A Language for End-user Web Augmentation: Caring for Producers and Consumers Alike

OSCAR DÍAZ, University of the Basque Country (UPV/EHU)
CRISTÓBAL ARELLANO, University of the Basque Country (UPV/EHU)
MAIDER AZANZA, University of the Basque Country (UPV/EHU)

Web Augmentation is to the Web what Augmented Reality is to the physical world: layering relevant content/layout/navigation over the existing Web to customize the user experience. This is achieved through JavaScript (JS) using browser weavers (e.g. Greasemonkey). To date, over 43 million of downloads of Greasemonkey scripts ground the vitality of this movement. However, Web Augmentation is hindered by being programming intensive and prone to malware. This prevents end users from participating as both producers and consumers of scripts: producers need to know JS, consumers need to trust JS. This paper aims at promoting end user participation in both roles. The vision is for end users to prosume scripts as easily as they currently prosume their pictures or videos. Encouraging production requires more “natural” and abstract constructs. Promoting consumption calls for augmentation scripts to be easier to understand, share and trust upon. To this end, we explore the use of Domain-Specific Languages (DSLs) by introducing Sticklet. Sticklet is an internal DSL on JS, where JS generality is reduced for the sake of learnability and reliability. Specifically, Web Augmentation is conceived as fixing in existing websites (i.e. the wall) HTML fragments extracted from either other sites or Web services (i.e. the stickers). Sticklet targets hobby programmers as producers, and computer literates as consumers. From a producer perspective, benefits are three-fold. As a restricted grammar on top of JS, Sticklet expressions are domain-oriented and more declarative than their JS counterparts, hence speeding up development. As syntactically correct JS expressions, Sticklet scripts can be installed as traditional scripts and hence, programmers can continue using existing JS tools. As declarative expressions, they are easier to maintain, and amenable for optimization. From a consumer perspective, domain specificity brings understandability (due to declarativeness), reliability (due to built-in security) and “consumability” (i.e. installation/enactment/sharing of Sticklet expressions are tuned to the shortage of time and skills of the target audience). Preliminary evaluations indicate that 77% of the subjects were able to develop new Sticklet scripts in less than thirty minutes while 84% were able to consume those scripts in less than ten minutes. Sticklet is available to download as a Mozilla add-on.

Categories and Subject Descriptors: D.2 [Software]: Software Engineering

General Terms: Standardization, Design

Additional Key Words and Phrases: Web Augmentation, Domain Specific Language, End-User Programming, Greasemonkey, JavaScript

1. INTRODUCTION

The evolution of Web applications can be staged based on the degree of user participation. Web 1.0 limits laymen activities to mainly reading and form filling. Next, Web 2.0 puts content authoring in the user’s hand: blogging, tagging or wiki editing are nowadays common practices. This work is about the last frontier of layman participation: End-User Programming in the Web. The challenge is for end users to provide/share their own functionality on top of existing Web applications. Mashups [Yu et al. 2008] and Web Augmentation [Bouvin 1999] illustrate this approach. Mashup techniques are available to build new applications out of existing Web resources (e.g. APIs, RSS feeds). By contrast, Web Augmentation does not create a new application. Rather, the rendering of an existing application is augmented at the client-side to improve the user experience. We focus on Web Augmentation.

Web Augmentation superimposes additional content/layout/navigation over existing Web applications. Examples of what this technology generically enables include reorganizing page content, supplementing page data, changing fonts and formats, etc. [McFarlane 2005; Filman 2006]. Concrete examples can be found at userscripts.org, a popular repository for augmentation scripts. This site reports thousands of consump-
tions (which evidences the effectiveness of the approach) but a far smaller number of providers, and, in any case, a number still far from the Internet's 1.4 billion end users. The use of general-programming languages such as JavaScript partially explains why Web Augmentation production is so far limited to programmers. These scripts are prone to malware, and difficult to create and maintain [Pilgrim 2005]. As a result, only dedicated programmers are prepared to produce scripts, and only courageous consumers are willing to install them. We strive to depart from this situation, moving Web Augmentation to a scenario of Web2.0 prosumers. The vision is for end users to prosume scripts as easily as they currently prosume pictures or videos.

However, prosuming pictures is not the same as prosuming scripts. First, producing scripts requires a deep knowledge of JavaScript. Second, consuming a script might require to understand what the script is about; next, trust it, then install it, and finally, enjoy it. “Consumability” has to do with all the hurdles you have to go through before using an artefact. The more cumbersome these obstacles are, the less consumable the artefact is. When users are not obliged to consume one’s artefact (as it is the case in Web2.0 scenarios), we risk “the consumability effort” to go above the “effort threshold” that users are willing to accept to enjoy such artefact. Therefore, “scripts 2.0” should not only be easy to specify but also easy to consume i.e. self documented, malware free by construction, and easy to install. Current JavaScript code is far from being 2.0: it is difficult to program, hard to understand, and open to malware [Arellano et al. 2010]. A list of JavaScript security risks is documented in the Google Caja website [Google 2007a].

Attempts to make Web Augmentation accessible to end users mostly focus on producers but overlook consumers. They mainly resort to either Visual Programming Tools (e.g. Platypus [Turner 2005]) or Application Programming Interfaces (API) (e.g. Chickenfoot [Bolin et al. 2005]). Visual Programming Tools facilitate the production of scripts but the final script (the one to be consumed) is still JavaScript code. No changes are made in the consumability front. Alternatively, API-based approaches result into a leaner code, easier to write and understand. However, these APIs are not full-fledged languages and hence, users need to resort to the full expressiveness of JavaScript to write their augmentations. This jeopardizes security, and keeps the production bar only accessible to programmers. As noted by the authors of an API for augmentation: “Chickenfoot customizations are essentially JavaScript programs, so Chickenfoot currently does not support non-programming users. We assume that a Chickenfoot developer has some knowledge of JavaScript and HTML...” [Bolin et al. 2005], and a related quote “Chickenfoot scripts run with no security restrictions, since they are developed and run by the end-user, not downloaded from a potentially malicious remote site” [Bolin and Miller 2005]. Certain we may be about the benefits of APIs, consumer requirements remain yet to be addressed.

Based on these observations, we introduce three innovative contributions. First, we provide a design space where to place different competing requirements for Web Augmentation tools. Drawing on the insights of the “Attention Investment” model [Blackwell 2002], this design space goes along two dimensions: the required attention to accomplish the augmentation task, and the available attention to perform this task. Augmentation tools should provide an equilibrium between the attention required to fulfil the augmentation, and the user's available attention to accomplish this augmentation. Besides traditional requirements such as expressiveness or maintainability, the search for this equilibrium introduces concerns such as familiarity (i.e. to what extent the tool resembles what the users might be acquainted to), operability (i.e. ability of the tool to be easily operated by a given user in a given environment) or shareability (i.e. the facility to share augmentation scripts).
The second contribution proposes Domain Specific Languages (DSLs) as a promising way to fulfil the previous requirements. DSLs are full-fledged languages tailored to specific application domains by using domain-specific terms [Fowler 2010]. Specificity brings many benefits, namely:

— from a producer perspective, DSLs facilitate development (e.g. domain abstractions are closer to how users conceive the problem, facilitating engagement and production of scripts), debugging (e.g. tracing is also conducted in DSL terms) and maintenance (e.g. abstraction brings declarativeness). Target producers are hobby programmers. They should know about URLs, and a bit of HTML is recommended though not strictly necessary. They do not need to know about either JavaScript or any other programming language as the proposed DSL does not have explicit control instructions (i.e. neither 'while' nor 'if').

— from a consumer perspective, DSLs bring trust based on both code understanding (due to DSL declarativeness) and built-in security as provided by the DSL engine. In this way, consumers only need to rely on the provider of the DSL engine (who can be externally certified e.g. as a Mozilla add-on partner) rather than trusting a potentially unknown JavaScript programmer. In addition, domain specificity permits to customize the installation/edition/sharing of DSL expressions to the target audience. As target consumers, we focus on computer literates: they might have an account in Facebook or regularly use text processors but they might be afraid of installing plugins.

Finally, we provide a fully-functional DSL to test these ideas: Sticklet\(^1\). Sticklet is available as a Firefox add-on at https://addons.mozilla.org/addon/Sticklet/. We evaluated Sticklet in two ways: we built non-trivial examples to validate the expressiveness of the language (available at http://userscripts.org/users/sticklet), and we ran a usability study to examine quality-in-use for both consumers and producers. The results indicate that 77% of the subjects were able to develop new Sticklet expressions in less than thirty minutes while 84% were able to consume (i.e. install) a third-party sticklet script in less than ten minutes.

The paper starts by setting the design space (Section 2). This space serves to frame design concerns along three main scenarios: (i) augmentation for producers (Section 4), (ii) augmentation for consumers (Section 5), and (iii), augmentation for discretionary prosumers (Section 6). Evaluation, related work and conclusions end the paper.

2. SETTING THE DESIGN SPACE
Web Augmentation is not new. Layering web code at the client-side is being used to improve the affordance of third-party services. If this service is Skype, SkypeButton [Skype 2005] turns any phone number found in a web page into a button that launches Skype to call that number. If this service is AVG security warnings, LinkScanner [AVG 2010] is an augmentation utility that permits to scan search results from Google, Yahoo! or Bing, and places a safety rating next to each recovered link. Besides in-place service invocation, Web Augmentation can support a broad range of situational scenarios:

— On browsing an online journal (e.g. USA Today), you can be interested in the coverage that a given headline receives in another online newspaper, e.g. The NY Times. Skipping to the TNYT and searching for a related headline could be too cumbersome to do regularly. Rather, you would like the USA Today website to be augmented with

\(^{1}\)The suffix `let` is commonly used in Web programming to denote Web components (e.g. "servlet" or "portlet"). For sticklets, the container role is somehow played by the hosting page.
Fig. 1. **Sticklet design drivers.** Criteria are qualified by “P” or “C” based on their bias towards producers or consumers, respectively.

- A button placed by the *USA Today* headline that directly pops up the summary at *TNYT* for this headline.
- When rendering a book at *Amazon*, it could be useful to know the prices/comments for this book at other online bookshops.
- On weighting a job post at *www.monster.com*, it could be of interest to supplement monster data with information about the range of wages and conditions of similar jobs as found in other web sites (e.g. *jobs.trovit.co.uk*).

These examples illustrate short-term situational scenarios of end-user Web Augmentation (hereafter just “Web Augmentation”). The purpose is to make the Web more responsive to the unique and individual needs of each user. From this perspective, augmentation shares aims with Web Personalization [Rossi et al. 2001]. The difference stems from *who* sets the personalization. In the case of personalization, the application designer is in charge while users are passive consumers. The issue is that some personalizations might be of interest only for a small number of users (hence, lacking the scale that makes the personalization payoff), fall outside the business model of the Web application (e.g. *Amazon* lacks the interest in putting up comparative prices from other online bookshops) or be difficult to foresee by the application designer. This discrepancy between what application developers can build, and what individual end-users really need can be addressed with End-User Development [Repenning and Ioannidou 2006].

The theory of “Attention Investment” has been proposed as a basis for the design of End-User Development systems [Blackwell and Green 1999; Blackwell 2002]. This theory describes users’ decisions about how to allocate their attention in problem-solving as investments. Drawing on these insights, we arrange requirements based on the two parameters of this scenario: the users and the tasks. Users are characterised in terms
of the available attention to perform the task. Tasks are described in terms of the required attention to accomplish them. Although these dimensions go along a continuum, we provide just two discrete values for each dimension: "low" and "high" (see Figure 1).

Based on the available attention, users are clustered into conditioned prosumers (the available attention is high) versus discretionary prosumers (the available attention is low). The former conducts augmentation within an organization, and it is commonly job-oriented. This results in users being focused in their tasks. By contrast, discretionary prosumers conduct augmentation in a less pressing environment, and usually for self reward. This might lead to a more dispersed activity.

Based on the required attention, tasks are classified as lightweight (the required attention is low) versus heavyweight (the required attention is high). This criterion might depend not only on the sophistication of the endeavour itself but on the envisaged audience of the script. As aptly noted in [Ko et al. 2011], when the audience goes beyond oneself producers might need to consider concerns such as reliability, reuse, and maintainability, and be engaged in activities that reinforce these qualities, such as testing, verification, and debugging.

These two dimensions help to arrange requirements for augmentation tools along the so-identified quadrants. The more stringent the demands, the larger the set of attributes the tool should cater for. The bottom left quadrant (low required attention, high available attention) represents production by programmers that use augmentation to speed up some personal routine tasks. Basically, scripts are for self consumption. One of the main challenges in this scenario is to find a balance between expressiveness and learnability. More complex languages can address a wider range of problems but impose an increasing learning burden on users.

As we move rightwise, production becomes either more sophisticated or targets a wider audience. The latter introduces consumers as first-class stakeholders. Producers will look for support in testing, verification, and debugging of their scripts since reliability and maintainability become main concerns. From the consumer perspective, they surely value security, understandability or tailorability (i.e. permitting the consumers to adapt the script by themselves [Dittrich et al. 2006]). Some DSLs fall in this quadrant. Here, it is not expected for end-users to produce the DSL expressions on their own but just to read the code, understand what it means, and talk to programmers directly about necessary modifications. In this setting, DSLs are not a substitute for programmers but a way to increase their productivity while improving the reliability and understandability of software.

If we move upwards, we confront the shortage of attention. If attention is scarce, new requirements come into play. First, the operation of the whole tool should be intuitive (i.e. operability). If possible, tool design should capitalize on whatever aspect the target audience is familiarized with so that users can reapply what they already know (i.e. familiarity). Moreover, production tends to be less systematic where users commonly resort to evolutionary and exploratory prototyping (i.e. provisionability) [Green et al. 2006]. In addition, producers might be motivated by the possibility of exhibiting the final product to others as a demonstration of skill and technical mastery. Web2.0 scenarios make consumption a main incentive for production. This moves us to the upper right quadrant, i.e. the promotion of consumption in discretionary scenarios (i.e. no obligation to consume the script). Two additional aspects emerge: installability (i.e. the quality of requiring minimum installation burden) and shareability (i.e. facilitating script sharing).

By no means, we claim this is a complete list of requirements. Rather, this design space serves to frame the main design criteria considered during the Sticklet implementation. Design choices are subject to tradeoffs between factors that will value some
attributes while penalizing others. For instance, JavaScript tools mainly value expressiveness at the cost of penalizing understandability and reliability. On the other hand, Visual Programming Tools and API-based approaches favour easy production by trading expressiveness for learnability. Unfortunately, no matter the approach, they all leave consumers to face raw JavaScript code.

This work sets augmentation in a Web2.0 scenario: scripts are not only for self-consumption but also for sharing, where sharing in turn fuels production. This virtuous cycle is not however a free lunch. Easing consumption might imply reducing the expressiveness as well as reducing the visual aids to favour the operability and installability of the solution. There is thus no ideal tool for any augmentation situation, only designs that are more or less well suited to the activities of the people doing the augmentation. For the purpose of this paper, these people are hobby programmers as producers, and computer literates as consumers.

Next sections introduce Sticklet. We do so by gradually addressing concerns for different audiences: conditioned producers (Section 4); conditioned consumers (Section 5), and finally, discreitional prosumers (Section 6). But first, we provide a brief on JavaScript to highlight the main difficulties to be hidden from end users.

3. WEB AUGMENTATION THROUGH JAVASCRIPT

Using special weavers, third-party JavaScript code can make on-the-fly changes to the currently loaded Web page. Weavers are available for Firefox (e.g. Greasemonkey [Lieuallen et al. 2005]), Internet Explorer (e.g. IE7Pro or Turnabout), Safari (e.g. SIMBL + GreaseKit), and natively supported in Opera and Google Chrome. The running examples for this paper were tested for Mozilla Firefox 18.0.2 and Greasemonkey 1.7.1.

As an example, consider a popular script: BookBurro\(^2\). This script embeds price comparison in Amazon pages. Figure 2 shows the outcome before and after applying the script that injects the BookBurroPanel. This is achieved at the browser through the weaver. Weavers permit scripts to act upon Web pages at runtime. Pages are realized as DOM trees\(^3\). The script is triggered by User Interface events (UI events) on this

---

\(^2\)BookBurro is available at http://userscripts.org/scripts/source/1859.user.js.

\(^3\)The Document Object Model (DOM) is a platform- and language-independent standard object model for representing HTML or XML documents as well as an Application Programming Interface (API) for querying, traversing and manipulating such documents.
DOM tree (e.g. load, click). Event payloads provide the data to feed script handlers which, in turn, update the DOM tree. The script is outlined in Figure 3. The process goes as follows:

— interacting with a page triggers UI events (e.g. load),
— the script reacts to this event by triggering a handler (lines 6-39). The association between an event and a handler (a.k.a. event listener) is achieved through the `addEv-entListener` function (line 6),
— a handler can access any node of the page (using DOM functions such as `document.evaluate` in lines 9-10), and create HTML fragments (e.g. line 21),
— a handler can also change the DOM structure at will by injecting HTML fragments (e.g. the `BookBurroPanel`). In the example, the output is injected at a point identified by an XPath expression on the underlying DOM structure (i.e. the injection point). DOM functions are used for this purpose (e.g. `appendChild` in line 23 and 36),
— this script is associated with a URL pattern that denotes the pages to which the script applies. This is specified through the `@include` annotation (line 3).

DOM tree (e.g. load, click). Event payloads provide the data to feed script handlers which, in turn, update the DOM tree. The script is outlined in Figure 3. The process goes as follows:

— interacting with a page triggers UI events (e.g. load),
— the script reacts to this event by triggering a handler (lines 6-39). The association between an event and a handler (a.k.a. event listener) is achieved through the `addEventListe)r` function (line 6),
— a handler can access any node of the page (using DOM functions such as `document.evaluate` in lines 9-10), and create HTML fragments (e.g. line 21),
— a handler can also change the DOM structure at will by injecting HTML fragments (e.g. the `BookBurroPanel`). In the example, the output is injected at a point identified by an XPath expression on the underlying DOM structure (i.e. the injection point). DOM functions are used for this purpose (e.g. `appendChild` in line 23 and 36),
— this script is associated with a URL pattern that denotes the pages to which the script applies. This is specified through the `@include` annotation (line 3).

Fig. 3. BookBurro in JavaScript (partial view).
From a producer perspective, development and maintenance are main focuses. Development wise, JavaScript is a powerful, expressive language but also cumbersome to program. APIs alleviate this burden (e.g. jQuery [Resig 2006], Chickenfoot [Bolin et al. 2005]), though users frequently end up writing large chunks of companion JavaScript code. From a maintenance perspective, scripts are subject not only to their own evolution but also vulnerable to changes on the underlying website. The problem is that websites are reckoned to evolve frequently. Back to our sample case, if the Amazon website is upgraded, all “the screen scrapping” can fall apart. For instance, BookBurro first retrieves the book’s ISBN from the current page, and next, injects the BookBurroPanel at a certain location. This is normally achieved through XPath expressions (line 9). If Amazon pages are changed then, BookBurro’s XPath expressions could no longer recover/identify the right DOM node. Therefore, augmentation scripts are specially prone to maintenance.

From a consumer perspective, two main concerns arise. The first is security. The stealing of cookies/credential is unfortunately not so rare (for a real example refer to 4). The second concern is reliability. Perfectly valid scripts can stop working due to script collision, i.e. the simultaneous access to the same DOM node by two different scripts. The very same Web page can be subject to different augmentations. Amazon is a case in point. To date, 268 scripts are reported to be available for Amazon at userscripts.org. If you are a regular Amazon visitor, it is likely you have several scripts installed. These scripts will be enacted simultaneously when you visit Amazon. It is important to notice that script execution is not in parallel but in sequence, i.e. scripts are launched in the order in which they were installed. This implies that the first script acts on the original DOM tree, the second script consults the DOM tree but once updated by the first script, and so on. The problem is that programmers develop scripts for the original DOM, being unaware of changes conducted by other companion scripts. This can end up in a real nightmare where code developed by different authors with different aims, is mixed up together with unforeseen results. Even worse, the final DOM tree can be dependent on the order in which scripts are enacted! The larger the set of (companion) scripts, the higher the likelihood of collisions. Hence, script collision raises scalability concerns on the current state of the technology.

The bottom line is that the expressiveness brought by a general-programming language such as JavaScript comes at the price of reliability and maintenance. Consumption also suffers from this freedom. Even fully-tested scripts (e.g. the Skype button) can collide when enacted simultaneously with scripts that access the same DOM regions. The problem is that these errors are detected (and suffered) by consumers with little help from producers who can hardly foresee the context in which their scripts are to be run. This potentially high cost of maintenance and consumption, compromises the “end-userness” of JavaScript for Web Augmentation. Next section trades JavaScript expressiveness for reliability and learnability.

4. WEB AUGMENTATION: CARING FOR PRODUCERS

This section focuses on conditioned producers, i.e. motivated end users whose scripts can be potentially consumed by others. The target profile is that of hobby programmers with no knowledge of JavaScript. The producer should know about URLs, and a bit of HTML is recommended though not strictly necessary. Users do not need to know either JavaScript or any other programming language.

The trade-offs exist between expressiveness, freedom, and being general-purpose on the one hand, and usability, learnability, control, and being domain-specific on the other [Klann et al. 2006]. The challenge is to abstract away from JavaScript by identi-
fying recurrent abstractions in augmentation scripts. Next, we restrict the full expressiveness of JavaScript to a set of patterns for augmentation which is finally captured through a DSL. This section is about maximizing expressiveness without compromising reliability and learnability.

Expressiveness requirements are first captured through domain analysis, and next, framed by the target audience (e.g. conditioned prosumers) [Mernik et al. 2005]. A main output of domain analysis is the feature diagram [Kang et al. 1990]. A feature diagram represents a hierarchical decomposition of the main concepts (i.e. features) found in the domain. The diagram also captures whether features are mandatory, alternative, or optional. Figure 4 depicts the feature diagram for the domain “Web Augmentation”. The diagram states that “a Web Augmentation script” includes a scope (that sets the ambit of the augmentation), data extractors (for variable assignment), an augmentation lever (that triggers the augmentation as such), the enactment of URL requests, and the insertion of its output in the augmented page. These are the “recurrent abstractions” to be potentially captured as primitives in Sticklet.

These features do not come out of the blue but as repeating concerns that are coded time and again. For comprehension purposes, it is convenient to go back to Figure 3, and to identify these abstractions on the raw code of BookBurro. This permits to better appreciate the abstraction effort (bold stands for features in Figure 4):

— **Scope** (line 3). It sets the context where the augmentation occurs: hosting sites that hold certain data. Hosting sites are captured by the Greasemonkey’s @include metadata that keeps a URL pattern (line 3). The content of the site is verified through the extractors.

— **Extractor** (lines 9-19). The extractor locates the content to be obtained from the page. They are realized as brick selectors (e.g. the existence of the string “ISBN-10” in a certain position in the DOM tree, lines 9-10) or content extractors (e.g. a string following a certain pattern, line 18).

— **Lever** (lines 21-26). Matching the scope might not directly trigger the augmentation. Enacting the augmentation might first require the user to undertake some actions (e.g. clicking a button, passing the mouse over a certain page region, etc.). These actions are realized as DOM events on an HTML object: the augmentation lever. An
augmentation lever is characterised through three elements: a lever event, which is raised by the user on interacting upon a lever element which is in turn, placed at a given lever position. For BookBurro, the lever is (onClick, link, after the ISBN).

---

**Augmentation Provider** (lines 28-39). Greasemonkey’s API function `GM_xmlhttpRequest` allows user scripts to get and post data to any site. These data can be retrieved from other Web applications (hence, returning HTML documents) as well as from Web services (which output XML or JSON). This is one of the main enablers of Web Augmentation. But also a main headache. Service fulfilment involves parameter construction, service enactment and error handling. **Recovery handling** for communication pitfalls might need to be considered.

---

**Rendering Directives** (lines 34-36). The outcome of service enactment is mashed up into the hosting site. Three output formats are considered: HTML, XML and JSON. This implies both to select the desired data from the service outcome as well as to provide presentation directives for these data.

Figure 4 outlines the main abstractions. Next, these abstractions are realized in a language by looking into variabilities and commonalities in the feature diagram. Variable parts must be specified directly in or be derivable from DSL expressions. In the first case, the variants become DSL constructs. On the other hand, some alternatives can be hardwired into the DSL engine as heuristics. Being heuristics, they might fail and hence, they are not as reliable as if provided by the user. The upside is that they simplify the user’s life, hence, improving learnability. For Sticklet, we decided outcomeRendering and recoveryHandling to be hardwired into the interpreter (rationales later). The rest of features are set by the user through the DSL. We then need to come up with a set of constructs to specify the rest of the augmentation features.
At this stage, it is most important to devise a metaphor that facilitates the understanding of the DSL constructs. Hence,

we regard the Web as a *wall* to be decorated with *stickers*. Stickers are *notes* (i.e. *HTML* fragments) dynamically obtained from other websites.

Web Augmentation is then phrased as fixing *sticky notes* into *walls*. The pair *(wall, sticky note)* conforms an augmentation unit: a *sticklet*. Hereafter, the term “*sticklet*” will denote the engine whereas “*sticklet*” will refer to a *Sticklet* expression (a.k.a. an augmentation script). As an example, Figure 5 provides the *sticklet* counterpart of the *BookBurro* script (italics denote DSL constructs). The expression reads as follows. *WhenOnWall* of *Amazon*, *SelectBrick* and *ExtractContent* *As* the *isbn* variable. Next, *InlayLever* *At* a given *brick*. Finally, *OnTriggeringLeverBy* a certain event, *LoadNote* from BookByte, *SelectBrick* price and, eventually, *StickNote*.

Figure 6 provides the abstract syntax. A *StickletBox* comprises a set of *Sticklets*. A *Sticklet* includes the scope (*WhenOnWall*), the extractors (*SelectBrick*, *ExtractContent*, *As*), the augmentation lever (*InlayLever*, *At*, *OnTriggeringLeverBy*), the augmentation requests (*LoadNote*), and the rendering of the augmentation (*StickNote*). This abstract syntax is realized through a concrete syntax, either graphically or textually. We opted for a textual DSL (see Subsection 6.1 for the rationales). The *Sticklet* BNF grammar can be found in the Appendix. A presentation describing *Sticklet* is available at [http://www.slideshare.net/onekin/sticklet-a-language-for-web-augmentation-16504935](http://www.slideshare.net/onekin/sticklet-a-language-for-web-augmentation-16504935). Next, we introduce *Sticklet* syntax through examples.
To see the *BookBurro* sticklet at work just type http://tinyurl.com/bewql5e, and you will be guided through the installation process. Otherwise, follow these steps *(order matters)*:

1. install the *Sticklet* validator: https://addons.mozilla.org/addon/sticklet/.
2. install the Greasemonkey weaver: https://addons.mozilla.org/addon/greasemonkey/.
4. install the *Sticklet* scripts by drag&dropping the scriptName.user.js file into the browser.
5. edit *Sticklet* scripts using any textual editor (e.g. *Notepad*), the Greasemonkey editing facilities, or the inline editor (see later).

### 4.1. Sticklets

We begin with *BookBurro* but now specified as a *sticklet* (see Figure 5). The constructs of the DSL include: *walls, bricks, notes and levers*.

**Walls** (line 9). A *wall* comprises those websites whose URLs match a given regular expression (*WhenOnWall* clause). They can be regarded as “views” upon the existing websphere. The scope of the *sticklet* is defined by its *wall* as well as by the existence of some *bricks*. For our sample problem, the *wall* expands along those *Amazon* pages that hold an ISBN *brick*.

**Bricks** (lines 10-12). They are named nodes upon HTML documents which are worth singularizing for either data extraction, scoping or layering purposes. A *brick* holds (1) an *XPath* to pinpoint the node (*SelectBrick* clause), (2) a regular expression to extract the node’s content (*ExtractContent* clause), and (3), the *brick*’s name (*As* clause).

**Notes** (lines 15-18). They are expressions that combine text and *bricks* (*StickNote* clause). *Bricks* can be obtained from the *wall* as well as from URL-addressable Web applications (*LoadNote* clause, line 15). For the sample problem, a request is made to *BookByte*. URL parameters are obtained from previously extracted *bricks* (e.g. $isbn). The outcome is used to pinpoint a new *brick*: $price. Finally, *bricks* from different sources might be mashed up into a *StickNote*. *Notes* are framed by a decorator. *Notes* can be dragged around, expanded to fit, minimized, or just closed. *Notes* might be readilily stuck as soon as the user enters the *wall*. However, this is not always the desired behaviour. Users might be looking at *Amazon* with no intention of buying a book. Readily sticking *notes* could lead to cluttered pages, being contra-productive and inefficient by forcing the enactment of the *sticklet*’s *note* with no purpose. Therefore, rendering a *note* might require an additional user interaction: acting on a *lever*.

**Levers** (line 13-14). They permit to obtain *notes* on demand. *Levers* are named after the *sticklet* name (e.g. “*Price at BookByte for $isbn*”) where variables (e.g. $isbn) are resolved at runtime. The specification of a *lever* includes the type and the position and the event. *Levers* can be of two types: links or buttons. *Levers* are positioned according to *bricks* (e.g. *BookBurro*’s *lever* is placed *after* the *brick* $isbn). Options include “after”, “before” and “upon”, where the latter replaces the *brick* by the *lever*. On acting upon the *lever* through a DOM event, a URL petition is conducted (i.e. the action). For *BookBurro*, the *lever*’s event is a *click*. On clicking, the *LoadNote* clause takes the control. In this way, *notes* might not be readily rendered but requested through an explicit user interaction.

Two important remarks about *bricks*. First:

**Bricks** denote entities of interest within the *wall*. These entities can be atomic or compound.
For example, our sample Amazon page contains an instance of the entity ISBN-10, but several entities of type CustomerReview (i.e. different reviews are contained in the page). Worth noticing, entities (i.e. bricks) can be atomic or compound based on their HTML representation.

For instance, the brick $isbn is atomic: the associated XPath expression returns a number. By contrast, the entity CustomerReview also in the Amazon page, is compound, i.e. its associated XPath returns an HTML fragment which comprises a score, a headline, a description, a reviewer and a review date. Since an entity is realized as an HTML fragment, XPath can be used to obtain bricks out of these compound bricks. An example follows:

\begin{verbatim}
SelectBrick("//tr...").ExtractContent("(.*)").As("$customerReview").
SelectBrick("$customerReview//span...").ExtractContent("\d").As("$score").
SelectBrick("$customerReview//div...").ExtractContent("by (.*)").As("$reviewer")
\end{verbatim}

The second remark is about the three-fold role played by bricks:

**Bricks can serve to (1) extract data from pages, (2) pinpoint locations for lever positioning, and (3), determine the number of sticklet instances.**

The latter requires further explanations. Operationally, sticklets can be regarded as triggers (e.g. on loading an Amazon page with an isbn, inlay a lever). The implicit event is not “on loading an Amazon page” but “on obtaining an ISBN from an Amazon page”. This is a paramount difference from the perspective of the operational semantics of Sticklet (see Subsection 4.7). It implies that if the Amazon page does not contain an ISBN, the sticklet does not apply. Likewise, if the Amazon page had contained several ISBNs then, distinct sticklet instances would have been fired (hence, placing different notes by each ISBN-10). This would have been the case if CustomerReview rather than ISBN-10 had been the brick. Since an Amazon page can hold different CustomerReviews then, a sticklet instance would have been triggered for each CustomerReview. This in turn, implies that ten notes would have been generated and placed by the lever. Had both bricks been introduced, a sticklet instance would have been triggered for each combination of [CustomerReview x isbn]. For our sample case, this means ten instantiations since the page for the sample book holds an isbn (i.e. a single node satisfies the XPath associated with $isbn) and ten reviews. This in turn, implies that ten notes would have been generated and placed by the lever.

Due to this operational semantics, users should be careful when defining the granularity of their bricks. For instance, CustomerReview need to be introduced even if the interest is only in part of its information (e.g. the score and the reviewer). Explicitly naming this entity indicates that score and reviewer are not two independent notions but that they belong to a higher concept (i.e. customerReview) in whose terms the sticklet is described: the number of customerReview (and neither scores nor reviewers) determines the number of times this sticklet is to be triggered (more in Subsection 4.7).

4.2. StickletBox

A sticklet accounts for a pair (a wall, a sticky note). However, a single sticklet might not be enough. For instance, the BookBurro script might be conceived as enhancing not just Amazon but a set of online bookshops (e.g. Amazon, BookByte, Powell). The price-comparison note is available for any of these bookshops. Since a sticklet supports a pair (a wall, a sticky note), the BookBurro functionality requires six sticklets: (Amazon, BookByte's price), (Amazon, Powell's price), (BookByte, Amazon's price), (BookByte, Powell's price) and the like. This grounds the notion of stickletBox:
A stickletBox is a set of sticklets that stand for a meaningful unit of augmentation. Sticklets can be added/removed at any moment. Sticklets are self-contained, no coupling exists among sticklets kept in the same stickletBox.

A stickletBox permits the very same wall (e.g. Amazon) to receive notes from different websites (e.g. prices at both BookByte and Powell). This raises the question of whether those notes should be obtained simultaneously (and hence, displayed in the same note) or not. This is regulated by levers. Levers are characterized through bricks (e.g. $isbn$).

Bricks with both the same name and associated XPath denote the very same position in the wall. Therefore, notes from different sticklets but attached to namesake bricks are simultaneously obtained and rendered.
BookBurro provides an example (see Figure 5, lines 13 and 25). Two sticklets are introduced. Both point to the same brick: same name (i.e. $isbn), same XPath. Therefore, their outputs are simultaneously rendered in the very same note. By contrast:

**If two notes account for different information needs, then, two differentiated levers permit to resolve these needs separately.**

“LibraryReservation for $isbn4reservation” illustrates this case (Figure 7). We want to know whether the book at Amazon is available at the Library of Manchester University, and reserve it.\(^5\) The augmentation captures the “reserve” button at this website, and sticks it on the Amazon page. Book identification is also achieved through the ISBN. Hence, the sticklet introduces a brick based on the same XPath as before. However, since this need is not geared towards purchasing the book, the brick’s name is changed: $isbn4reservation (line 13). This generates a different lever (in this case, a button) also located by the ISBN.

### 4.3. The Issue of Entity Linkage

Sticklets contextualize data from different websites into a single workspace: the wall. Contextualization implies the existence of a “sharing notion” between the wall (e.g. Amazon) and the note providers (e.g. BookByte). In the previous example, this notion was the ISBN: extracted from Amazon as a brick, and communicated to BookByte as a URL parameter. However, this is not always the case. An entity might exhibit distinct representations. A book can be denoted by the ISBN, the pair (title, author), an ad-hoc code, or even the book’s cover can be the only reference to a book. As an example, consider the www.walmart.com website. Walmart also sells books. However, Walmart’s URLs are not based on ISBNs but on an internal code. For instance, the URL for the book used as an example is http://www.walmart.com/ip/13443765, where the ending number has nothing to do with this book’s ISBN. This raises a mismatch between how the entity is represented in the wall (e.g. Amazon) and how the entity is captured in the URL parameter (e.g. Walmart).

This problem also arises in databases (known as Record Linkage) when two data sets need to be joined and they do not have a unique database key in common (e.g. passport number vs. national insurance number) [Winkler 2006]. In the Semantic Web, where resources are described through URLs, they also encounter the so-called Coreference problem, i.e. ascertaining where two distinct URLs stand for the same entity [Jaffri et al. 2007]. The use of mediate ontologies and shared resources such as the DBpedia [Bizer et al. 2009] can help to provide a common ground to facilitate integration. In ontology mapping, the challenge is to discover automatically alignments between entities described in different ontologies, exploiting lexical similarities, lattice structure or instance classification learning techniques [Sánchez et al. 2010]. These approaches tend to be time consuming and, in some cases, imply user intervention. However, we strive to minimize both user intervention and elapsing times (remember, low available attention). As a result, **Sticklet does not provide any module for “entity linkage”** but resorts to mapping and search.

**Linkage through searching.** Since ISBN-linkage does not work, we will mimic what users would do: go to Walmart and conduct a local search for e.g. the book’s title. This implies to access programmatically the search facilities of the Walmart website. Fortunately, the OpenSearch Discovery initiative has already standardized this process [A9 and Amazon 2005]. An OpenSearch description document can be used to describe the web interface of a search engine. This description holds parameterized URL templates that indicate how the search client should make search requests (the

\(^5\)You can find this service for our book example at http://catalogue.library.manchester.ac.uk/items/2049288.
Fig. 8. Entity linkage through searching: Walmart is queried about rather than requested for a specific URL.

This technology is used in Sticklet to resolve entity linkage by extending the semantics of the clause LoadNote. So far, LoadNote keeps a URL expression which is parameterized by bricks extracted from the wall. Now, LoadNote can also be instructed to transparently conduct an open search by turning the “http” protocol into the “osearch protocol” (see Figure 8, line 19):

LoadNote can be commanded to search into the note-provider website to find the wall-notion counterpart.

This process is conceived as a kind of protocol in so far as it initiates a conversation between the Sticklet agent and the guest website (e.g. Walmart). The “osearch protocol” commands Sticklet to go to www.walmart.com, locate the OpenSearch document as a <link> in the source page, recover the <Url> element, construct the search request using the associated template (as an attribute of the <Url> element), process the output (an HTML page), and finally, come up with the URI of the sought resource. Shouldn’t the website contain an OpenSearch document then, Sticklet goes back to the http pro-

Making explicit how to query a website permits to introduce “custom search engines” in browsers [eTeanga 2010]. When viewing an HTML page that includes the <link> tag above, browsers can either automatically collect the OpenSearch file (e.g. Chrome) or highlight some icon of the browser search box (e.g. Firefox) that permits users to explicitly add the current site as a source for their queries. You can check this out by navigating to www.walmart.com. On detecting <link rel="search"> in the page source, Firefox faintly highlights the little arrow of its search bar. Click on this arrow, and observe how a new menu item prompts to include Walmart as a new custom search engine.
Fig. 9. Entity linkage through searching. Clicking the lever makes Sticklet initiate a conversation with Walmart: (1) Walmart is queried about the title “JavaScript: The Definitive Guide” using OpenSearch; (2) Sticklet looks for the URL of interest among the URLs held in the returned page; (3) Walmart is requested based on this URL; (4) Sticklet extracts the price from this second page. This navigation is transparent to the user.

tocol: loads the URL (without parameters), attempts to locate a search box in the returned page, and finally, feeds the search box with the search parameter (e.g. $title). A final hardwired strategy is to conduct the search in Google (i.e. “site:www.walmart.com AND $isbn”) and retrieve the first link.

No matter the way, Sticklet retrieves a page. This page can stand for the sought resource (and then, the process ends) or deliver a list of resources that match the query parameters. This is the case when the osearch protocol is used for Walmart for the title “JavaScript: The Definitive Guide”. Figure 9 shows the output. In this case, Sticklet applies a set of heuristics to ascertain the URI of the sought resource. Since LoadNote has to return a single URI, the algorithm focuses on those DOM nodes that hold a URI. Next, the algorithm identifies those text nodes containing text values used in the query on the assumption that the values used in the query will typically appear
Fig. 10. Entity linkage through mapping: (1) clicking the lever, (2) causes a GoogleMaps to be loaded which contains the $zip; (3) this zip is used to build a request to worldweatheronline which holds the $weather note; finally (4), this HTML fragment is sticked. Only the final step is visible to the user.

with higher probability in the list of results than in other lists of the page. Once the list of books is singularised from other lists in the Walmart page, it rests to identify the sought resource within this list. If the URI nodes at Walmart are accompanied by additional information (sibling nodes), then Sticklet will compare those sibling nodes as a unit with the compound brick that holds the query parameter (e.g. $title and $book, respectively), should such brick be available. If this process does not filter a single URI then, Sticklet will return the URI in the first position. This process is built into Sticklet as the semantics of the "osearch" construct.

It could be argued why not to conduct the query in terms of $book in the first place rather than in terms of a book property such as $title. The reason is that $book might contain a broad range of data other than that used by the user in the query. In the example, the user characterizes books in terms of title and isbn, thought book nodes (as pinpointed by the associated XPath expression) encompass a wider range of other data (e.g. authors). Querying by $book (better said, the content of $book) might be too stringent, causing the query to output no result at all. Therefore, users decide the properties to search for, and next, Sticklet applies the heuristics above to filter out the sought URI.
Linkage through mapping. This approach resorts to intermediate websites to act as mediators.

Mediator websites can be introduced to map the notion as capture on the wall into its counterpart in the note-provider website.

Figure 10 provides an example of a website about camping spots: www.lat-long.com. Before camping, we would like to know the weather forecast at the selected spot. Since this information is not included, we extend this website with a weather-forecast note from www.worldweatheronline.com. The shared notion is that of location. Lat-long captures locations through geo coordinates. However, worldweatheronline can be queried using different criteria (e.g. the zip code, the city name) but not through geo coordinates. Fortunately, GoogleMaps can act as a mediator. Figure 11 provides the sticklet code. Mediators are introduced as note providers (lines 17-18).

Generally, search engines can act as universal mediators in so far as they tend to accurately identify websites based on search terms. For sticklets, these search terms are bricks obtained from the wall. Consider a film listing (e.g. http://kedin.es/cartelera-cine) as a wall where to place sticklets with the movies’ ratings obtained from www.imdb.com. Figure 12 depicts the wall before and after the augmentation. The shared entity is that of “movie”. The wall captures movies through their titles while IMDB’s URLs use an ad-hoc code for movie identification. The mapping can be resolved through Google⁷ (see Figure 13). First, the brick $title is extracted from the wall (lines 10-11). On clicking the lever (i.e. the “What IMDB tells about” link), a Google search is initiated with focus on the IMDB site, and passing the title as an additional parameter (line 14). Google returns a list of entries from IMDB. The first one serves to extract the brick $code (lines 15-16). Next, we navigate directly to the rating page (line 17), extract the rating chart, and place it on the note (line 20). Worth noticing are the multiple instantiation of the $title in the film listing. A lever is introduced for each instantiation of the $title in the wall.

⁷It could have been also possible to use the osearch approach.
Fig. 12. Augmenting the kedin film listing. The entity to be augmented (i.e. movie) appears several times. Each appearance causes the inclusion of the augmentation link (i.e. What IMDB tells about). The figure shows the note for “The Master” movie.

```javascript
1 // ==UserScript==
2 // ... 
3 // @onekin:sticklet
4 // @sticklet:twitter
5 // @sticklet:facebook
6 // ==/UserScript==
7 StickletBox({
8   Sticklet("What IMDB tells about?").
9     WhenOnWall("kedin.es/"cartelera-cine").
10 SelectBrick("/span[@class='movie_title' and position()==1]").
11 ExtractContent("(.+)"), As("$title"), // text
12 InlineLayer("link"), At("after", "$title").
13 OnTriggerLayerBy("click").
14 LoadNote("http://www.google.com/search?q=imdb.com+$title").
15 SelectBrick("/li[@class='g']/cite").
16 ExtractContent(".com/title/(.+)"), As("$code"), // text after
17 LoadNote("http://www.imdb.com/title/$code/ratings").
18 SelectBrick("/div[@id='tn15content'").
19 ExtractContent("[^\."\s]+"), As("$rating"), // text
20 StickNote("$rating"));
```

Fig. 13. Entity linkage through mapping. Google acts as a mapper from the movie’s title to the movie’s code in IMDB.

ACM Transactions on the Web, Vol. V, No. N, Article A, Publication date: January YYYY.
4.4. The Issue of XPath Complexity

XPath is outside the competences of our target audience. Assistance is required that hides this complexity from producers in writing their sticklets as well as from consumers in understanding third-party sticklets. The question is how to specify an extraction pattern without knowledge of XPath or regular expressions, or understanding of HTML. Answers include the use of:

— heuristics, that permit to refer to buttons, links, and other web page elements in terms of nearby text (termed as “text constraint patterns”). In this way, users do not use XPath to pinpoint the desired data but just indicate the nearby text (e.g. the data that is after the “ISBN” text), and let the system guess the right location. This approach is illustrated by Chickenfoot [Bolin et al. 2005] and CoScripter [Leshed et al. 2008] (Section 8 illustrates the BookBurro example using Chickenfoot). For our purposes, this approach brings two main benefits: learnability (for producers)
and understandability (for consumers). The downside is efficiency. Text constraint patterns imply processing the whole document on the search for the patterns every-time a new page is loaded. For complex pages/patterns, such process can incur in noticeable delays every time the script is run.

— programming-by-example, where users first highlight elements of the web page which serve to infer the matching pattern (e.g. the XPath expression). Systems reform [Toomim et al. 2009], Karma [Tuchinda et al. 2011] or MashMaker [Ennals et al. 2007] use programming-by-example. For our purposes, this approach brings efficiency since the inferring algorithm is only run at definition time. This option eases production but still leaves consumers to face XPath expressions.

Sticklet explores a hybrid approach:

**Sticklet permits to substitute XPath expressions by the value “assisted”.** This makes Sticklet differ till run time the binding of these clauses, and resolve them visually.

As an example, consider the BookBurro script but now using assisted as the value for SelectBrick and ExtractContent clauses. The semantics of assisted works as follows (see Figure 14):

1. on loading the Amazon page, the sticklet is enacted. The engine detects that the sticklet is not fully resolved, i.e. it contains some assisted-valued clauses. Hence, the engine layers a panel for the assisted editing. The panel includes: a progress tracker, the regenerated sticklet (“Sticklet after edit”) and the current sticklet (“Sticklet before edit”). The progress tracker provides an ordered way to resolve each of the unbounded clauses. For the sample problem, this tracker provides four stages, one for each unbounded clause, and finally, the regenerate option.

2. the user clicks on the first unbounded clause (e.g. SelectBrick $isbn). This makes Sticklet intersperse a grid-like structure on top of the current DOM tree. As the user moves the cursor around the screen, the DOM node under the current cursor location is highlighted. By clicking, the user feeds the inferring algorithm with the selected node, and Sticklet highlights all the nodes that fulfil the extraction pattern generated so far. For the ISBN, there is only a single node in the hosting page that fulfils this notion, and hence a single interaction might suffice. However, the augmentation might impact different nodes of the hosting page. The brick $customerReview provides an example. This brick might be instantiated several times throughout the page. This requires the user to pinpoint distinct reviews till the correct extraction pattern is abstracted by the inferring algorithm. Heuristics from XPath generation are inspired in work described in [Paz and Díaz 2010; Álvarez et al. 2010; Lingam and Elbaum 2007].

3. Next, the following node of the progress tracker becomes active (i.e. ExtractContent). Now, the outcome is a regular expression. The user selects the content of interest, and Sticklet obtains an expression that matches this content. When bricks act as filters, the content might be the filtering criteria (e.g. books whose price is 103). That is, the expression is a constant (e.g. “103”). This makes the engine look for nodes whose content is “103”. If this is the desired behaviour, just skip the ExtractContent clause.

---

8Sticklet abstracts away from absolute paths into a relative path that strives to capture the essence of the rendering of $customerReview (e.g. "/tr[2]/td[3]/div/a"). Sticklet highlights in the canvas all the nodes that account for the so-obtained XPath. Should some node be missing, the user can click on the missing node and let Sticklet regenerate the XPath expression. Once all nodes of interest are highlighted, the user clicks in the next step of the progress tracker, and the so-generated XPath becomes the value of the SelectBrick clause as visualized in the “Sticklet after edit” panel (see Figure 14).
tractContent step. This makes the content of the current node become the constant value of ExtractContent.

(4) Finally, regenerate the script, i.e. the script is automatically updated and re-installed.

The "assisted" option could suggest that the explicit introduction of XPath is no longer needed. However, two scenarios advice to keep this option open. First, the inferring algorithm might fail to extract the correct data. Both Chickenfoot and reform warn about the heuristic nature of this process. Keeping this option open permits at least to ask for assistance to an XPath expert. Second, some data might not have a rendering counterpart, hence, no way for the user to pinpoint this data. A common example is extracting URLs. URLs tend to be provided as attributes of HTML anchors. The anchor's content is what you see while the anchor's URL is what you might want. In this case, "assisted" is of little help. Therefore, Sticklet keeps the possibility of explicitly providing the XPath expression open.

So far, we address the complexity of XPath from the producer perspective. The "assisted" approach hides this complexity for producers but consumers are still exposed to XPath expressions. Understandability is a major benefit of text constraint patterns (e.g. the data that is after the "ISBN" text) that we lost when moved to XPath. In an attempt to overcome this limitation,

Sticklet automatically generates a comment by each XPath expression that phrases in natural language terms the XPath and the regular expression.

Figure 5 provides an example (lines 12 and 24). In this way, Sticklet strives to bring the best of both worlds. Using programming-by-example, Sticklet obtains efficient and accurate XPath expressions without imposing a major burden on producers. Generating pattern-like comments, Sticklet attempts to be as understandable as the alternative of directly providing text constraint patterns.

A final comment on maintainability. Unlike text constraint patterns, XPath expressions are fragile: changes on the structure of the underlying Web page can make the XPath expression stop working. This is true. But the overhead of re-generating the script is affordable: edit the broken sticklet, substitute the XPath expression by "assisted", and finally, regenerate the xpath. We believe this maintainability burden compensates for avoiding working out the location of data everytime the script is run, as in text constraint patterns.

4.5. The Issue of Non-HTML Sources

Notes can be obtained from sources other than HTML pages. RSS feeds or URL-addressable programmatic interfaces deliver data-centric XML or even, JSON [Crockford 2006]. Being an agent, Sticklet can consume these documents. For instance, the previous Lat-long camping example could have resorted to an XML-based service to obtain the ZIP out of the geo coordinates. Lines 17-18 in Figure 11 might be replaced by:

```
LoadNote("http://maps.google.com/maps/geo?output=xml&q=$lat,$lon").
SelectBrick("/PostalCodeNumber").ExtractContent("(\d+)").As("$zip")
```

The problem is that XML/JSON documents are thought for agent consumption rather than human consumption. Human consumption implies reading and interacting via HTML documents. Specifically, Sticklet expects user interaction to select bricks.

---

9So far, the algorithm can learn from positive but not negative examples as suggested in [Toomim et al. 2009].
Note rendering. Default rendering can be supplemented with HTML directives. The figure depicts (a) the default rendering (i.e. no HTML tags) and (b), the HTML-enriched outcome as specified in the sticklet (lines 22-25). The user-provided HTML directives prevent the note $AmazonAverageReview from overlapping with the $GoodreadsReview note.

and render notes. If the source of the note happens to be an XML/JSON document, Sticklet needs first to convert this document into HTML.

For XML/JSON sources, Sticklet applies basic rendering templates to the returned XML/JSON document.

In this way, users select/read bricks in the very same way regardless of whether the original representation was HTML, XML or JSON.
4.6. The Issue of Note Rendering

*StickNote* commands the rendering of a *note*. A *note* is an expression that combines text and *bricks* (i.e., *HTML* fragments) from potentially different websites. Each website has its own rendering (i.e., set of *CSS*). The question is how this bulk of disparate *HTML* fragments can deliver “a harmonious note”. This endeavour requires *HTML* skills. We decided to remove such burden from the user, and hardwire some rendering heuristics into *Sticklet*.

*Sticklet* holds a set of heuristics that provide “good-enough rendering” for sticky notes. In addition, users can supplement the *note* with *HTML*-sanitized markup.

Let’s augment *Amazon* with reviews from *www.goodreads.com*¹⁰ (see Figure 15). Specifically, the *note* gathers the following *bricks* (lines 11-16 and 20-21): the book title (**$title$**), the average customer reviews from *Amazon* (**$AmazonAverageReview$**) and reviews from *GoodReads* (**$GoodreadsReview$**). These *bricks* can be provided in quick succession without *HTML* ornaments. In this case, *Sticklet* will decide the best rendering (see Figure 15 (a)). Heuristics are inspired in those applicable for the automatic rendering of pages through small-screen devices such as mobile phones [Buyukkokten et al. 2002; Xiao et al. 2009]. As a general norm, *bricks* are rendered along the *CSS* directives of their source websites (in this example, those of *Amazon* and *GoodReads*).

We are aware these heuristics fall short to account for sophisticated results. At worst, the rendering looks like a simple list with no additional ornament. But the main argument here is that *Sticklet* is designed for *HTML* ignorants. Amateurs love to do things by themselves while accepting good-enough results. Forcing users to provide the rendering by themselves would sacrifice better appearance for self-sufficiency.

All in all, Sticklet also permits to introduce HTML directives. Similar to the editing of wiki articles (and on similar grounds), notes can contain basic HTML tags. Due to security reasons, HTML is sanitized. This basically means that JavaScript is not permitted (restrictions are similar to those of Wikipedia). Two remarks about bricks. First, bricks are HTML fragments whose CSS classes are inherited from their source websites. Those classes can be overridden by HTML directives explicitly given in the note. Second, bricks which can potentially be instantiated several times (e.g. $\text{GoodreadsReview}$) are regarded as lists, and hence, they should be enclosed using a directive for lists (e.g. `<ul>`, see line 25). Otherwise, all instances are rendered as a single row.

4.7. The Operational Semantics of Sticklets

This subsection outlines the operational semantics of sticklets interpreted as sequences of computational steps. These sequences then are the meaning of the “sticklet” construct. Augmentation proceeds along two states (see Figure 16): the “lever available” state and the “panel available” state. Transitions proceed as follows:

— on page loading, the URL of the loaded page is checked against the regular expression of the WhenOnWall clause. If met, the engine raises a matching event occurrence for each combination of brick instances that fulfil the matching conditions set by the bricks. Hence, the event payload is a combination of the brick instances. If you have bricks $a$, $b$ and $c$ which are matched in the current wall $2$, $3$ and $5$ times, respectively, then, the number of matching event occurrences will be $30$: occurrence$(a_1,b_1,c_1)$, occurrence $(a_2,b_1,c_1)$, and so on. Each matching event occurrence gives rise to a sticklet instance (represented as an AND state in Figure 16).

— on rising a matching event, the engine moves to the “lever available” state. This causes the rendering of the lever along the behaviour defined in clauses InlayLever, At and OnTriggeringLeverBy,

— on rising the lever event (e.g. mouse over, click, etc.), the LoadNote service is enacted. The system moves to the “panel available” state where the note is rendered,

— at any moment, page unload causes the current page stop displaying. This ends the augmentation.

Worth noticing is the parallelism behind this semantics. First, matching event occurrences are handled in parallel as denoted by the AND states in Figure 16. If you have 30 matching occurrences then, 30 transitions to the “lever available” state will happen. Second, some of the previous transitions might entail the same lever (e.g. sticklets “Price at BookByte for $\text{isbn}$” and “Price at Poweel1 for $\text{isbn}$” involve the same lever). In this case, the very same lever event is shared among these two AND states. Raising this event causes both states to transit to the “panel available” substate, and jointly render their outputs as a single note. In short,

The rule-like semantics of sticklets go beyond improving modularity to account for parallelism and atomicity. Efficiency wise, the speed of a stickletBox augmentation is that of its slowest lever/matching transition.

The latter entails that if collecting the price from ten online bookshops that share the same lever, clicking on this lever will cause nine parallel HTTP requests (the tenth bookshop is the wall). The note is constructed gradually as answers arrive. This means that the order in which prices are displayed in the note might vary depending on the traffic load of the sites.
Table I. Sticklet built-in error handling strategies.

<table>
<thead>
<tr>
<th>Error Type (HTTP code)</th>
<th>HTTP Error Handling Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client-side Recoverable (408, 413)</td>
<td>Retry in a short period</td>
</tr>
<tr>
<td>Client-side Unrecoverable (400, 404-6, 409-11, 414)</td>
<td>Error Sticky Note</td>
</tr>
<tr>
<td>Client-side Authorization (401-3)</td>
<td>Notify and disable the sticklet</td>
</tr>
<tr>
<td>Server-side Recoverable (500, 502-4)</td>
<td>Retry in a short period</td>
</tr>
<tr>
<td>Server-side Unrecoverable (501, 505)</td>
<td>Error Sticky Note</td>
</tr>
<tr>
<td>Credential Required (no code returned)</td>
<td>Error Sticky Note</td>
</tr>
<tr>
<td>&quot;OSearch protocol&quot; retrieves the empty result</td>
<td>Error Sticky Note</td>
</tr>
<tr>
<td>Brick description is not met</td>
<td>Error Sticky Note</td>
</tr>
</tbody>
</table>

*Sticklets* as rules might suffer from similar problems as those of triggers in active databases [Paton and Díaz 1999]:

— termination (is rule processing guaranteed to terminate?). *Sticklets* always terminate since they cannot raise triggering events (i.e. matching events or lever events) that enact other *sticklets*.

— observable determinism (i.e. is the effect of rule processing as observed by a user of the system independent of the order in which triggered rules are selected for processing?). In databases, this notion seeks to extend the notion of confluence beyond the boundaries of the database itself. In Web Augmentation however, the notion of confluence and observable determinism coincides since the shared resource and the observed resource is the same: the HTML page.

— confluence (i.e. is the result of rule processing independent of the order in which simultaneously triggered rules are selected for processing?). Acting on the same DOM tree, *sticklets* could potentially suffer for confluence. We refer to this problem in Section 3.

5. WEB AUGMENTATION: CARING FOR CONSUMERS

Previous section focuses on producers. The challenge was to find a balance between expressiveness and learnability/reliability. Now, we introduce consumers. The consumer profile is that of a computer literate with e.g. basic knowledge about MS Word or installing add-ons for Firefox. Now, scripts are no longer for self-consumption but for use by a large number of users with varying needs. Being “for use by a large number of users” moves reliability to the forefront. Catering for “varying needs” boosts maintainability. This section presents how *Sticklet* tackles these concerns.

5.1. Reliability

Reliability refers to the assurance that a system will perform as expected despite environmental disruptions, hostile attacks (raising security issues), and implementation errors [Becker et al. 2006]. Mechanisms to improve reliability include fault prevention, fault tolerance and fault removal.

5.1.1. Fault Prevention. Fault prevention aims at preventing that faults are integrated into the system. Besides non-deliberate faults, we should also consider intentional threats such as creation of/redirection to phishing pages, stealing history information (or sensitive data stored on either pages or cookies), or port scanning upon the...
user’s local network (refer to [Google 2007b] for further details). The full expressiveness of JavaScript, its intricate coding and its interpreter-like nature make peering at the code of the script not an option. Sticklet abstracts away from some dangerous JS operations into more abstract and declarative description of the solution.

In addition, Sticklet also ensures confluence for simultaneous sticklet enactments, avoiding the collision problem (see Section 3). Traditional scripts are enacted sequentially based on the order they were installed. This implies changes made by the first script are visible to ulterior scripts. Two types of dependencies arise: read dependency (a script can accidentally read data written by a previous script), and write dependency (the injection point can be displaced by the writing of a node made by a previous script). As a result, the very same set of scripts can deliver different outcomes depending on the order they were installed. Sticklet addresses this issue (technical details at [Díaz et al. 2010]). Read dependencies are avoided by making sticklet changes transparent to other sticklets. Sticklets can only access the raw DOM (i.e. the DOM corresponding to the hosting page) previous to being updated by any sticklet. No way for a sticklet to see changes conducted by other companion sticklets. As for write dependencies, they are avoided by preventing sticklets from altering the basic structure of the hosting page. Augmentation can only add new anchors (i.e. levers) and notes. In this way, sticklets do not change the position of the data in the hosting page, hence, avoiding breaking companion sticklets.

Similar semantics if handwritten in JavaScript scripts would require clumsy algorithms. By abstracting into a DSL, Sticklet handles this control complexity automatically, consistently, and formally.

5.1.2. Fault Tolerance. Fault tolerance copes with the presence of faults. A system is fault tolerant if it can mask the presence of faults in the system by using redundancy. Sticklet hardwires basic handlers for a foreseeable, fixed set of cases. Table I indicates Sticklet recoverability strategies for the most common faulty scenarios. Besides HTTP errors, Sticklet also provides support for some ad-hoc defective situations, namely:

— the need for credentials when enacting LoadNote. In BookBurro, access to the library of the University of Manchester might require some credentials. The HTTP protocol already provides some codes to indicate this scenario. But, not all websites are so careful, and forget to inform the agent about this situation. That is, Sticklet can overlook this situation if only checks for the HTTP codes. This is the case for the University of Manchester. This scenario is frequent enough for Sticklet to be attentive and perform some term search in the returned page (e.g. “identification” or “account”) when the sought resource is not found in the returned page. If so happens, an error note is produced,

— the “osearch protocol” retrieves the empty result. It could happen that the current entity does not have a counterpart in the guest site (e.g. Manchester Library does not hold the book at hand). Sticklet just indicates this fact as an error note.

It could have been possible to enhance the DSL, and let users explicitly indicate a customized contingency action. Some examples follow: if the book is not available at the University Library, try the City Library; if GoogleMaps fails to retrieve a place then, query again by changing the order of parameters (might be latitude and longitude where wrongly placed in the host site). However, we did not experience a number of such scenarios large enough to ground the introduction of a new DSL construct.

5.1.3. Fault Removal. Fault removal aims at reducing the number of faults. This is related with testability which in turn, involves tracing and error reporting. Sticklet supports these two features. An important point is that tracing and error reporting should be conducted in Sticklet terms.
Fig. 17. Sticklet tracing exemplified for BookBurro: JavaScript messages are abstracted into Sticklet terms (i.e. WhenOnWall, SelectBrick, etc.). Trace format: "Sticklet [<projectName> - <stickletName>] [PHASE #][<phaseName>]".

**Tracing.** When developing scripts for others, debugging is a must. Tracing is a basic mechanism to follow and understand the flow of the scripts process. Since scripts are described in Sticklet terms, this flow is more abstract, and hence, easier to follow than its JS counterpart. Sticklet provides a trace message for each clause. Traces are displayed on the log console of the browser (for Firefox, press CTRL+Shift+J). Figure 17 provides an example while tracing the BookBurro script. Worth noticing: phase 2 reports the number of nodes fulfilling the match condition (i.e. this indicates how many times the sticklets are to be fired); phase 3 displays the content of bricks (hence the user is aware of which data is being extracted); phase 5 announces the URL to be

ACM Transactions on the Web, Vol. V, No. N, Article A, Publication date: January YYYY.
called. In this way, consumers are aware of the data/services being used to achieve the augmentation, hence improving the trustworthiness on sticklets.

Error Reporting. JavaScript engines tend to be poor on error reporting. Basically, developers are responsible for the handling of error states using “try/catch” blocks. This is so because the faulty cause could greatly differ among JS scripts. By contrast, Sticklet restricts the expressiveness of JS. This permits to limit faults to four scenarios: guest variable not bounded, call-response error, server not found, and content-type not supported (see Figure 18). In addition, Sticklet accounts for syntax checking and type checking\(^\text{11}\).

5.2. Maintainability

We distinguish two rationales for sticklet maintenance. First, the evolution of the augmentation functionality. The augmentation functionality might need to be extended/reduced in either its scope or its content. This functionality is realized as a stickletBox, and its evolution will more likely imply addition/deletion of sticklets. Since sticklets are decoupled, additions/removals will have no impact on the remaining sticklets of the stickletBox.

The second rationale for maintenance is changes on the underlying website. This is more cumbersome since it might impact the layout, content or even URL structure that grounds the sticklet. Unfortunately, websites are reckoned to evolve frequently, with evidences in the figure of twice a year [Dontcheva et al. 2007]. This may represent that the associated sticklet stops working twice a year. For only 5 or 6 sticklets, this rate might account for maintenance becoming a monthly burden. This is hardly bearable if directly conducted in JavaScript. This scenario calls for approaches where development is so straightforward that makes easier to develop from scratch than maintain.

\(^{11}\)Type checking is provided for the following types: XPath, Regular Expression, URL Regular Expression, URL Call and String.

---

ACM Transactions on the Web, Vol. V, No. N, Article A, Publication date: January YYYY.
Besides declarativeness, Sticklet incorporates assistance mechanisms (see Subsection 6.2 “No time to code”) that greatly facilitate and speed development. According to first evidences (see Section 7), a sticklet takes around thirty minutes to develop. This limits to thirty minutes the maintenance penalty.

6. WEB AUGMENTATION: WHEN ATTENTION IS SCARCE

So far, we considered the bulk part of script prosumption: handling the code. However, this is not enough. We should also turn to other ancillary aspects of prosumption that could be negligible in an organizational setting, but become crucial in a Web2.0 scenario, namely, provisionability (i.e. facilitating prototyping), tailorability (i.e. ease to customize the code), shareability (i.e. the facility to share an augmentation script), familiarity (i.e. to what extent the tool resembles what the users might be acquainted to), or operability (i.e. ability of the tool to be easily operated by a given user in a given environment).

Being full-fledged Greasemonkey (GM) scripts, sticklets can tap into the tools available for GM script operation: editors, script repositories or installation utilities are also available to sticklets. However GM and Sticklet target different audiences: dedicated programmers versus occasional programmers. This difference impacts the operability requirements. We then reconsider GM operation at the light of three main tasks: coding (producer perspective) and, installation and sharing (consumer perspective).

6.1. Greasemonkey Operation

Traditionally, Web Augmentation is conducted by JavaScript (JS) programmers. Although programmers are not our target audience, they know what augmentation is, and they can act as heralds of Sticklet. After all, Sticklet can serve for quick prototyping before moving to JS for more sophisticated outcomes. Therefore, we want Sticklet to sound familiar to GM users. This leads to a main decision: supporting Sticklet as an internal DSL of JavaScript\textsuperscript{12} [McCrea 2007]. Indeed,

sticklet are installed and operated in the very same way as JS scripts in GM, i.e.:

— edited as any other text file with extension “scriptName.user.js”,
— installed by just dropping the sticklet file into the browser,
— enabled/disabled (i.e. temporarily stopping the augmentation) using GM facilities,
— consulted through the GM library that keeps the list of scripts currently installed,
— shared through the userscripts.org repository.

Our hope is to facilitate the transition of the 38 million GM users, the most active community of Web Augmentation to date. That said, GM targets programmers who can spend several hours developing their scripts, and consumers who do not hesitate peering at script repositories. However, this is not our scenario.

6.2. No Time to Code

New Scenario. Provisionability refers to the difference in production between professional coding and end-user coding [Green et al. 2006]. When attention is scarce, users tend to be less systematic, commonly resorting to evolutionary and exploratory prototyping. We consider two likely coding scenarios:

— impulsive prototyping. We address short term and situational needs where programming is not a planned activity but a more circumstantial activity. For instance, a

\textsuperscript{12}This syntactic dependency from JS is surfaced in the use of the dot notation to concatenate the distinct Sticklet constructs.
user can first augment Amazon with BookByte prices, get immediate feedback to see if it works, and differ to the next spare slot the completion of the script. This can be hours, days or, might be, the next time a purchase at Amazon reminds him about this sticklet.

—opportunistic reuse. The complexity and difficulty of programming can be reduced by giving end users a head-start with existing code, which they can adapt to their own purposes. Opportunistic reuse is regarded as a main enabler for end-user programming: “reuse is often what makes a project possible, since it may be easier for an end user to perform a task manually or not at all than to have to write it from scratch without other code to reuse” [Blackwell 2002]. This facilitates tailorability, i.e. consumers can customize third-party sticklets to their own preferences. In addition, users might be impelled to code when they try to mimic the augmentation achieved by someone else.
Tackling the new scenario. This way of working requires of agile means for enhancing and editing existing sticklets. Our aim is to make sticklet edition a more “impulsive” action so that editing can occur at the time and at the place where the augmentation takes place. Back to our first sample, the user reminds his one-sticklet BookBurro script at the time he is purchasing at Amazon. He wonders which could be the price at the Powell’s bookshop, and, at this very moment, he is impelled to enhance the script. The aim is to drive this impulse at the time it rises.

So far, the edition of GM scripts requires two clicks: one click on the GM icon at the status bar which opens a menu with the list of installed scripts; next, another click to select the script which makes the code to pop up in the default text editor. It is not a big burden (just two clicks) but you have to move around the browser window, and hence, be determined to edit the script.

Intended for general-purpose scripts, GM uses a general-purpose editing process where enactment and edition are two separated processes. By contrast, domain specificity might be go beyond specification to impact also editing. The editing process can be contextualized within the most likely scenarios of use. Sticklet looks for impulsive prototyping and opportunistic reuse. This calls for edition to be integrated as part of the augmentation itself, i.e. you see the augmentation, you are impelled to edit the code. To this end, Sticklet introduces an inline editor as part of the sticklet decorator (see Figure 19). Click on this icon and you can update, clone or delete the sticklets.

Back to our example, the user can readily (1) edit BookBurro when at Amazon; (2) clone the sticklet “Price At BookByte”; (3) substitute the LoadNote clause with Powell’s URL, (4) assign “assisted” to the XPath-valued clauses, and (5), click the “Save” button. Sticklet will assist in setting the values for the unbounded clauses, and will regenerate the script. If you want to create a bright new script, just provide a different “name”. Otherwise, the BookBurro script is updated. The inline editor not only makes the code one-click away, but frames the edition within the most likely context of use: at the time the sticklet is enacted.

The bottom line is that

... domain specificity not only impacts the constructs used to describe the solution but also the way to reach this solution. JavaScript and Sticklet not only differ in their primitives but also in their development processes.

6.3. No Time to Install

New Scenario. GM scripts are very easy to install: drop the script file into the browser and you are done. Since sticklets are functional GM scripts, so can be done for sticklets. However, this scenario assumes (1) GM is already installed, and (2), the script file is already in either the desktop or a remote repository. Sticklet departs from this setting by considering (1) consumers might not have GM installed, and (2), they may never hear about Sticklet, not even about Firefox add-ons.

Tackling the new scenario. This new scenario advises the installation to be somehow linked to the sharing process, better said, the sharing artefact. Fortunately, and unlike “desktop artefacts”, “browser artefacts” (i.e. those enacted within the browser’s boundaries) can download and deploy their interpreters at runtime as Web Services. The Sticklet engine is also available as a Web service at webaugmentation.org. In this way, we introduce “the sticklet URL”, a URL that not only identifies univocally the sticklet but also denotes an implicit petition to the site “webaugmentation.org” to install this sticklet. This request is codified as a tinyURL. Note that the feasibility of this solution also rests on the compactness of the code: DSLs account for lean code which can be packed as a URL.

ACM Transactions on the Web, Vol. V, No. N, Article A, Publication date: January YYYY.
These tinyURLs can be obtained from the inline editor when clicking the Twitter/Facebook icon (see later). Next, you can share it through Twitter or just send it through email. No matter the means, when clicking a sticklet URL, the Sticklet service cares for the burdens of your installation. The Sticklet service first verifies the receiver's browser configuration (i.e. Sticklet validator plug-in available, Greasemonkey plug-in available, Sticklet engine on), and next, installs the sticklet (provided user consent is granted). For instance, the BookBurro tinyURL is http://tinyurl.com/bewql5e. Just copy this URL in the browser bar to see this service at work. The rendered page should look as the one in Figure 20. This page contains a progress tracker that guides the user throughout the installation process. Depending on the current browser configuration, the starting point changes. Though this verification is conducted every time a sticklet is installed, the overhead is negligible while saving users from downloading themselves the Sticklet add-on.

In brief, 'browser artefacts' allow for installation to be a side-effect of sharing. When in a trusty setting, URLs can be used not only to univocally identify the artefact but transparently request the installation of this artefact.
6.4. No Time to Share

**New Scenario.** This paper starts with the vision of end users *prosuming* scripts as easily as they do for pictures or blog posts. One distinctive feature of this scenario, and key ingredient of the Web2.0, is sharing. Traditionally, script sharing is achieved through repositories (e.g. Greasemonkey and Chickenfoot follow this approach). Being valid JavaScript code, *sticklets* can be uploaded and managed through Greasemonkey repositories. Indeed, you can find several *sticklets* at http://userscripts.org/users/sticklet. On the upside, repositories provide public access to not only the code but also the reviews or discussions about the script. On the downside, repositories force producers to disclose their scripts to the wider public which could be intimidating for end users. Indeed, studies from social networks indicate an increase in sharing if conducted within smaller groups [Bogart et al. 2008].

**Tackling the new scenario.** *Sticklet* explores a new scenario where consumption is reactive (i.e. someone tells you about the *sticklet*) rather than proactive (i.e. you looking into a script repository). The aim is to let users share *sticklets* without the burden of uploading them in a repository. Like pictures. After all, sharing pictures...
does not demand uploading them first at Flickr. We explore "inline sharing" whereby sticklets themselves offer the means to be shared through the social networks. Specifically, Sticklet syntax includes two annotations: @sticklet:facebook and @sticklet:twitter (see Figure 21). These boolean annotations permit the producer to indicate whether the script can be shared through Facebook or Twitter, respectively. These annotations make the sticklet decorator exhibit the icons of these popular sites. By clicking the Twitter icon, users create a tweet that includes the sticklet's tinyURL.

The bottom line is that

sharing "should be made as simple as possible but not simpler" as a way to foster production.

7. EVALUATION

The main goal of Sticklet is to make Web Augmentation accessible to consumers and producers alike. In this sense, the matter is mostly about affordance, i.e., making augmentation accessible to a wider audience. In this setting, the quality of use becomes paramount, i.e. "the user's view of the quality of a system containing software, and is measured in terms of the result of using the software, rather than properties of the software itself" [ISO/IEC 2001]. ISO-9126 provides a framework to evaluate quality in use. This section provides a preliminary evaluation of Sticklet along the ISO-9126's quality-in-use dimensions: (i) effectiveness (i.e., the capability of the software product to enable users to achieve specified goals with accuracy and completeness), (ii) productivity (i.e., the capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness), (iii) safety (i.e., the capability of the software product to achieve acceptable levels of risk), and (iv) satisfaction (i.e., the capability of the software product to satisfy users). Effectiveness and productivity can be measured objectively: number of completed tasks and minutes to complete them, respectively. On the other hand, while safety is an objective measure, our target population (i.e., people with no previous knowledge of JavaScript) may not have the means to perform such evaluation. Hence, we opted for evaluating trustworthiness, i.e. how trustworthy participants perceived Sticklet to be. Both trustworthiness and satisfaction were assessed through specifically designed questionnaires.

At the time of this writing, Sticklet has been available as a Mozilla add-on for over a year. According to the Mozilla figures\footnote{https://addons.mozilla.org/addon/13/Sticklet/statistics/?last=90}, Sticklet has 37 average daily users: 20% of their browsers are in Spanish (which can be anticipated by the nationality of the authors); 30%, 8% and 7% are in English, German and Italian, respectively (which can be explained by the presentation of Sticklet at a demo session of an European conference); finally, 35% come from other languages (Chinese, Russian, etc.) which can be attributable to serendipity searches. These numbers are still very low to permit a proper evaluation of Sticklet "in the wild". Hence, this evaluation is based on subjects mainly taken from academia.

Sticklet design is driven by the aim to satisfy producers and consumers alike. Nevertheless, these two types of users have different requirements (e.g., expressiveness vs. reliability/trustworthiness) and backgrounds (hobby programmers vs. computer literates). Thus, each evaluation targets different audiences.

7.1. Sticklet Consumption for Computer Literates

A comparative design was adopted to examine the influence of distinct background variables in the perception and use of Sticklet. The sample was composed of a group of users with similar characteristics of the potential consumers of Sticklet.
7.1.1. Research Method.

**Setting.** In order to eliminate differences in the perception of Sticklet due to hardware or bandwidth differences, the study was conducted in a laboratory of the Computer Science Faculty of San Sebastian. All participants used computers with the same features (i.e., Intel Core 2 1.86 GHz, 3 GB RAM and Windows XP Professional SP3) and a clean installation of Firefox 12.0.

**Procedure.** The study intended to mimic as closely as possible the circumstances potential consumers might encounter when deciding to try a sticklet script. Hence, explanations and instructions were reduced to the minimum. Before the participants started, they were informed about the purpose of the study and were given a brief description of it (5 minutes). The sample sticklet augments the movie listings page at a popular Spanish portal, Terra, with scores and comments from the website www.imdb.com\(^{14}\). Next, participants were handed out a sheet that instructed them to access the Onekin Group twitter page\(^{15}\), where a tweet, supposedly sent by a mate, commented about the wonders of a sticklet he had made and provided this sticklet’s tinyURL. From then on, Sticklet assisted participants through a progress tracker (see Figure 20). In order to measure productivity, participants were then asked to note down the time when they clicked the link in the tweet, and again after installation, when they were able to see the augmented Terra page. Last, participants were directed to a GoogleForms online questionnaire to gather their opinion about Sticklet.

**Subjects.** Thirty three first year undergraduate computer science students participated in the study. The majority of participants were male (78.8%). Regarding age, 81.8% were in the 18-20 age range and all participants were below thirty years old. 63.6% check movie listings before watching a film but only 6.1% were already familiar with the Terra website. 90% access social networks on a daily basis and 45.5% tweet with a weekly frequency. Last, concerning the participants’ browsing behaviour, in the last year participants had installed between 0 and 20 applications/plugins/add-ons, with a mean of 5.4.

**Instrument.** An online questionnaire served to gather users’ experience using Sticklet. It consisted of five parts, one to gather the participants’ background and one for each of the quality-in-use dimensions. In order to evaluate effectiveness, the questionnaire contained the proposed tasks so that participants could indicate if they had performed them, while productivity was measured using the minutes taken in such tasks. Trustworthiness and satisfaction were measured using 7 and 10 questions, respectively, with a 5-point Likert scale (1=completely disagree, 5=completely agree).

**Data Analysis.** Descriptive statistics were used to characterize the sample and to evaluate the participants’ experience using Sticklet. Moreover, t-test analyses were performed to assess differences among groups of users (e.g., familiarized vs. non-familiarized with film listing websites). PASW Statistics 18 for Windows [SPSS ] was employed to perform the different analyses.

7.1.2. Results. We begin by describing the qualitative evaluation. Two observers were present during the evaluation, who were instructed not to interfere unless strictly necessary. Only 2 out of 33 participants had doubts during installation and only in one case was the intervention of one of the observers necessary. This was due to the user leaving the page before accepting a pop-up window that prompts the user to accept the installation of the sticklet\(^{16}\). Five of the participants failed the exercise since they forgot to reload the page to see the augmentation at work. That is, since the exercise

---

\(^{14}\)This sticklet can be downloaded from http://tinyurl.com/ad2uld2.

\(^{15}\)http://twitter.com/onekin

\(^{16}\)To prevent this from happening in the future, the installation process was later modified to wait for the pop-up (modal window) before continuing the process.
began in the *Terra* page, and next, they were instructed to download the *sticklet*, these participants expected the *Terra* page to change all of a sudden as part of the script installation. Despite being so instructed, these participants missed to reload the page, and hence, failed to see the augmentation.

Besides the questionnaires for the quantitative evaluation (see below), an optional open question was left to gather the participant's comments. Eighteen participants chose to give their impression of the tool. One participant found *Sticklet* too convoluted for her needs. She argued that the browser's tabs are sufficient to keep different sites open simultaneously, despite the cognitive fragmentation that goes in moving between tabs. Otherwise, the perception was positive. Only one participant found the installation process cumbersome. Three participants suggested the *Sticklet* decorator to resize automatically to fit the *sticklet* output. One participant had the impression that the sharing of a *sticklet*, even from an acquaintance, does not guarantee that the *sticklet* is virus free since “tweeting is not a serious way to share code”. The point to note here is that the trustworthiness of the object (that being some data or code) is that of the informer. All in all, this observation introduces the conduit as a source of trust. Another noteworthy happening is that two subjects were playing around with the *Sticklet* inline editor, though no hint about *Sticklet* editing facilities was given to the participants. This suggests that incorporating the inline editor as part of the *Sticklet* decorator might encourage consumers to play around, and eventually, become producers. By permitting editing from the very same place where the output is rendered (no need to deploy additional browser menus), the *sticklet* code is just one-click away from its visible effects. Next, the quantitative evaluation of the dimensions described above is detailed.

**Effectiveness.** Effectiveness is the capability of the software product to enable users to achieve specified goals with accuracy and completeness [ISO/IEC 2001]. For consumers, effectiveness has to do with installation and enactment of *sticklets*. The questionnaire asked for these two tasks (see Table II). All participants were able to complete the installation, i.e., install *Greasemonkey*, *Sticklet* and the *sample sticklet*. Interestingly enough, five participants did not catch the need to reload the original page for the augmentation to show up, but expected this process to take place automatically.

**Productivity.** Productivity is the capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness [ISO/IEC 2001]. We asked participants to note down the time twice, first when they clicked on the tweet's *tinyURL* and again when they accessed the augmented page, after installation. Productivity was collected by requesting the number of minutes between the two moments. Participants reported between 2 and 10 minutes, with a mean of 3.22 minutes, from the moment they clicked on the tweet to the point where they accessed the augmented page.

**Trustworthiness.** This section intends to measure trustworthiness perception of participants (i.e. whether *sticklets*’ behaviour is perceived to be consistent, predictable and trustworthy). Table III summarizes the results. Items 2 and 3 reveal how trust on both *Firefox* (as the container of plugs-ins) and friends (as the authors of *sticklets*) are perceived as the strongest source of confidence. It is interesting to observe that item 4, i.e. the fact that friends publicize their *sticklets*...
Table III. Trustworthiness Perception Results from 1 (completely disagree) to 5 (completely agree).

<table>
<thead>
<tr>
<th>Item</th>
<th>Computer Literates</th>
<th>Hobby Programmers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. In general the demo has inspired me confidence</td>
<td>3.73</td>
<td>3.89</td>
</tr>
<tr>
<td>2. The fact that Sticklet is downloaded from the official Firefox plugin page inspires me confidence</td>
<td>4.24</td>
<td>4.34</td>
</tr>
<tr>
<td>3. To know that a friend has developed the sticklet would inspire me confidence</td>
<td>4.12</td>
<td>4.00</td>
</tr>
<tr>
<td>4. To know that a friend has developed the sticklet and has shared it on Facebook would inspire me confidence</td>
<td>4.09</td>
<td>3.78</td>
</tr>
<tr>
<td>5. The fact that I understood the code inspires me confidence</td>
<td>3.79</td>
<td>4.28</td>
</tr>
<tr>
<td>6. To know that Sticklet does not allow malicious JavaScript code would inspire me confidence</td>
<td>4.27</td>
<td>4.27</td>
</tr>
<tr>
<td>7. To know that Sticklet was recommended by a friend via Twitter, even if she/he was not the developer, would inspire me confidence</td>
<td>3.03</td>
<td>2.73</td>
</tr>
</tbody>
</table>

via Facebook (as a sign of pride in the sticklet), seems to add no extra confidence to consumers when compared with being developed by a friend (i.e., item 3). Both the understanding of the Sticklet code (item 5) and its built-in security rules (item 6) are also highly regarded. Notice how code understanding (item 5) is the one with the lowest standard deviation (i.e. 0.857) which might suggest the highest unanimity as a confidence builder. Finally, sharing via Twitter (item 7) is ranked lowest as a confidence builder. This might suggest that the notion of “friend” in Twitter is diluted to merely mean list of contacts rather than true friends you can rely upon. This seems to indicate that Sticklet online tweeting did not scale to our expectations as a trust-builder.

Relations among variable differences among groups, according to background variables, were analysed using t-test. We were specially concerned about the impact that being familiarized with film listing websites could have in the perception of the sticklet, so that regular users of those sites could have measured up Sticklet. However, we found no statistically significant differences in either trustworthiness related to being an official Firefox plugin, trustworthiness related to being produced by a friend, or trustworthiness related to understanding the code and its built-in security rules between participants familiarized with film listing websites and those who were not. Where we did find statistically significant differences was among the participants that had installed few (two or less) and multiple (three or more) applications/add-ons/plugins in the last year. Specifically, those who installed multiple apps showed higher confidence related to being an official Firefox plugin (Mean=4.63) than those who installed less (Mean=3.92, p=0.033). The same happened when considering trustworthiness related to being produced by a friend (Mean=4.58 vs. 3.69, p=0.009) and trustworthiness related to the built-in security rules (i.e., not allowing malicious JavaScript code) (Mean=4.63 vs. 4.00, p=0.024).

Though consumption was mainly targeted to computer literates, we were also interested in analysing the impact that a more technical background has on trustworthiness perception. To this end, we handled the same questionnaire to the “hobby programmers” group (see next subsection). Results can be seen in Table III. It is interesting to observe how confidence by code understanding (item 5) raised to a mean of 4.28. No statistically significant differences were found in trustworthiness perception between computer literates and hobby programmers.
Table IV. Satisfaction Results from 1 (completely disagree) to 5 (completely agree).

<table>
<thead>
<tr>
<th>Item</th>
<th>Mean</th>
<th>St. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. It is comfortable to be able to click on a tweet to install the tool</td>
<td>4.21</td>
<td>0.857</td>
</tr>
<tr>
<td>2. During the installation process, I always knew what I was meant to do</td>
<td>4.48</td>
<td>0.795</td>
</tr>
<tr>
<td>3. The companion <em>Twitter</em> message helped me understand what the <em>sticklet</em> did</td>
<td>3.52</td>
<td>1.064</td>
</tr>
<tr>
<td>4. The <em>sticklet</em> code helped me understand what the <em>sticklet</em> did did</td>
<td>3.42</td>
<td>1.001</td>
</tr>
<tr>
<td>5. Looking at the final augmentation outcome helped me understand what the <em>sticklet</em> did</td>
<td>4.06</td>
<td>1.088</td>
</tr>
<tr>
<td>6. There were no errors during the installation and execution of the <em>sticklet</em></td>
<td>4.45</td>
<td>1.201</td>
</tr>
<tr>
<td>7. The <em>Sticklet</em> engine is fast enough</td>
<td>4.33</td>
<td>0.890</td>
</tr>
<tr>
<td>8. I think augmenting a website with data from another website is a good idea</td>
<td>4.27</td>
<td>1.008</td>
</tr>
<tr>
<td>9. <em>Sticklet</em> saves me time, since I don’t have to browse through different webs</td>
<td>4.24</td>
<td>0.902</td>
</tr>
<tr>
<td>10. I found the demo interesting enough to share it with my friends</td>
<td>3.42</td>
<td>1.001</td>
</tr>
</tbody>
</table>

Satisfaction. Satisfaction is the capability of the software product to satisfy its users [ISO/IEC 2001]. Satisfaction can be measured along three dimensions: perceived usefulness, perceived ease of use and willingness to use in the future [Davis 1989]. In this case, the product is the *sticklet* script, but our interest does not lie in measuring the usefulness of a specific *sticklet* but on what aspects of the *sticklet* script are derivable from the *Sticklet* engine. Therefore, we do not measure the “perceived usefulness” of this particular *sticklet*. Rather, we investigate which aspects of the *sticklet* script can be traced back to features of the *Sticklet* engine i.e. those that can be applied to any *sticklet* script no matter its function (e.g. efficiency, understandability, soundness, etc.).

The list of items and the results are summarized in Table IV.

As in the case of trustworthiness, being familiarized with the website did not establish statistically significant differences in any of the satisfaction items. Regarding app installation behaviour, statistically significant differences were found in items 2 to 5, suggesting that participants that install apps more often had a better understanding of both the installation process and what the *sticklet* did (item 2, Mean=4.7 vs. 4.15, p=0.029; item 3, Mean=3.8 vs. 3.07, p=0.0220; item 4, Mean=3.8 vs. 2.85, p=0.0030; item 5, Mean=4.4 vs. 2.54, p=0.0186). As for the use of *Twitter*, a statistically significant difference was found between participants that tweet at least once a week and those who tweet monthly or never in item 1 (i.e., comfort of installation clicking on a tweet, Mean=4.53 vs. 3.54, p=0.0303).

7.2. Sticklet Production for Hobby Programmers

A comparative design was adopted again to examine the influence of distinct background variables in the perception and use of the tool. A sample of users with more technical qualification was sought to evaluate the perception of potential *Sticklet* producers. In this case, a call to participate in the study was issued among the faculty and Ph.D. students of the Computer Science Faculty of San Sebastian. The evaluation also included questions about background in related technologies (e.g. *JavaScript*, *XPath*, etc.).

7.2.1. Research Method.

Setting. The study was conducted in the same laboratory the consumer study had taken place in.

Procedure. All participants were handed out a sheet with instructions for each task (e.g., what web to access, when to take note of the time, etc.). The study was divided in three tasks. Before they started, a general description of Web Augmentation was given (15 minutes).

First task: consuming a *sticklet*. It was exactly the same the consumers had performed, i.e., installation of *Sticklet* and the execution of a particular *sticklet* starting
with a tweet. In this case, the BookByte online bookshop\textsuperscript{17} was augmented with the price for the same book in Powell's Books\textsuperscript{18}.

Second task: modifying a sticklet. A brief description about each clause of the language was provided, with an emphasis on the \textit{assisted} option (15 minutes). This option was shown at work for the Powell's Books sticklet. For the second task participants repeated what they had seen and regenerated the script using the assisted option to obtain the same functionality.

Third task: cloning a sticklet. At the beginning of the task a description of the \textit{osearch} option was provided, using the book prices at Walmart\textsuperscript{19} as example (15 minutes). Then, participants augmented the BookByte website with the reviews and ratings found in GoodReads\textsuperscript{20}. To perform this task, participants had to use both the \textit{assisted} and \textit{osearch} options, as well as searching the book by title (so introducing a new \textit{brick} since the original sticklet searched by ISBN). Finally, participants were directed to a GoogleForms online questionnaire to gather their opinion about Sticklet.

**Subjects.** Eighteen faculty and Ph.D. students participated. The majority were male (61\%) and the mean age was 33.3. Concerning browsing behaviour, in the last year participants installed between 0 and 15 applications/plugins/add-ons, with a mean of 5.5. 61\% access social networks on a daily basis while 66\% never tweet. We also gathered information about their background on related technologies (e.g., JavaScript\textsuperscript{21}) using a 5-point Likert scale (1=none, 5=expert). Participants reported a mean of 2.1 for JavaScript knowledge and 33\% had no previous knowledge. As for XPath, a mean of 2.1 was reported. Regarding the example, 50\% buy books online and 66\% check online reviews before buying a book. Last, only 11\% had accessed BookByte before the study.

**Instrument.** The questionnaire consisted of four parts: background, effectiveness, productivity and satisfaction. Production does not involve any major risk. However, as we wanted to measure the difference between both groups of participants on trustworthiness perception, the same questionnaire on this issue was also provided in this case (see above).

**Data Analysis.** Same as those for the consumers case.

7.2.2. Results. First the qualitative evaluation is provided. Following the same design we used in the previous evaluation, the same two observers were present in this case. The first task was successfully completed by all participants with the only help of the previously provided explanation (15 minutes). In the second task (i.e., modification of the provided sticklet), three participants encountered difficulties when filling the regular expression of the ExtractContent clause using the \textit{assisted} option. An observer intervened to help these participants with the regular expression, to allow them proceed with the task. Four of the participants were not able to finish the third task, failing to understand the semantics of \textit{osearch}.

Besides the questionnaires (see below), an open question was included. From the 18 participants, 13 commented. The opinions about Sticklet were positive. Some participants provided suggestions on how to improve Sticklet in the future. One participant suggested to shorten the installation process by eliminating some of the steps. Another cared about browser interoperability i.e. whether sticklets can run in browsers other than Firefox, which, so far, is not the case. Four participants commented on the \textit{assisted} option being useful when building sticklets. Two indicated that it would be nice to have the \textit{assisted} option for all Sticklet clauses (e.g., LoadNote), thus eliminating

\textsuperscript{17}http://www.bookbyte.com/
\textsuperscript{18}http://www.powells.com/
\textsuperscript{19}http://www.walmart.com/ 
\textsuperscript{20}http://www.goodreads.com/
Table V. Effectiveness Results.

<table>
<thead>
<tr>
<th>Task</th>
<th>Frequency</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1: Installation and see augmentation on the BookByte website</td>
<td>18</td>
<td>100</td>
</tr>
<tr>
<td>Task 2: Modify the sticklet using the assisted option</td>
<td>18</td>
<td>100</td>
</tr>
<tr>
<td>Task 3: Create new sticklet to see GoodReads comments</td>
<td>14</td>
<td>77.78</td>
</tr>
</tbody>
</table>

the need to write the expression manually. One more subject aligned to this thesis by suggesting that the assisted option should be the default mode of dealing with Sticklet, thus shielding the user from the syntax. This is certainly an interesting follow-on: providing a wizard that guides the user throughout the development of the whole sticklet.

Next, we look at the quantitative evaluation.

Effectiveness. Table V provides the fulfillment for the three tasks (i.e. installation, modification and cloning to create a new sticklet). Only the last presented some difficulties, where four people were not able to create the new sticklet in the allocated time (30 minutes). The elapsed time is based on a previous study were participants required 21 minutes on average to develop a new sticklet.

Productivity. Productivity is measured as the number of minutes required for each task: installation took between 2 and 7 minutes, with a mean of 3.39 minutes; modification took between 2 and 8 minutes, with a mean of 3.16 minutes, and finally, cloning (only for those that successfully completed the task) required between 8 and 24 minutes, with a mean of 14.5.

Satisfaction. Satisfaction is the capability of the software product to satisfy its users [ISO/IEC 2001]. In this case, the product is the Sticklet engine, and its ability to develop a working sticklet. While consumers focus on the usefulness of a specific sticklet, producers take a step back and evaluate whether augmentation itself pays off for the effort to learn the Sticklet language. Along the lines of the approach proposed in [Davis 1989], we first evaluated the perceived usefulness of augmentation itself (items 1-4). Next, we focused on Sticklet as a mean to obtain the augmentation end. Specifically, the usefulness of Sticklet’s inline editor, the assisted option, and the inline sharing facilities (items 5-10) were measured. Finally, we wanted also to measure the willingness to use Sticklet in the future (items 11-13). Results are summarized in Table VI.

We found no statistically significant differences in any of the items between participants that reported previous expertise (3 or higher) or no expertise (2 or lower) in JavaScript. In the case of XPath, item 10 (i.e., ease of developing using the assisted option, Mean=4.87 vs. 4.2, p=0.0154) was higher valued by XPath knowledgeable users. This seems to suggest that participants with previous knowledge of XPath were able to appreciate the advantages of the assisted option as opposed to having to write the XPath expression by hand. Statistically significant differences were also found among the participants that had installed few and multiple applications/add-ons/plugins in the last year. Participants that install more believed that Sticklet is more useful (item 2, Mean=4.42 vs. 3.5, p=0.062). Moreover, participants that install more ranked higher the questions related to the Sticklet inline editor (item 6, Mean=4.08 vs. 3.17, p=0.0026; item 7, Mean=4.58 vs. 3.83, p=0.0309), which may be due to being more accustomed to trying new things on the Web.

8. RELATED WORK

This work takes inspiration from two main areas: End-User Development and Web Mashups. Like mashups, Web augmentation also reuses existing Web resources. Unlike mashups, augmentation does not aim at creating a new application but complementing an existing one. This implies that common mashup techniques for the dis-
Table VI. Satisfaction Results from 1 (completely disagree) to 5 (completely agree).

<table>
<thead>
<tr>
<th>Items</th>
<th>Mean</th>
<th>St. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. I think augmenting a website with data from another website is a good idea</td>
<td>4.62</td>
<td>0.6781</td>
</tr>
<tr>
<td>2. I think <em>Sticklet</em> is useful for avoiding going back and forth between websites</td>
<td>4.11</td>
<td>0.8089</td>
</tr>
<tr>
<td>3. I think <em>Sticklet</em> helps to keep focus without being distracted by browsing</td>
<td>3.95</td>
<td>1.1772</td>
</tr>
<tr>
<td>4. I think <em>Sticklet</em> is useful for decreasing the comfort threshold to recover data for decision taking (e.g., which book to buy)</td>
<td>4.00</td>
<td>0.8164</td>
</tr>
<tr>
<td>5. There were no errors during the installation and execution of the <em>sticklet</em></td>
<td>3.72</td>
<td>1.2355</td>
</tr>
<tr>
<td>6. I think <em>Sticklet</em> in-line editor is easy to use</td>
<td>3.75</td>
<td>0.7857</td>
</tr>
<tr>
<td>7. I think <em>Sticklet</em> in-line editor eases clone &amp; own sharing</td>
<td>4.33</td>
<td>0.6667</td>
</tr>
<tr>
<td>8. I think <em>Sticklet</em> in-line sharing facilities (e.g., <em>Twitter</em> button) impulse sharing</td>
<td>3.33</td>
<td>1.0055</td>
</tr>
<tr>
<td>9. I think the <em>Sticklet</em> assisted option and its intersperse grid are easy to use</td>
<td>4.44</td>
<td>0.8315</td>
</tr>
<tr>
<td>10. I think it is easy for me to develop <em>sticklets</em> using the assisted option</td>
<td>4.50</td>
<td>0.6872</td>
</tr>
<tr>
<td>11. I would like to install other user’s <em>sticklets</em> in the future</td>
<td>3.72</td>
<td>0.9313</td>
</tr>
<tr>
<td>12. I would like to exchange <em>sticklets</em> in the future</td>
<td>3.87</td>
<td>0.8819</td>
</tr>
<tr>
<td>13. I would like to keep developing <em>sticklets</em> in the future</td>
<td>3.94</td>
<td>1.0259</td>
</tr>
</tbody>
</table>

covery, selection or composition of Web resources [Cappiello et al. 2011; Daniel et al. 2009; Daniel et al. 2012] should now be contextualized by its relationship to the targeted website. In addition, *Sticklet* regards HTML pages as the main resource to tap into rather than RSS or API services, as it is the most common case in mashups. At this respect, an interesting work is that of [Ghiani et al. 2011] where the "mashup components" are wrappers upon websites, and the "composition model" is side-by-side integration. The resulting mashup application looks like a "quilt" of website windows. Notice, however, that the "quilt" is a standalone application, different from the "website patches". In brief, current mashup abstractions and composition paradigms depart from the mental model proposed for augmentation development. The notion of "the sticker wall" might better capture the asymmetry that exists between the website being augmented and the rest of the websphere. Consequently, existing mashup work is of interest for Web Augmentation but needs first to be tuned to these peculiarities.

As for End-User Development (see [Ko et al. 2011] for a survey), *Sticklet* introduces a lightweight development process for Web Augmentation. The aim is to enable a larger range of users to define their own augmentation scripts other than leaving this work to *JavaScript* programmers. Most works in this area take a producer perspective where the focus is on the availability of intuitive development tools and a high level of assistance [Burnett et al. 2004]. We complement this vision by moving to the foreground the consumer perspective as a main motivator for production in Web2.0 scenarios. According to this dual perspective, we introduce in Section 2 a set of quality attributes to guide the development of *Sticklet*, namely, familiarity, operability, provisibility, expressiveness, maintainability, learnability as quality criteria for production whereas shareability, installability, security, understandability and tailorable guide consumption strategies. This section discusses related work along these criteria.

The aim of simplifying Web Augmentation has also been addressed using a range of techniques (e.g. Visual Programming Tools, APIs, DSLs or hybrid architectures) and tackling different augmentation scenarios: generic augmentation (i.e. augmentation that can be conducted for any website), niche augmentation (i.e. augmentation that focuses on a specific kind of sites), and opportunistic augmentation (i.e. unplanned augmentation). The rest of this section uses a representative tool to ground each approach: *Sticklet* (as the DSL representative), *Platypus* [Turner 2005] (Visual Programming approach), *Intel MashMaker* [Ennals et al. 2007] (hybrid approach), *ActiveTags* [Hagemann and Vossen 2009] (niche augmentation), *Chickenfoot* [Bolin et al. 2005]
Table VII. Web Augmentation frameworks.

<table>
<thead>
<tr>
<th></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platypus</strong></td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td><strong>Mashmaker</strong></td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td><strong>ActiveTags</strong></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Chickenfoot</strong></td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td><strong>IE Accelerator</strong></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Sticklet</strong></td>
<td>++</td>
<td>++</td>
</tr>
</tbody>
</table>

Operability (P)

<table>
<thead>
<tr>
<th><strong>Operability (P)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>(graphical editor for specification but no debug/tracing)</td>
<td>+ text/graphical with no debug/tracing</td>
<td>+ text editor with introspection</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Provisionability (P,C)

<table>
<thead>
<tr>
<th><strong>Provisionability (P,C)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>not applicable (n.a)</td>
<td>n.a.</td>
<td>++ (code increments)</td>
</tr>
</tbody>
</table>

Expressiveness (P)

<table>
<thead>
<tr>
<th><strong>Expressiveness (P)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (single-page customization)</td>
<td>+++</td>
<td>+++ (button addition)</td>
</tr>
<tr>
<td>+ (mashup customization)</td>
<td>-</td>
<td>+ (augmentation)</td>
</tr>
<tr>
<td>- (focuses on tagging sites)</td>
<td>+++ (customization &amp; automation)</td>
<td></td>
</tr>
<tr>
<td>+++ (JS + library)</td>
<td>++</td>
<td></td>
</tr>
</tbody>
</table>

Learnability (P)

<table>
<thead>
<tr>
<th><strong>Learnability (P)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+++ (graphical)</td>
<td>-</td>
<td>+ (augmentation)</td>
</tr>
<tr>
<td>(JS+XML+...)</td>
<td>+++</td>
<td></td>
</tr>
</tbody>
</table>

Maintainability (P)

<table>
<thead>
<tr>
<th><strong>Maintainability (P)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>n.a. (scripts can not be opened in the editor)</td>
<td>- (Widget /gadget code + GUI)</td>
<td>n.a.</td>
</tr>
<tr>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

Shareability (C)

<table>
<thead>
<tr>
<th><strong>Shareability (C)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+++ (JS general repository)</td>
<td>+++ (proprietary repository)</td>
<td>+++ (JS general repository)</td>
</tr>
</tbody>
</table>

Installability (C)

<table>
<thead>
<tr>
<th><strong>Installability (C)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+++ (1 click)</td>
<td>+ (1 to 6 click depending on the script)</td>
<td>+ (7 clicks)</td>
</tr>
</tbody>
</table>

Security (C)

<table>
<thead>
<tr>
<th><strong>Security (C)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- (that of JS)</td>
<td>++ (built-in)</td>
<td>- (that of JS)</td>
</tr>
<tr>
<td>(built-in)</td>
<td>++ (no security measures)</td>
<td>++ (no security measures)</td>
</tr>
</tbody>
</table>

Understandability (C)

<table>
<thead>
<tr>
<th><strong>Understandability (C)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- (that of JS)</td>
<td>- (procedural and multiple artifacts/languages)</td>
<td>++ (library provides abstractions)</td>
</tr>
<tr>
<td>++ (n.a)</td>
<td>+ (library provides abstractions)</td>
<td>++ (XML document)</td>
</tr>
</tbody>
</table>

Tailorability (P,C)

<table>
<thead>
<tr>
<th><strong>Tailorability (P,C)</strong></th>
<th><strong>Visual</strong></th>
<th><strong>TEXTUAL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- (directly on the JS raw)</td>
<td>- (configuration-based via widget properties)</td>
<td>+ (via prompts)</td>
</tr>
<tr>
<td>+ (via prompts)</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

(API approach) and IE Accelerator [Microsoft 2008] (opportunistic augmentation). Table VII summarizes the insights.

**Sticklet** uses a DSL to face generic augmentation. Broadly, production is promoted by using a declarative, domain-specific language where the notion of “wall” and “sticker” are the main metaphors. Being an abstract language, it lacks the expressiveness of general-purpose languages such as *JavaScript* but it offers in return easiness to learn and debug. User assistance is addressed not through visual means but through the “assisted” widget and the use of heuristics for generalizing *XPath* expressions, rendering of XML/JSON documents, rendering of *Sticklet notes* to *HTML*,

ACM Transactions on the Web, Vol. V, No. N, Article A, Publication date: January YYYY.
and entity linkage using OpenSearch. As for provisionability, sticklet modularity (ease to add and remove sticklets) and the in-place editor look to facilitate prototyping. Next, sticklet consumption is promoted by looking into ways to easy sharing (e.g. integrating tweeting as part of the Sticklet decorator), facilitating installation (requiring a single click), building trustworthiness through declarativeness (‘I understand, hence I trust’) and built-in security, and easing sticklet customization by adding/removing sticklet clauses.

Platypus is a Visual Programming Tool for generic augmentation. It obtains full-fledged JavaScript code for Greasemonkey using a graphical toolbar. Users directly act upon the current page through the Platypus toolbar, e.g. supressing banners, moving parts of the page to different locations, changing the style and format of page elements, or inserting their own HTML code. From a producer perspective, Platypus is a neat tool for its purpose: changing a web page based on the page itself. On the downside, visual tools might restrict the expressiveness to facilitate code generation (e.g. in Platypus, no page other than the current page can be accessed). Hence, it is not clear how a Visual approach will scale up as the augmentations become more complex. The more detailed claim that visual notations avoid the need to learn a syntax appears dubious (e.g., [Glinert 1989; Green et al. 1991]). In addition to the practical problems of real state of the screen and visual clutter, graphical programming languages suffer from being difficult to port (because of the graphics) and expensive to develop because of the high cost of building the necessary editors, compilers, and debuggers [Myers 1990]. From a consumer perspective, visual tools frequently behave as generators of JavaScript code. This hides the complexities for producers but leaves consumers with convoluted, machine-generated code, hence, obfuscating the origin of errors, or interfering with effective communication and explanation. Notice however that these tools tend to be used for self-consumption, and hence, consumption of third-party scripts is not an issue.

MashMaker illustrates a hybrid approach for generic augmentation. A distinctive aspect is that programmers and end-users asynchronously collaborate to come up with the augmentation. A MashMaker project encompasses three artefacts: the data extractor (graphically defined), the augmentation widget (which is separately coded in JavaScript), and the so-called “mashup” (graphically defined). The “mashup” links the two previous artefacts so that the widget is fed from the extractor. A library of widgets is made available by programmers to end users. This introduces two actors during augmentation: widget programmers and end-user “linkers”. By contrast, Sticklet relies on a single user.

ActiveTags is a visual tool that illustrates “niche augmentation”. Here, the scope of the augmentation is restricted so that the system can automatically infer how to extract some data, relieving the user from this burden. ActiveTags limits augmentation to tagging systems (e.g. del.icio.us, Flickr, etc.) where tags are always the data to be extracted and the augmentation levers. That is, ActiveTags permits to associate “an augmentation service” to the appearance of a tag. By clicking on a tag, the service is invoked and the returned markup is popped up. This focus permits extractors (supported in Sticklet through SelectBrick, ExtractContent, As) and augmentation levers (InlayLever, At, OnTriggeringLeverBy) to be hidden from users. This improves learnability for this niche domain. On the other side, consumer concerns are not explicitly addressed since ActiveTags expressions are thought for self-consumption.

Chickenfoot illustrates the API approach for generic augmentation. An API introduces some abstractions that shelters users from how these abstractions are implemented but without leaving the hosting language. In this way, users of API-leveraged languages can use API methods, and resort to general instructions when they require to do so (e.g. Chickenfoot methods can hide complex heuristics about how to extract...
some data based on nearby text). This results into a leaner code, easier to write and understand. Figure 22 provides the BookBurro example now as a Chickenfoot script (bold is used for the Chickenfoot API calls). Chickenfoot pioneers content extraction from HTML pages based on so-called text constraint patterns using the LAPIS library [Miller and Myers 2000]. The use of LAPIS (a Java API) in Chickenfoot (a JavaScript programming system) implies a penalty in terms of the loading of the virtual machine. This might be the reason why the latest releases of Chickenfoot resign from using LAPIS, and support an abbreviated form of content extraction which is realized as JavaScript functions. For instance, retrieving the ISBN from Amazon pages is expressed as "after(text isbn-10)" (line 8)\(^\text{21}\). In the example "text" is an HTML type whereas "ISBN-10" is a literal. Functions "after" and "before" are available to retrieve the content of the node that follows/preccedes the node identified by this expression. Therefore, Chickenfoot scripts are easier to develop and understand than using di-

\(^{21}\)Unfortunately, Chickenfoot heuristics do not work properly for the Amazon page, and the "after" call does not retrieve the ISBN. This can be settled by substituting the "after" call by "find(ISBN-10: \(\d(10)\))".

Fig. 22. BookBurro using Chickenfoot API. Bold stands for calls to the Chickenfoot library.
rectly JS. However, users still need to resort to general JS instructions (see Figure 22).

In short, Chickenfoot is more expressive than Sticklet. Sticklet just focuses on a special kind of customization: augmentation. But this limited focus permits to come up with a self-contained, domain-oriented vocabulary, and to stick to this vocabulary throughout the script lifecycle, i.e. specification, debugging, maintenance and consumption of sticklets are all conducted using the very same abstract terms. Unlike APIs, DSLs have to do without resorting to the underlying language. Sticklets can only contain Sticklet terms. No general JS sentences are permitted. This trades expressiveness for understandability and security (“malware free by construction”).

Finally, IE Accelerator illustrates “opportunistic augmentation”. This functionality of Internet Explorer (IE) permits to augment Web pages with the HTML output obtained through a service request. In the authors’ own words: “simply highlight text from any webpage, and then click on the blue Accelerator icon that appears above your selection to obtain driving directions, translate and define words, email content to others, search with ease, and more” [Microsoft 2008]. The resulting accelerator can next be shared through the IE Add-on Gallery webpage. Accelerator is an attractive tool for “opportunistic augmentation”, i.e. you are browsing, look at a city and want to see what the weather is like in this city is. You did not plan to do so. Accelerator permits to highlight the name of the city and feed this data to a weather forecast service (should this be available). Ascertainment the weather at other places would require the same process. By contrast, Sticklet supports “planned augmentation”. If you consult the weather in a routine way, developing a Sticklet script can save you from repeatedly selecting both the data and the forecast service. Expressiveness wise, Accelerator does not account for entity linkage. From a consumer perspective, accelerators are XML files.

In short, there is not a universal, one-size-fits-all Web Augmentation tool. It much depends on the task and target audience. Sticklet most distinctive feature is to move consumer requirements to the forefront. This implies to care for reliability but also shareability and installability, which are commonly overlooked in other works.

9. CONCLUSIONS

We introduced Sticklet, a textual DSL for Web Augmentation targeted to end users. Sticklet is based on JavaScript but limits JavaScript generality for the sake of learnability and reliability. Learnability wise, “end-userness” is pursued by limiting the set of constructs, and hardwiring a collection of heuristics in the engine. Based on a range of previous works, heuristics shelter users from the intricacies of (1) generalizing XPath expressions, (2) rendering of XML/JSON documents, (3) rendering of Sticklet notes to HTML, and (4) entity linkage using OpenSearch. In addition, Sticklet tackles consumer concerns in a scenario of stingy attention: provisionability, familiarity, shareability or operability have played an important role in designing Sticklet. Some of these concerns are rather new but we believe will play an important role as Web2.0 practices spread along.

Future work includes extending Web Augmentation to smartphones. Mobile users will benefit from the reduction in the number of interactions that augmentation brings. Users can augment their favourite websites to act as a hub to easily access companion sites without the need to type complex URLs or conduct lengthy searches. This would be taken care for by the Sticklet engine. Another follow-on is the introduction of sticky notes using desktop resources. Although the tendency is to move resources to the cloud, the desktop still keeps an important set of confidential material. Since augmentation is a client technology, desktop resources can safely participate in the augmentation. Finally, further evaluation is required, mainly for the consumer aspects. This requires the existence of a real community of users. We are currently conducting some experi-
ments around basketball fans to measure whether Sticklet sharing mechanisms might lead to viral sharing through Facebook or Twitter.

**APPENDIX**

```xml
<stickletBox> ::= "StickletBox" <sticklets>
<sticklets> ::= <sticklet> | <sticklet> <sticklets>
<sticklet> ::= "Sticklet" <stickletName> <scope> <requests> <rendering>
| "Sticklet" <stickletName> <scope> <lever> <requests> <rendering>
<scope> ::= "WhenOnWall" <urlPattern> <extractors>
<extractors> ::= <extractor> | <extractor> <extractors>
<extractor> ::= <selectBrick> <extractContent> <as>
<selectBrick> ::= "SelectBrick" <pointInDocument> | "selectBrick" "assisted"
<extractContent> ::= "ExtractContent" <contentPattern> | "extractContent" "assisted"
<as> ::= "As" <varName>
<lever> ::= "onlay" <at> <onTriggering>
<inlay> ::= "inlayLever" <domElement>
<at> ::= "At" <position> <varName>
<onTriggering> ::= "OnTriggeringLeverBy" <domEvent>
<requests> ::= <request> | <request> <requests>
<request> ::= "LoadNote" <urlCall> <extractors>
<rendering> ::= "StickNote" <noteContent>

<stdioName> ::= string
<stdioPattern> ::= xpath expression
<stdioPattern> ::= regular expression
<varName> ::= "$" string
<domElement> ::= "link" "button" | ...
<position> ::= "after" "upon" "before"
<domEvent> ::= "click" | "mouseover" | "mouseout" | ...;
<urlCall> ::= url
<noteContent> ::= string
```

Fig. 23. Sticklet: BNF syntax.

**ACKNOWLEDGMENTS**

This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2011-23839 (Scriptongue). Arellano has a doctoral grant from the Spanish Ministry of Science & Education. We thank Laura Vozmediano for her invaluable help and all the members of the Computer Science Faculty of San Sebastián who participated in the evaluation.

**REFERENCES**

A Language for End-user Web Augmentation: Caring for Producers and Consumers Alike


