0.9, p < .01$). As previously explained, that page is rather simple and it seemed natural to follow a visually sequential order. Then, consensus dropped for about the next 10 elements to stabilize at $\rho = 0.74$. This is due to the fact that participants took different routes to approach the localization of the UI. For example, most users decided to localize the main menu in the first place, whereas other users focused on localizing table headers and links to begin with. Then, after localizing about 10 menu items, the first group of users switched to localizing the main table, which increased consensus to $\rho = 0.79$ around $n = 30$ elements. Afterward, the remaining elements were considered as less relevant, and so they were localized in

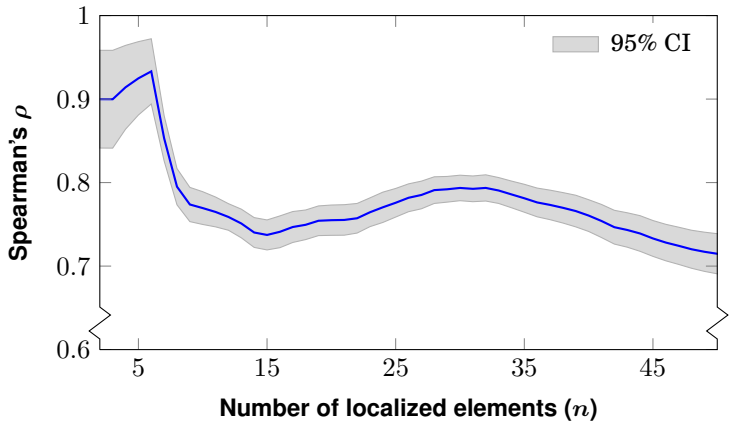


Fig. 18: Localization order analysis. Correlation is shown as a function of the number of localized elements. Shaded area represents 95% confidence intervals.

random order to a greater or a lesser extent. As a result, consensus decayed again to stabilize in $\rho = 0.71$. All in all, ρ was found to be greater than 0.7 in all cases, which means that there was significant consensus regarding to which strings should be localized earlier.

These results are particularly interesting for practitioners because it seems to be highly predictable which UI elements are likely to be localized in the first place. Furthermore, often it is necessary to localize just the essential parts of a UI, either because of competitive advantage or economical reasons. For instance, an advanced word processor may comprise an important number of menus and options, but actually only a few of these will be used by regular users. Then, localizing first what is most important would allow to reach emerging markets or even introduce a new product sooner than the competence. Therefore we believe JITL is an appealing approach from this perspective.

6. DISCUSSION AND IMPLICATIONS

Software internationalization has not changed substantially in the last years, mainly because it is still a process that takes place deep in the source code [Gross 2006]. Further, it is difficult to provide automation for something that usually is done by hand. In this regard, JITL provides a clear advantage over current internationalization technology for websites and web applications. We now briefly highlight the key features of our method, as compared to the traditional i18n+L10n process, followed by a discussion about its limitations.

First of all, JITL empowers the user (pull strategy), while the traditional process requires the intervention of a website owner (push strategy). This should be the takeaway message from this article, see Figure 1. To make such pull strategy possible, JITL enables localization without previous internationalization. This eventually results in lower development costs and less iterations in the localization process, saving thus time and money. Also, less iterations allow for frequent updates at a faster pace in comparison with the traditional process.

	JITL	Traditional
Localization strategy	pull	push
Manual internationalization	no	yes
Costs (manpower, time, money)	low	high
Frequent & immediate updates	yes	no
Runtime overhead	yes	no
Localization metadata	yes	no
Dialect variants	yes	no
Community-based localization	yes	no*
In-place localization	yes	no*
Partial localization	yes*	no
Prioritized localization	yes	no
Trusted localization	no*	yes
Infrequent & dead-code translations	no*	yes

Table III: Approaching UI localization in "the JITL way" vs. the traditional i18n+L10n process. An asterisk indicates that some exceptions apply, which we comment below.

As a result of automatic internationalization, JITL introduces a runtime overhead (in the order of milliseconds, though). In contrast, the traditional process is mostly static, as typically UI strings are compiled to machine-readable code.[31]

Both JITL and the traditional process allow localization data to be exported as an external file, but JITL also exports localization metadata; e.g., each string is augmented with a URL that points to the UI element that contains such string. Among other applications, these metadata make it possible to generate dialect variants for the same language over time. For instance, by geolocating the user IP we can estimate better user agreement rates (e.g. in USA, American English words should be worth more than British English words) or infer linguistic divergence (e.g., one word may have different interpretations in the same country). In the traditional process this is by far more costly or difficult to achieve, as translators for each possible dialect are hard to find. What is more, translating to such dialects or minor languages may have a low strategic value for the company, ending up not being localized. However, this can be of great interest or cultural value for the user, who now is empowered by JITL to do so.

JITL enables community-based localization out of the box. To achieve the same effect with the traditional process, a third-party tool and possibly some engineering effort would be required. Because JITL enables in-place localization, the user can correct mistakes as they go and the visual context is always available to the user. In the traditional process this capability is seldom available, see Section 2.

JITL makes it easier to produce an incremental localization, e.g., one page at a time. This is specially interesting not only to companies that cannot afford the entire localization of an application, but also to shorten times to show an implemented idea to an international audience, develop early prototypes when there is no option for turnaround times, or promoting the ease of access to people that do not have sufficient linguistic competence to use the application in a particular foreign language.

Through the traditional process, an application is localized all at once and always at the testing phase. JITL fundamentally operates at the distribution phase, according to the JIT philosophy. This typically results in prioritized localization, since the most relevant strings (according to the users' criteria) are translated in the first place. On the other hand, it is possible to use JITL during testing as well. This would resonate with the JIT philosophy, as strings would no longer be translated "only when they

[31] Dynamic string localization is also possible, which avoids locale recompilation.

are needed." Instead, the user would have to translate all strings showing up during testing.

JITL entails a great opportunity for open source projects or freely available web pages, since users can seamlessly contribute to produce translations into potentially every language. This in turn can be leveraged to construct feature-rich multilingual corpora to train machine translation systems [Alabau and Leiva 2014], for which translating UI messages is still a challenging issue [Muntés-Mulero et al. 2012]. Nevertheless, JITL translations in principle are less confident in comparison with the traditional process, since potentially *anyone* can contribute with JITL. Therefore, JITL requires data analysis methods such as input agreement (Section 5.3) or simply paying for human revision.

JITL can also be used to identify "dead code;" for example, a software patch that has removed some widgets from the UI but has forgotten to remove their associated lists of options. As Kovacs [2012] pointed out, identifying and removing this dead code could make the codebase more robust, and reduce the burden on translators, as they would no longer need to localize these strings. On the contrary, everything gets translated under the traditional process. Moreover, in JITL infrequent messages are translated as soon as they appear.

Finally, we should mention that the checklist shown in Table III is not definitive nor exclusive. For instance, JITL can be integrated in already-internationalized code. Simply put, the developer should instrument the function used for translating text (e.g. `gettext()`, `translate()`, or similar) so that each UI element gets the data attribute `jitl-localizable="true"`. This can be achieved by writing a function decorator; i.e., the translation function does not only return plain text, but also the translated text is enclosed by a `<span>` element. In any case, having already internationalized elements marked up in the source code would allow JITL to skip the first four steps described in Section 3; see also Figure 2.

6.1. Limitations of Current Implementation

We have shown that JITL enables a number of interesting scenarios, nonetheless our implementation is not exempt of limitations. First and foremost, because of its automated nature, in principle only leaf DOM nodes are internationalized, since leaf nodes are the ones that contain text. This may lead JITL to mismatch relevant resource strings, although most of them are correctly retrieved, which ultimately allows developers to save an important amount of engineering effort; see Section 5.1. Also, it is possible to supervise all strings that should be decoupled from the source code by using the interactive internationalization application; see Section 4.1.

One technical limitation that cannot be avoided is the localization of buttons that do not have a `value` attribute (e.g., `<input type="file" />`). These particular form elements are internally handled by the browser itself, and so they are unfortunately out of JITL's control. This means that they would be localized into the browser's language according to the localization guidelines of the browser vendor, instead of a particular user's needs.

String tokenization is a critical procedure in order to get the UI internationalization right. So far, there might be some exceptions where our approach fails, but until now we are not aware of any failure case because of an ill-executed regular expression. However, it might be desirable to make string tokenization more sophisticated. A particular case would be the automatic detection of entities (e.g., names, verbs, etc.) or user IDs (e.g., login names). For achieving this, highly refined techniques would be needed, such as integrating a lexical parser. However, this would not scale well because of the potential languages and/or locales that JITL would need to support. Moreover, our highest priority was keeping JITL simple.

JITL makes user interaction performance-friendly by attaching event listeners to the `document` element. As a downside, there is a special case where this would preclude user interaction. This is is the case of elements that have specified `event.stopPropagation()` in `mouseover`, `click` or `keyup` event handlers, provided that these handlers were attached prior to loading JITL. In our Tokenization and Interaction study (Section 5.2) we have not observed any collision in this regard, but we believe it is worth mentioning for the insightful reader.

Another limitation, derived from the in-place localization capability, is related to those elements that are difficult to localize at runtime; e.g., messages that appear for a very limited amount of time on the UI (e.g., `"Loading..."`). To address this issue, we have mentioned two alternatives. On the one hand, since JITL can generate a PO file on the fly, strings that cannot be localized at runtime can be eventually localized offline. On the other hand, the browser extension provides a queue of all seen strings that would allow the user to localize those strings online. Also, we should mention that some texts only appear under special circumstances like error messages or hidden options in menus, and so they would be localized less frequently than elements that are visually salient.

A problem that arises by the fact that only text nodes are localized is when these nodes should change order in the translated language. Ideally, interactive internationalization (Section 4.1) could be used to mark those nodes, and then expose the sub-DOM tree to the user so that she can rearrange both text and tags. For instance, `<span data-jitl-group="true">This is a <em>test</em> messsage.</span>` could be translated into `<span data-jitl-group="true">Esto es un mensaje de <em>prueba</em>.</span>`. However, this would imply that users could insert and modify any HTML in the text, leading to a potential security flaw. Hence, node reordering could be approached in a more explicit way, possibly by means of a drag-and-drop interaction that rearranges the inner nodes in the desired order.

It must be pointed out that DOM modifications may cause CSS and JavaScript errors when these are tightly coupled with the DOM structure. Although we have not observed this problem in any of the analyzed websites, we are well aware that, in this specific situation, DOM modifications could prevent a site to behave as it was designed. At this moment, this cannot be easily solved, and thus we would need the user intervention to inform that a page is problematic for JITL. Then, we could investigate ways to address that particular issue.

True to its philosophy, JITL aims to localize what is required, when it is required. This is the best case scenario, i.e., when a user spots a sentence that has not been translated into her language, or a translation error is bothering her, she is simply able to amend the text on the UI. Therefore, end-users are not expected to perform an exhaustive exploration of the pages on a website and so it is not clear when the website would be localized in its entirety. One option would be building a crawler with node.js (so that it can execute JavaScript code) and keep track of the less browsed pages or the less interacted elements. Then, when users had collaboratively localized, say, half of the site, the site owner could be informed so that she could hire a professional software translator to complete the localization of the site. In addition, we foresee other incentives to motivate people to contribute and share their localization data. Just to name a few: rely on volunteers' work or pay the users straight away, introduce gamification techniques, or encourage people to practice their language skills.

It is also worth of discussion the "tracing from code to UI" question. For instance, how much restructuring of an existing site can be made to accommodate professional-level localization? How could developers integrate *persistently* all user-submitted translations in their website? We have discussed that JITL could be integrated in already-internationalized code. Furthermore, we argued that developers can include a single

line of JavaScript code that retrieves all user contributions from a centralized repository (Section 4). This is of special importance for websites deployed at a shared hosting, like Google sites or Github. Websites at most of these shared hostings cannot be internationalized, either because the shared hosting does not allow it or just because developers do not have the appropriate tools (e.g. PHP was compiled in the shared hosting without gettext support). Even in this case, these sites could be benefited from JITL, since it does not have to manipulate server-side code. All in all, JITL can be seen as an additional, textual layer that sits on top of any website.

Finally, UI localization may require incorporating changes to aesthetics and colors, something our method cannot cope with at the moment. This is definitely a research avenue worth considering for future work. All in all, we believe that JITL might influence largely the way the Web can be localized, specially for those companies with highly valued products but with low budgets, and open source projects with low economical resources but with a huge user base.

7. CONCLUSION

JITL notably advances the state of the art on web-based software localization. Our work contributes with core methods and implementation design principles for automatically internationalizing web-based UIs, together with demonstrations of their value through a number of implemented applications and a series of empirical evaluations.

We apply an industrial design principle: resources are pulled out of the UI, not pushed by a company. To date, UI internationalization requires explicit webmaster's intervention to deploy translation resources. In contrast, because of its automated nature, JITL allows developers, translators, and arbitrary users to localize websites that are not under their control. This is specially interesting for websites or web-based applications that companies do not have in mind to localize. Ultimately, JITL could become the right approach for localizing the "long tail" of websites, and for expanding the number of languages that are available in the Web. Our work thus connects with HCI researchers and practitioners interested in making web-based UIs linguistically accessible to users worldwide.

There is still an opportunity for future work that characterizes other localization methods. For instance, although we have primarily focused on web-based UIs, we believe that our method could be extrapolated to other applications that support structured data hierarchies; e.g., interfaces created in XUL, ActionScript, or GTK. As a consequence, we are studying how JITL could be deployed in other platforms beyond the browser. Another avenue for future work is studying how the JITL idea could work outside of structured platforms, for instance, in compiled software like C++ applications.

Looking forward, we believe this work enables research opportunities in web-based software localization that were inconceivable in the past. It is our hope that JITL will be a useful and complementary method to existing techniques, and that end-users, translators, and other professionals working in the localization industry will appreciate such a help. JITL is released as open source software, so that anyone will be able both to contribute and benefit from it.

ACKNOWLEDGMENTS

We thank Germán Sanchis-Trilles and Roberto Silva for fruitful discussions about the potential of JITL. We also thank the anonymous TOCHI reviewers for providing valuable feedback to strengthen this manuscript.

REFERENCES

L. R. Abraham. 2009. Cultural differences in software engineering. In *Proceedings of International Solvent Extraction Conference (ISEC)*. 95–100.

V. Alabau, and L. A. Leiva. 2014. Collaborative Web UI Localization, or How to Build Feature-rich Multilingual Datasets. In *Proceedings of the European Association for Machine Translation (EAMT)*. 151–154.

L. Bentivogli, M. Federico, G. Moretti, and M. Paul. 2011. Getting Expert Quality from the Crowd for Machine Translation Evaluation. In *Proceedings of MT Summit*. 521–528.

J. Cardeñosa, C. Gallardo, and Álvaro Martín. 2006. Internationalization and Localization after system development: a practical case. In *Proceedings of i.TECH, Information Research and Applications*. 1–8.

P. K. Chilana, A. J. Ko, and J. O. Wobbrock. 2012. LemonAid: Selection-based Crowdsourced Contextual Help for Web Applications. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI)*. 1549–1558.

R. W. Collins. 2002. Software Localization for Internet Software: Issues and Methods. *IEEE Software* 19, 2 (2002), 74–80.

M. Dixon, and J. Fogarty. 2010. Prefab: Implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI)*. 1525–1534.

M. Dixon, D. Leventhal, and J. Fogarty. 2011. Content and hierarchy in pixel-based methods for reverse engineering interface structure. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI)*. 969–978.

B. Esselink. 2000. *A Practical Guide to Localization*. John Benjamins Publishing Company.

B. Esselink. 2003. The evolution of localization. Position paper. Available at `http://isg.urv.es/seminars/2003_localization_online/esselink.pdf`. (2003).

S. Gross. 2006. *Internationalization and Localization of Software*. Master's thesis. Eastern Michigan University.

J. M. Hogan, C. Ho-Stuart, and B. Pham. 2004. Key challenges in software internationalisation. In *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation (ACSW Frontiers)*. 187–194.

A. Hsieh, T. Hausman, N. Titus, and J. Miller. 2008. If you build it, they will come... if they can: pitfalls of releasing the same product globally. In *Proceedings of Extended Abstracts on Human Factors in Computing Systems (CHI EA)*. 2591–2596.

H. Huang, and E. Trauth. 2007. Cultural Influences and Globally Distributed Information Systems Development: Experiences from Chinese IT Professionals. In *Proceedings of the ACM SIGMIS CPR conference on Computer personnel research*. 36–45.

T. Hunt. 2013. Cost effective software internationalisation. *Journal of Applied Computing and Information Technology* 17, 1 (2013), 1–6.

S. Kawanaka, Y. Borodin, J. P. Bigham, D. Lunn, H. Takagi, and C. Asakawa. 2008. Accessibility Commons: A Metadata Infrastructure for Web Accessibility. In *Proceedings of the International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS)*. 153–160.

K. Keniston. 1997. Software Localization: Notes on Technology and Culture. Working Paper #26, Massachusetts Institute of Technology. (1997).

G. Kovacs. 2012. ScreenMatch: providing context to software translators by displaying screenshots. In *Proceedings of Extended Abstracts on Human Factors in Computing Systems (CHI EA)*. 1375–1380.

J. R. Landis, and G. G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174.

L. A. Leiva, and V. Alabau. 2012. An automatically generated interlanguage tailored to speakers of minority but culturally influenced languages. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI)*. 31–34.

L. A. Leiva, and V. Alabau. 2014. The Impact of Visual Contextualization on UI Localization. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI)*. 3739–3742.

T. V. Luong, J. S. H. Lok, D. J. Taylor, and K. Driscoll. 1995. *Internationalization: Developing Software for Global Markets*. John Wiley & Sons.

K. A. McKethan, and G. White. 2005. Demystifying Software Globalization. *Translation Journal* 9, 2 (2005), 1–8.

V. Muntés-Mulero, P. P. Adell, C. España-Bonet, and L. Màrquez. 2012. Context-Aware Machine Translation for Software Localization. In *Proceedings of the European Association for Machine Translation (EAMT)*. 77–80.

T. Ohno. 1988. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press.

D. Patel, C. Lawson-Johnson, and S. Patel. 2009. The Effect of Cultural Differences on Software Development. In *Proceedings of ICT Structural Development*. 250–263.

M. Paul, E. Sumita, L. Bentivogli, and M. Federico. 2012. Crowd-based MT Evaluation for non-English Target Languages. In *Proceedings of the European Association for Machine Translation (EAMT)*. 229–236.

M. A. Pérez-Quiñones, O. I. Padilla-Falto, and K. McDevitt. 2005. Automatic Language Translation for User Interfaces. In *Proceedings of the 2005 Conference on Diversity in Computing (TAPIA)*. 60–63.

K. Reinecke, and A. Bernstein. 2011. Improving performance, perceived usability, and aesthetics with culturally adaptive user interfaces. *ACM Transactions on Computer-Human Interaction (TOCHI)* 18, 2 (2011), 8:1–8:29.

T. Steiner, R. Verborgh, and R. Van de Walle. 2012. Fixing the Web One Page at a Time, or Actually Implementing xkcd #37. In *Proceedings of the World Wide Web conference (WWW)*. 1–3. Developers Track.

H. Sun. 2001. Building a culturally-competent corporate web site: an exploratory study of cultural markers in multilingual web design. In *Proceedings of the Annual International Conference on Design of Communication (SIGDOC)*. 95–102.

SurveyMonkey 2013. SurveyMonkey Goes Global – A case study. Available at `http://www.smartling.com/static/pdf/smartling-case-study-surveymonkey.pdf`. (2013). Retrieved Aug 2, 2013.

H. Takagi, S. Kawanaka, M. Kobayashi, T. Itoh, and C. Asakawa. 2008. Social Accessibility: Achieving Accessibility Through Collaborative Metadata Authoring. In *Proceedings of the International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS)*. 193–200.

O. D. Troyer, and S. Casteleyn. 2004. Designing Localized Web Sites. In *Proceedings of Web Information Systems Engineering (WISE)*. 547–558.

M. Tschernuth, M. Lettner, and R. Mayrhofer. 2012. Unify localization using user interface description languages and a navigation context-aware translation tool. In *Proceedings of the ACM SIGCHI symposium on Engineering interactive computing systems (EICS)*. 179–188.

X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. 2009. TranStrL: An automatic need-to-translate string locator for software internationalization. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 555–558.

X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. 2010. Locating Need-to-translate Constant Strings in Web Applications. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 87–96.

A. W. Yeo. 2001. Global-software development lifecycle: an exploratory study. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI)*. 104–111.