Hindawi Publishing Corporation
Advances in Human-Computer Interaction
Volume 2012, Article ID 893575, 19 pages
doi:10.1155/2012/893575

Research Article

Constrained Wiki: The WikiWay to Validating Content

Angelo Di Iorio,[1] Francesco Draicchio,[1] Fabio Vitali,[1] and Stefano Zacchiroli[2]

[1] *Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy*
[2] *Université Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France*

Correspondence should be addressed to Angelo Di Iorio, diiorio@cs.unibo.it

Received 9 June 2011; Revised 20 December 2011; Accepted 3 January 2012

Academic Editor: Kerstin S. Eklundh

The "WikiWay" is the open editing philosophy of wikis meant to foster open collaboration and continuous improvement of their content. Just like other online communities, wikis often introduce and enforce conventions, constraints, and rules for their content, but do so in a considerably softer way, expecting authors to deliver content that satisfies the conventions and the constraints, or, failing that, having volunteers of the community, the WikiGnomes, fix others' content accordingly. Constrained wikis is our generic framework for wikis to implement validators of community-specific constraints and conventions that preserve the WikiWay and their open collaboration features. To this end, specific requirements need to be observed by validators and a specific software architecture can be used for their implementation, that is, as independent functions (implemented as internal modules or external services) used in a nonintrusive way. Two separate proof-of-concept validators have been implemented for MediaWiki and MoinMoin, respectively, providing an annotated view functions, that is, presenting content authors with violation warnings, rather than preventing them from saving a noncompliant text.

1. Introduction

Since their first apparition more than 15 years ago, wikis have gained wide acceptance and now convey a nonnegligible share of the whole content delivered via the web. Wikipedia alone suffices to showcase what we would be missing without wikis: Wikipedia is among the top 10 most visited websites, (Alexa ranking, http://www.alexa.com/topsites/, retrieved 07/02/2010) is the most popular general reference work on the web [1] and delivers about 12 millions of articles. All this would probably not have been possible with "traditional", prewiki webauthoring.

The peculiar innovation brought by wikis is what has been termed "the WikiWay": an open editing philosophy that allows users to easily collaborate on web content, imposing as few restrictions as possible on content authors [2]. That in turn fosters the creation of possibly immensely large, as in the case of Wikipedia, communities which author contents collaboratively. In a sense, a wiki can be considered a state-of-mind, an inclination shared by the users, rather than a collection of scripts and pages. Technically, the WikiWay is incarnated by careful technological choices which reduce content contribution barriers. Let us briefly review such choices.

Direct Editing within Browsers. To relax the software and infrastructure requirements which are needed to contribute content on the web, for example, owning a web server or asking for hosting.

Versioning. To favor a posteriori rollback of, intentional or not, undesired contributions over a priori access restrictions.

Plain-Text Markup, Mostly Unstructured. To rely on "learn as you go" and "learn by comparison" instead of requiring specific author training in markup languages.

Wiki authors are free to change and produce from scratch content at will, with no particular restrictions on their contributions (modulo sporadic or temporary access restrictions, mainly used to counter abuses [3]). In spite of all this freedom, it can be observed that in practice the wiki editing process is often bounded by implicit rules descending from the communities themselves, taking the form of customs or *community guidelines*: pages can be required to have no spell checking errors, to be not longer than a given character or word count, to (not) contain specific sections, to match specific page templates, and so forth. These suggested guidelines, discussed in more detail in Section 3, are just some examples of a more general trend that we observed: wiki communities tend to agree on sets of *conventions* that one or more pages should preferably adhere to. Such conventions can evolve over time and sometimes (usually for large communities) they get written down as explicit authoring guidelines, it is for instance the case of wikipedia, where page banners are used at the same time to highlight the lack of adherence to specific editorial guidelines and point to them, otherwise they stay as nonwritten conventions.

The communities hence need ways to ensure, and at the very minimum check, that those de facto constraints are respected. Looking from the outside, the freedom offered by the WikiWay seems to help in building communities which initially learn to cooperate by reaching consensus. Then consensus is often used to decide upon authoring constraints, which are self-imposed and self-enforced within communities. Those constraints preserve however the characteristic of not being "strict": thanks to the wiki technical characteristics, they *can* be violated, temporarily or not, without rendering the wiki unusable, as opposed to what would happen in more controlled environments such as Data Base Management Systems, where violations of properties such as referential integrity are not acceptable even for short-time frames. For this reason, we call them *light constraints* and we define them as follows.

Definition 1 (light constraint). A *light constraint* is a decidable constraint, that is, a boolean predicate denoting the fulfillment of some requirement, applicable to the content of a wiki, or part of it, which its community wants to be adhered to as much as possible.

The existence of the WikiGnomes [4] (users who work behind the scenes to fix minor nuisances) phenomenon confirms the community desires entailed by the definition, since much of the work of WikiGnomes is based on monitoring and converging towards community best practices. At the same time the existence of (human) WikiGnomes testifies the lack of technical support for light constraints, whose presence could help in maximizing adherence to them. Having such support, the efforts of WikiGnomes can be better directed elsewhere, for examples where nondecidable (and hence nonlight, by definition) constraints are challenged. For instance, specific decisions over the style of natural language text are mostly nonmechanizable and hence need human review.

In this work we design and implement a generic framework, called *constrained wiki*, that supports wiki communities in establishing their light constraints so that they can be mechanically checked by the wiki engine, helping in adhering to them. The framework is generic in the sense that it does not mandate a specific implementation, but rather shows how it can be easily implemented on top of existing wiki software. The challenging part of the design is to preserve compatibility with the open wiki-editing philosophy, resisting the temptation of creating an excessively constrained authoring environment that would defeat the essential wiki principles, which in turn foster community creation.

The research question within this topic is to design constraint validators that preserve compatibility with the open wiki-editing philosophy, resisting the temptation of creating an excessively constrained authoring environment that would defeat the essential wiki principles, which in turn foster community creation.

We believe in particular that this is obtained by changing as little as possible the usual wiki work-flow, for instance by providing mechanisms that

(1) enrich the plain page viewing used by all users with a validation report that fully discloses (non-)adherences to community best practices;

(2) turn page saving into a conditional, yet overridable, step controlled by an intermediate validation process that encourages the submission of "community-correct" content but does not prevent variations nor violations to best practices.

If a wiki was able to provide such features, it would implement checks while respecting the lightness of constraints, as users may still read and write pages in violation of some constraints. Validation is meant to be helpful for both readers and authors, helps in targeting community efforts in fulfilling constraints, and does not sacrifice "the WikiWay".

Document Structure. The structure of this paper is as follows. Section 3 gives the problem statement showcasing and classifying light constraint scenarios. Section 4 then investigates the requirements of a satisfactory solution. Then we present in Section 5 our own solution: the constrained wiki framework. Before presenting its implementation in Section 8, we discuss how to deploy the framework (Section 6) and how to implement validators (Section 7). Related work and conclusions are discussed in Sections 2 and 9, respectively.

2. Related Work

The study of the synergies between wikis and structured, potentially constrained, content can be framed in the more general area of Computer Supported Cooperative Work [5] which dates back to the 1980s. In this respect, the goals of constrained wikis have interesting analogies with Object Lens [6] and its predecessor system Information Lens [7].

The objective of both Information Lens and Object Lens was to provide users an environment for building their own collaborative applications. The approach proposed by the authors consists of letting users customize and combine atomic building blocks, where an atomic block could be a message, a data record, a form field, a table, and so on. A simple template-based interface supports users in creating new *customizable folders* from those atomic pieces of information.

Two aspects of Object Lens are very relevant for this research: the semistructured organization of content and the semiconstrained editing model. Object Lens was classified as a semiformal system, a system "processing some information in formally specified ways and other information in ways that are not formally specified". There is no predefined and rigorous content structure but users are free to organize their content (and applications) according to their needs and preferences. That is very similar to the goal of this work: supporting users in creating structured content without imposing them restrictions on the authoring process. In fact, Object Lens provides users template-based interfaces that (i) use forms to add and select applications components, (ii) suggest information about each object and existing relations between them and (iii) allow users to leave unset some fields and even create inconsistent information.

On the other hand, two main differences can be outlined between this work and Objects Lens: the first, as expected, is the fact that massive collaboration was not foreseeable in the 80s and Objects Lens was not designed to work in a complete open environment such as the World Wide Web; the second is the fact that Object Lens documents, although not prestructured, could not be fully modified and twisted as suggested by the WikiWay.

The idea of letting users to customize their content and applications is rooted in the early days of computer-supported collaborative research. Several researchers stressed the importance of tailorability, that is, the possibility for each user, regardless of her skills, to customize "systems" by combining general-purpose and modular assets. Some researchers were in favour of allowing customization to only few skilled authors [8], others were inclined to full customization available for all users [9]. In both cases, the semiconstrained combination of modular data was recognized as a key aspect in the content production process.

In the world of wikis a very similar approach has been adopted by the WikiTemplates system [10], authors introduced the concept of WikiTemplates, which are pairs of edit/display templates. When a page is viewed it is formatted according to its view template; when it is edited a set of editable text area will be supplied, one for each "hole" in the edit template. Users cannot modify the whole content and structure of a page, rather only the areas identified by the "holes". Editing freedom is guaranteed though the tailoring process: users can freely modify the templates, so that no limitation is imposed over the editing.

The community of wiki developers gave multiple answers to the dilemma between unstructured and structured wiki content. Traditional wikis rely on a complete open approach: users access the full source code of a page and they are entitled to change any bit of that content, through a simplified syntax. Other systems include facilities for organizing content in lists and tables without mastering directly the wiki syntax. Dynatable [11] is a MediaWiki extension that helps users in creating sophfisticated tables, treated as high-level objects that can be included in multiple pages and automatically fed by dynamic data.

The need of creating more controlled wiki content lets developers introduce the idea of structured wikis. A structured wiki is an enhanced wiki providing users traditional functionalities plus applications for managing structural content, database tuples, calendars, access permissions, and so on. TWiki [12] is the most representative platform, classified as an enterprise content management system built on the top of a core wiki engine. Similarly, TikiWiki [13] integrates groupware applications focusing on collaborative editing of content.

Structured wikis have also been studied by Trattner et al. [14]. They propose an alternative approach to create structured content without sacrificing the wiki openness: the idea is to enable wikis with a specific publishing procedure. Users are left to freely edit any page, while some of them are entitled to mark well-structured content and to make it available for safe citing. The process, in fact, includes a revision/approval workflow by a board of about 60 editors that collaborate to build a layer of annotations on (the trustworthiness of) fragments.

While this work is mainly focused on the process of authoring content, DBWiki [15] achieves similar results without adding any constraint on the editing workflow. The system, in fact, allows users to intermix unstructured content with database queries. These queries are fetched and converted into wiki fragments shown into the final pages. The editing process is still open and free, and sophisticated features are available on those inline fragments. As expected, users are required to add valid queries and are supported by ad-hoc validator tools.

The idea of exploiting wikis for the creation and management of structured information has been pushed forward by the *Semantic wikis*. A semantic wiki is a wiki that allows users to decorate plain content with semantic data, to be used for sophisticated analysis, reasoning, and searching. The creation of such a semantic knowledge base opens interesting perspectives with respect to the (more or less constrained) editing process. Semantic Media Wiki [16] allows users to add semantic properties by using an extended in-line syntax. RDF statements can be embedded within the plain source content of a page with a quite little effort. The editing area is completely unconstrained and users can also create incorrect or inconsistent statements. In order to facilitate the insertion of semantic data, Semantic Media Wiki has been extended to provide users with forms [17]. The editing space is now constrained by these forms, but users are helped in the creation of content. Semantic forms are optional and remain a relatively-intrusive interface for semantic editing. Other wikis integrates the traditional editing textarea with interfaces which help users to produce semantic data. AceWiki [18] exploits a controlled natural language to let users write unambiguous statements in English. Although users can write inconsistent statements, the system also integrates a predictive authoring tool which suggests options and values to the users.

Pushing this trend to its extreme, some wikis provide users rich interfaces for the creation of semantic data based on widgets, WYSIWYG modules and graphical environments. Sweetwiki [19] exploits Ajax technologies to provide users a very dynamic and usable editing environment. Such solutions are very far from the basic and free-editing paradigm of traditional wikis, although they might be helpful for producing structured wiki content.

An alternative and very promising approach is implemented by VisualWikiCurator [20]. The tool is a MediaWiki extension, supported by a powerful backend and a semantic engine, that helps users in creating and refining content throughout the editing process. Instead of validating content before or after editing, the system interacts with authors while they are modifying pages. Some data are automatically extracted, from other pages or external resources, and suggested to the users that can preview and correct these suggestions before committing the changes. Sophisticated interfaces allow users to reorganize structured content, to synchronize fragments with other pages, and to refine content interactively. The system will also implement machine-learning techniques to polish such automatic extraction, by exploiting intermediate users' feedback.

3. Light Constraints Scenarios

The problem attacked by the present work can be defined as follows:

> Design and implement a framework that supports wiki communities in respecting self-imposed light constraints (as per Definition 1), without sacrificing the WikiWay authoring paradigm.

To further understand the boundaries of the problem we briefly skim through scenarios that can be found in current wiki deployments on the web, which exhibit light constraint checking needs.

(1) All or specific pages of a wiki can be required to match simple quality criteria that can be mechanically checked, such as minimum or maximum length requirements on pages or sections.

Real-life instances of this need can be found in Wikipedia itself, where specific banners exist to suggest merge or split of under-/oversized articles. On the same lines, wikis used for conference organizations (a fairly common use case) can have requirements of maximum word lengths over presentation proposal abstracts or other parts of the proposals.

Some ad hoc features found in mainstream wiki engines are just instances of simple quality requirement checks. For instance, spell checking is offered by various wiki engines as a specific action that can be invoked during editing, while it fits within the above more general scenario.

A very similar approach could be used to also check if some forbidden words are used in wiki pages. Going further, we could even think of sophisticated analysis based on NLP or speech recognition techniques [21].

(2) More generally, various sorts of structural markup validation can be needed to verify that specific pages match a given "document structure", as in: *the section* "*introduction*" *must be present* or *the section* "*in popular culture*" *must not be present*. (The latter being a (simplification of a) real example coming from a recent Wikipedia trend of splitting away or removing oversized "in popular culture" sections) The presence of specific page elements can be verified against usability criteria too. For instance, it can be checked if forms, very common in wiki pages too, follow the rules discussed in [22] that suggests to use checkboxes instead of selection lists.

(3) Due to the ease of referring to pages (more than that of referring to their fragments), wikis often use sets of pages to describe different elements of the same ontological classes, of software plugins, cities, instance, musical albums, and A need faced by many wiki communities is then that of preserving the uniformity of pages describing elements of the same class. The desired structure is often captured in page templates (stored in the wiki itself) and the need of uniformity reified as a template-matching problem [23].

Surprisingly, the most widely used approach to create and manage such structures is copy & paste and iterative manual refinement and checking. Wiki content templating mechanisms [23] offer more general solutions for this and related needs. Still, they lack mechanisms to enforce template matching between instances and the origin templates.

(4) Due to typical wiki limitations, it often happens to have the same content replicas spread among different pages of a wiki. For example, ordered lists of all the elements of a class described on a wiki (e.g., the class of "European countries") can be listed on different pages according to different sorting criteria.

The fact that all those lists should contain the same elements is an obvious quality requirement, which remains implicit with current wiki technology.

From the above scenario, we can draw the potential benefits of constraint-checking support and some characteristics of light constraints. In particular, we observe that light constraints can be classified along two axes: *scope* and *expressivity*.

The *scope* indicates the cardinality of the set of pages related by a specific light constraint. As an alternative intuition, the scope can be seen as a measure of how many pages are needed as input of the boolean predicate representing the light constraint (see Definition 1) to evaluate it.

In simple scenarios, like (1) and (2) above, constraints apply to the content of a single page (i.e., the set of pages related by the light constraint is a singleton): checking the length of a section or the presence of some content are all examples of such intrapage constraints. In other scenarios constraints span multiple pages: preserving the uniformity of all pages describing objects of the same class or checking the consistency of lists in different pages are examples of such interpage constraints. The fact that a constraint is local or not obviously impacts on the way it is verified: intrapage constraints are easier to verify and can be processed by external entities that do know or very little about the origin wiki; checking an interpage constraint requires in general more knowledge of the wiki structure and extra communication (to retrieve all the involved pages).

On the other hand, the expressivity of a light constraint indicates how much of some semantic meaning of pages is needed to establish whether the constraint is respected or not. For instance, to establish the fulfillment of light constraints in scenario (1) above, only a basic knowledge of the surface syntax is needed (e.g., to spell check it is enough to distinguish words from meta commands). On the other hand, to verify if a wiki page matches a specific structural markup validation schema, as in scenario (2), more semantic knowledge is required, in particular an understanding of the structure of the page is needed to recognize headings, paragraphs, flow markup, and so forth. Once more, this distinction in expressivity impacts on how the constraint can be verified and in particular how much knowledge of the specific syntax of a wiki engine is needed.

Furthermore, semantics wikis [16, 19, 24] offer the ability to check light constraints on top of semantics facts naturally entailed by the wiki semantics. It then becomes possible to check the semantic content of a page, rather than the syntactic one. For instance, users might be interested in checking that in SemanticWikipediA [16] specific kinds of pages (e.g., those belonging to the category "countries") contain a specific statement (e.g., "... is the capital"). It also becomes possible to express constraints that involve both syntactic and semantic data: for instance, a constraint checking that a given statement exists and belongs to a given section (information about the capital of a country is always in the introduction or in a summary table) or checking that two sections contain the same semantic information (the capital of a country does not change throughout a page) and so on.

Constraint scope and expressivity are effectively orthogonal aspects. A trivial example of intrapage syntactic constraint is spell checking; one of interpage syntactic constraints is checking broken links. (Wikis may contain "broken links" that do not represent a real problem when intentionally created to add new pages, but it's still generally useful to spot them). The properties shared among pages belonging to the same group suggest a further example of interpage syntactic constraint. Many times wiki users define the structure of specific pages, the type of content, the order of the elements, and so on. These requirements are often nonwritten and manually checked or simply ignored. For instance, in [25] authors discussed the adoption of wikis within an Italian academic community, reporting examples of repeated pages, structures, and patterns developed in that context.

Even semantic constraints can be either intrapage or interpage. Checking that a triple in a wiki page expresses a given information that belongs to the first category, as well as checking that semantic data are consistent within a page. On the other hand, users might be interested in checking that the overall wiki does not contain incoherent statements or subareas of the wiki are consistent, as in scenario (4).

These scenarios taken all together emphasize that the relationship between wiki systems and light constraints is already pervasive within the wiki community, even if not explicitly. In fact, some of the most popular wikis support a limited version of light constraints as described in this work. For instance Twiki [12] presents conditional plugin, currently no longer needed because IF-THEN-ELSE construct is now part of the core engine that allows to define nonnested conditional expressions used mainly to show or hide portion of content upon the occurrence of the declared condition. For instance, it can be used also to check absence of important section block and thus print message. Similarly dokuwiki [26] provides a macros plugin that permits users to create simple conditional statements to generate text. Finally media wiki presents conditional template. This extension enables the conditional execution of a template, allowing users to define rules regarding structural composition of the page in terms of inclusion of sections if the condition occurs.

The rest of the paper will identify a possible enhancement of wikis to generlize such approaches and to helps users in establishing and verifying all kinds of light constraints.

4. Requirements

Before introducing the constrained wiki framework we need to discuss the requirements that a proper solution should strive to address. Sections 1 and 3 have shown evidence of the need of supporting light constraints in real world collaborative editing and the challenges in doing so. To our best knowledge, no wiki supports them in their full generality. For this reason, even what does "supporting them" mean is not yet clear in the literature. This work postulates that a system can be said to support light constraints when

(1) it helps the editing work of all authors giving visibility to constraint violations;

(2) it helps the work of some expert users, called "tailors", enabling the description of constraints and their association to pages. (We refer to these users as *tailors* adapting to the wiki context the definition ("users who coordinate the collaboration") of [9].)

Instantiating such a system in the wiki setting poses additional requirements on the way users should interact with it; in particular, the WikiWay should be preserved to avoid drifting from the wiki paradigm. We propose the following requirements as a merger between the above general needs and the WikiWay.

Requirement 1 (unconstrained save). "Authors should not be forced to resolve all light constraint violations in order to save a page". In apparent contrast with the purpose of light constraints support, this requirement embodies the essence of the "lightness" of constraints; it stresses the fact that constraints are meant to help in formalizing their best practices but that should not come at the cost of diminishing editing freedom.

Requirement 2 (constraint definition freedom). "Tailors should be able to work on constraints and associate them to pages using legacy wiki techniques". This includes providing simplified markup for constraint definitions and versioning of both constraints and of their association to pages.

Requirement 3 (constraints visibility). "Information on constraints should be visible to all users". This requirement is meant to provide visibility of all information relative to light constraints (which are associated to pages, which are currently being violated and which are not. Such information is required to be public beside being visible to authors during page editing. This helps diminishing the gap between page producers and page consumers. (The more is visible that something needs to be fixed, the more is likely that someone will fix it.) Ideally, the adherence to this requirement would reduce the workload (and need) of WikiGnomes.

Additionally, we recognize in light constraints a global nature that extends beyond single wiki instances. In fact their occurrence in large wiki communities is precisely what enabled the identification of light constraints in the first place. Hence, we additionally put forward the following requirement:

Requirement 4 (decoupling constraints/wikis). Light constraints are neither intimately related to pages, nor to wiki engines. Light constraints should be reusable across both pages (i.e., the same constraints associated to multiple pages of the same wiki) and wikis. The intuition behind this requirement is that constraints are conceptual entities distinguished from wiki-specific concepts such as pages or set of pages, as such it is reasonable to expect constraint reuse.

Requirement 5 (constraint generality). "All light constraints should be expressible". As wiki community needs changes and cannot be predicted, the framework should account for generality leaving open the possibility to define currently unforeseeable constraints and enforce them. Clearly, it is likely to exist a tradeoff between the achieved generality and the ease of encoding constraints by random users (see Section 7).

We claim that all the above requirements are fulfilled in the light constraint framework, whose architecture is described in the next section.

5. Constrained Wiki Framework

The definition of light constraints (Definition 1) bounds constraints only to be decidable, that is, every computable function. While apparently too liberal, that choice inherits from the observation that constraint scenarios are very general. For instance, document validation needs to be paralleled with mainstream validation languages (since they are there, experienced user are likely to want them), which are able to express complex constraints escaping the expressivity of simpler languages [27]. Similarly complex needs can be found in domain-specific wikis, for example, mathematical libraries, which exhibit constraints as complex as type checking in higher order logics. All in all, the choice of decidable constraints offers a paradigm which is fully general and on top of which more specific limits can be deployed for specific settings, without having to push them at the basis of the framework.

As a consequence, the most natural way to encode light constraints is by the means of functions, which we will call *validators*.

Definition 2 (validator). A *validator* is a function from a wiki page (the page being validated, or *subject*) and, optionally, an additional set of pages (the *validation context*) to a validation outcome.

A (validation) *outcome* is a boolean value paired with an optional explanation, meant for user consumption.

Intuitively, validators are computational entities able to decide whether part of the content of a wiki, usually a page, fulfills the light constraint embodied by the validator. Validators will be associated to pages and will produce a validation outcome. View, save, and other actions on pages will be changed to exploit that outcome. Most notably: save will become conditional on the validation outcome (or on an explicit "forced saving" required by the author) and view will notify every wiki user of the validation status of the viewed page.

In the rest of this section we will go into details of the Light constraints architecture that implements this proposal. The discussion is split in two parts: we first introduce the main concept and static entities of the framework and then we discuss how the usual wiki work flow is affected by the presence of validators.

5.1. Data Model.

One of the most important aspects of the constrained wiki framework is generality: the framework does not rely on a specific wiki platform, a specific language or a specific protocol. Figure 1 is an unified modeling language (UML) sketch of the general data model behind our proposal that can be in fact adapted to any wiki environment.

The basic entity is naturally the page, which is reported on the left. At the very minimum each page is characterized by the following properties.

Markup. a text string containing the actual wiki markup the user sees when editing a page and that is rendered on-the-fly upon page viewing. Its actual syntax is system dependent.

Name. a text string denoting univocally a page inside the system, the name should follow system-specific conventions (like CamelCase) since it is used to ease linking mechanisms.

Version. a text string denoting the version of a page; over the set of versions a total order should be defined.

Let us consider the spell checking scenario mentioned in Section 1. A sample page might be represented in the discussed minimal data model as follows, including a gratuitous spell-checking mistake. The used syntax is inspired by the object as record metaphor [28].

```
Page about = {
    markup = "Tranzactions on the Web,
    constrained wiki";
    name = "AboutThisArticle";
    version = "3.14";
}
```

The above three properties are enough, together with a distinguishable page naming convention (CamelCase in the example) to implement a basic wiki system, with versioning capabilities.

The constrained wiki framework essentially relies on two additional properties added to the page.

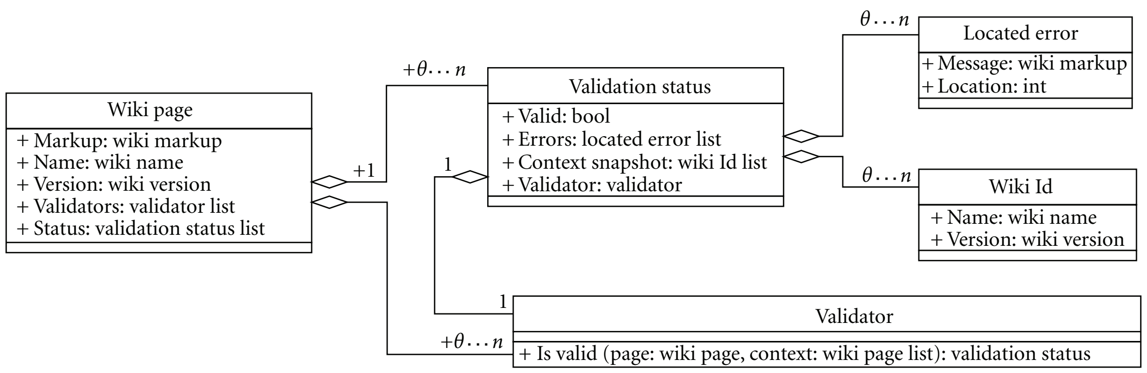


Figure 1: UML sketch of the constrained wiki data model.

Validators. A list of validators (Definition 2), one for each light constraint, which should be enforced on the page.

Status. A list of validation statuses which were associated to the owning page when the last validation attempt has been performed.

The validator and status properties are strongly interconnected. In fact, each validator exports a isValid() method producing the validation outcome and there is a 1-to-1 relation between validators and validation results. Note that validators are conceptual entities at this point of discussion. There are a lot aspects that need to be discussed about validators: the languages used to code them, their position within a wiki, the communication protocol between wikis and validators, and so forth. Sections 6 and 7 will be devoted to go into details of these aspects. What is relevant here is the logical distinction between validators and wikis and the leading role of validators in the overall LC framework.

The isValid() method abstracts the actual validation process. It takes as input both the page is being validated and a validation context. From several of the scenarios discussed in Section 3 we learned that light constraints are not always local to a single page. They often need additional information that should be found on wiki pages or even external to the wiki site. (The latter form of additional information should, however, be minimized in order to preserve the ability of users to influence validation.) In the spell checking scenario for instance, the validation can be parametric on a dictionary external to the wiki site and on extra pages containing additions to the dictionary. That page is likely to be editable by users. For this reason we extended the LC data model to also include contexts.

ValidationContext. a list of "context" pages involved in the validation. It represents the set of wiki pages (referenced by their names) on which a validator is parametric. Note that pages referenced from validation contexts are nonversioned, to better ensure liveness of wiki content validation will always be attempted using the right version of a page.

The output of the isValid() method represents the actual validation outcome (status), for each validator applied to the page. In order to meet Requirement 3 it has to provide users as much information as possible. For this reason, the ValidationStatus is not limited to be a boolean value (indicating whether or not that page is valid), but it may also contain a list of located errors, if the page turns out not to be valid.

LocatedError. Textual message which is bound to particular characters in the page markup.

This choice is motivated by the need of guiding authors toward fulfilling constraints: localized errors are easier to be spotted than global ones and hence faster to solve (at the very minimum the spotting time is reduced). The Located Error is then characterized by a location property that indicates the offset of the error in the page markup. Once again, syntactic details are not relevant at this stage, though the offset changes according to the wiki format and syntax. Note also that the textual part of message can actually be wiki markup to provide fancier (hence more expressive) messages to the user. In the example above, the ideal error message, representable in our data model, would be located at the beginning of the string peper and would contain a statement that the word does not spell check together with a link for adding the word to the current spell checking exceptions page.

Error messages are not enough to explain to users why a page is invalid. Pages which are part of the validation context of other pages may indeed change, and that can have effect on the validity of other pages. Consider once more the spell checking scenario, a user removing a word from an exceptions list may change the amount of spell checking errors in other pages. The information on why this page is no longer valid needs to be available to users. That is why we also included contextSnapshot as a property of validation status.

ContextSnapshot. A list of page references corresponding to the validation context, together with their version.

The ContextSnapshot is slightly different from the input validation context as it also indicates the actual version number of each page involved in the validation. This way it will always be possible to retrieve the exact set of pages which leaded to a particular validation outcome and to better explain that outcome to the final user. (Note that changes to external information used by validators cannot, in general, be captured in the same way: yet another good reason to keep as much validation information as possible represented as wiki pages.)

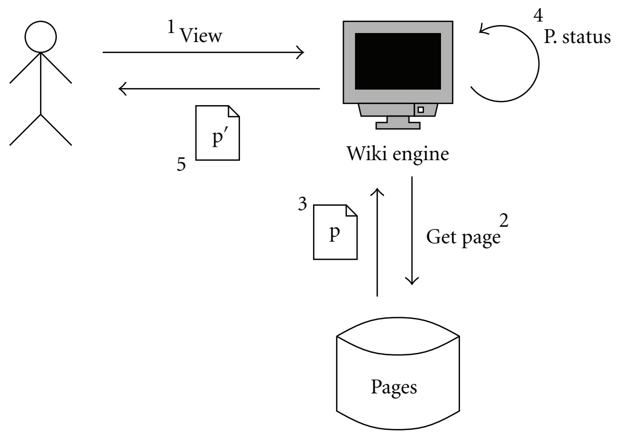


Figure 2: Runtime behaviour of the View action.

5.2. Changes on the Wiki Work Flow. The presence of validators changes substantially the work flow of wikis based on our model. Each operation on a page becomes parametric in the set of validators associated to that page. In particular, viewing a page becomes viewing both its actual content and its validation status (supplied with all the relevant information to spot and fix validation errors), while saving a page becomes invoking validators and, if not valid, deciding whether saving it or not.

Two main classes of users are involved in this scenario (as often happens, the same user can play different roles at different times).

Visitors and Authors. Users who view or edit the actual content of wiki pages. No particular skills are required nor more expertise than that required by common wiki sites.

Tailors. Users in charge of configuring and selecting validators associated to a given page. Usually, but not necessarily, users playing this role are more experienced than others [9].

In order to explain the role of each user and the revised LC wiki work flow, we discuss individually the two main operations of a light-constrained wiki, View and Save focusing on their differences with the corresponding wiki operations.

View. Our architecture transforms the view operation in *annotated viewing* whenever a user accesses a page. The page content is rendered as usual, but is enriched with a detailed report of the validation process. Figure 2 summarizes the runtime behaviour of the View action.

The user involved in such scenario is a common user who simply requires a page (step 1 in Figure 2); the wiki engine retrieves it (steps 2-3) and its associated validation status (step 4) before returning it to the user (step 5).

Some points are worth being remarked about the generalization of our schema. First of all, we have depicted a content repository without dealing with its actual implementation: wiki systems use different techniques to store pages, from MySQL (http://www.mysql.com/) databases (as WikipediA) to plain text file (as most wikis), from RDF tuples [29] to version control system repositories (e.g., [30]) Moreover, they implement specific solutions to associate metadata to pages (fields in databases, external log file, specific lines in text files, and so on) and these metadata can be usually customized or extended. We propose to introduce a new class of metadata about the validation status of a page. The key point is that the wiki engine gets pages and retrieves such status, previously set by validators: no matter how these actions are actually performed. Note also that the position of validators, as well as their internal behavior, is not relevant at this stage.

The analysis of the View action from the user perspective is interesting as well. Few changes are introduced on the behaviour of readers, who access wiki content as they always do, but can even read suggestions from validators or simply ignoring them, if not interested. The wiki engine produces a compound page where content and validation outcomes are displayed together (see Figure 5 for a sample screenshot). It is worth spending some words about the format and detail of such outcome: different pages can be involved in the validation process so that several information could be displayed, often not stored in the page being validated. Consider for instance, the spell-checking scenario, where pages are validated against another page listing exceptions, or the template matching, where the same template can be referred from many pages it applies to. It is very useful to show users such a chain of dependencies and relationships. Moreover, errors should be localized, as discussed in Section 5.1. Issues related to the usability and cognitive overhead problems in managing a so huge amount of information are as inevitable as complex, but we consider them out of the scope of this paper.

Save. While the presence of light constraints does not affect the actual text editing, it changes the operation of saving. Our architecture transforms such operation into *conditional saving*: whenever an user saves a page, validation is performed and according to its outcome a proper page is returned. Two outcomes are possible: the page is valid, and a simple acknowledgement is return to the user, or it is not, and a detailed report of errors is returned. Then the user can choose whether saving that page or not. Figure 3 summarizes the runtime behaviour of the SAVE action.

After submitting a page (step 1 in Figure 3), the wiki engine invokes each validator associated to that page on the submitted content (steps 2–4). Note that the position of validators, as well as the language used to implement them and/or configure the validation itself, are not relevant for the purposes of this section. The crucial point is the *logical separation* between the validation process and the common wiki work flow: such distinction makes it easy to apply a general model to different wiki clones by introducing few modifications and importing external validators or implementing them with less effort (see Sections 6 and 7 for more details about strategies for deploying and coding validators). Note also that no limitation is imposed over the number and variety of validators: different kind of light constraints can be checked over the same page by invoking different validators,

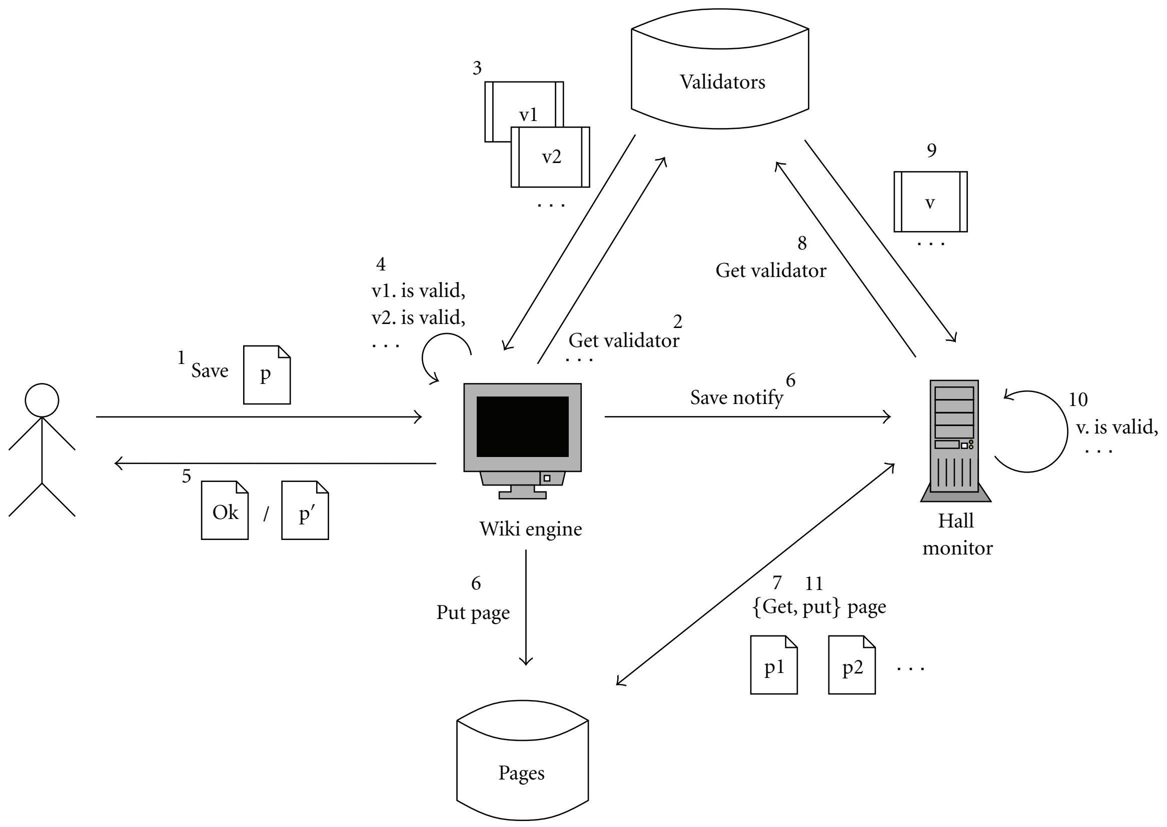


Figure 3: Runtime behaviour of the Save action.

each implementing its own strategies and each interacting with the wiki in a different way.

In case a page is valid, a confirmation message is delivered to the user (step 5) who goes on surfing (or continue editing) normally. On the contrary, in case a page is not valid, two options are provided (step 5 as well): the user can "forcibly save", being aware the page violates some light constraints or can fix errors and try saving again.

"Forced Saving" Is Crucial. It fully adheres to "the Wiki Way", as users can freely modify content and ignore validators, and allows users to save work in progress pages (not yet valid) or intentionally invalid pages (for instance, as examples of common errors and bad practices). For these reasons, we claim that collaboration is not hindered when adopting our approach.

When a user accepts saving a page, two events are triggered (steps 6): the new page is stored into the wiki page repository (notice that it implies storing both its content and its validation status) and a notification is sent to a component we call hall monitor, which is a process running in background that addresses context-related issues. As discussed before, validation is not limited to a single and isolated page, but rather a global process that can involve sets of pages up to the whole wiki. Then, running validators only on the submitted content is not enough, since changes can affect validity of other pages too. Our solution is notifying save events to the hall monitor and letting it invoking validators over each page included in the current context. Note again that the overall architecture and the function of this new component do not change whether validators are deployed in the wiki itself or as remote services.

The hall monitor proceeds as follows: it identifies all validators associated to all pages in the context (steps 7–9), invokes them (step 10), and updates pages accordingly to the validation outcome (step 11). The latter action of updating does not trigger any further validation. The hall monitor works behind the scenes, while the user has simply received a saving confirmation message. This choice is motivated by the possible huge amount of pages involved in validation.

The presence of the hall monitor drives us into a very interesting field: analysing how versioning is affected by validation. In the classical wiki model a new version of a page can be created only by an editing session (actually some wikis allow users to group minor changes or adjacent versions into a single one) but such approach is not enough in our setting. Users, in fact, can be interested in knowing that a page changed its validation state but, as outlined so far, this can even happen without explicit modifications on that page. In a sense, there exist two overlapping and intermixing version trees (one directly created by the users and one derived from the validation processes) and users should be able to see both.

Adding such enhanced versioning functionalities to a wiki might be quite difficult. In fact, the implementation is rather invasive and strictly dependant on the internals of the wiki itself (how versions are stored, how versions are organized and displayed in the interface, how users can surf multiple versions, and so on). On the other hand, such a feature is optional and can be omitted without impacting on the overall LC architecture.

6. Deploying Validators

The general discussion of the LC framework was deliberately independent on the "position" of validators and their actual interaction with the wiki engine accessing them. The goal of this section is to give an overview of the various possibilities to deploy validators, while the next section will investigate how validators can be internally implemented.

Firstly, validators can be implemented as internal modules of a wiki engine. This solution is very viable as most wiki clones are open source and maintained by a lively community of developers. In most cases, wiki clones are natively designed to be customized: just think about MediaWiki extensions or TWiki plugins. Another advantage of this approach is the efficiency: embedded modules guarantee a faster access to internal data structures since no external communication is required and all actions are completely local. On the other hand, such a locality is a limitation. Besides being a matter of configuration and maintainability, considering that administrators have to manually install plugins for each wiki instance, it also opens some security problems on who associates validators to pages: in fact, validation can be a resource- and time-consuming activity and allowing all users to activate them could even lead to a denial of service scenario. Administrators hence need to integrate some controls in the wiki to prevent this risk, by enabling only trusted users to tune the page-validator association. Last but not least, internal modules do not allow users to share and reuse validation facilities among different wiki sites.

The best solution to mitigate the above issues is to externalize the validation process. Such an approach is in line with the recent evolution of the world wide web where an increasing number of tools are hosted by SaaS (Software as a Service) providers. Even word processors, spreadsheets, mail readers, and calendars are hosted "somewhere in the cloud" and can be easily accessed through a web browser. Suffice it to mention the Google Apps framework, (http://www.google.com/apps/) a rich set of web-based tools for personal use, companies, and educational institutions. In this context, a very central role is played by mashups, that is, applications which combine other available services into a new integrated one. Thousands and thousands of mashup applications are available on the web (see [31, 32] as introductory references).

The composition of external services into mashup applications has been widely studied in the literature. In [33] authors proposed a dynamic approach for composing external web services and checking on-the-fly whether or not they are compatible and satisfy user requirements. Such a user-driven approach relies on a clear distinction between the composition itself and the end-users interfaces. Other researchers proposed a semantically-driven discovery and composition [34]: OWL-based descriptions of services are combined into a shared knowledge-based software agents can search and integrate. Experiments have also been carried on the way users interact with mashup applications and authoring environments and outlined interesting usability issues [35] and editing patterns and regularities [36]. The overall trend is that (even nonexpert) users are getting more and more confident in assembling external services.

A similar approach can be used for validators: they become remote services that multiple wiki engines can invoke to verify light constraints. This solution requires implementers to customize each wiki in order to interact with validators, albeit only once and for all possible future validators, which was not the case with the previously discussed deployment possibility. That extension consists of adding a module that passes content to the validator, retrieves the validation report, and mashes it up into the rendered page. There are evident advantages for both wiki administrators and users: validators can be implemented and installed once for all, without requiring each wiki to run its own duplicate; different wiki sites can share the same validator, even if they are actually running on different clones; as a consequence, communities of users can share validation facilities and also foster the creation of new ones. Existing technologies are sufficient to make such an approach possible. Section 6.1 goes into details of an abstract API to let wikis and validators communicate.

A slightly different strategy can also be implemented to deploy fully external validators. Instead of coding the module that invokes the validator as an internal component of the wiki (that acts server side) the same functionalities can be implemented client-side through Ajax technologies. The client-side deployment of rich applications is a very common approach nowadays: [37] discussed issues and guidelines for partitioning web applications between servers and clients and outlined very well the contexts where some strategies are preferable to others; [38] proposed a model-driven approach to design and deploy sophisticated client-side applications. The model adapts "traditional" web engineering solutions to the new context of RIAs (Rich Internet Applications) and helps users in integrating synchronous/asynchronous communication, client-side DOM manipulation, and friendly interface widgets. Several examples of client-side applications can be listed: BrowserShield [39], for instance, inspects the dynamic content of HTML pages on-the-fly and detects vulnerability issues before running it in the browser. No intermediates are required and the overall computation is performed in the browser. Actually external services can also be invoked but they do not interact directly with the original HTTP request.

In the LC context, a client-side module can be added to the wiki layout, able to invoke validators, and to add on-the-fly a validation report to the page. The wiki engine could even be unaware of validation facilities or be limited to store a very few information about that.

In particular, the View operation is implemented as follows. The original page does not contain any information about validation. When loaded in the browser, a JavaScript module annotates the page, by invoking remote validators and manipulating the DOM. The Save operation changes accordingly. After the editing session, in fact, the client-side module invokes the validator and shows the validation outcome to the user that can select saving the page content or not. Note that the Ajax-based architecture makes also possible to activate validation facilities when previewing a page and, more important, even while editing it. In fact, validation reports could be built "behind the scenes" and updated periodically (or on demand) through HTTP requests and DOM manipulations. Yet, usability issues must be taken into account (how often a validation report should be updated? where to show that information? how to integrate it with the legacy wiki editor?), but interactivity and user feedback can be greatly improved.

Another aspect is worth remarking: very few modifications to the wiki engine are required in order to deploy such a solution. The overall validation machinery (or better, the interaction with external validators that actually perform validation) is completely coded in JavaScript and can be added by simply modifying the skin of the wiki itself and by overriding the default handlers for very few events. The same code can even be used across multiple wiki clones as it does not depend on the internals of the wiki.

Note also that in the scenario discussed so far the wiki is completely unaware of the validation process. Hybrid approaches are also interesting as a way to distribute the computation, optimize some actions, and improve user feedback. For instance, it would be possible to send both the page content and the validation outcome to the wiki engine and to store both of them in the internal database. Such a solution would make the information available for all users (without being computed at every access) but require more invasive changes to the server-side modules. In that case the client-side module has also to send a separate request for activating the hall monitor that will check all pages in the context (see Section 5.2) by invoking remote validators. A crucial aspect is that either client-side modules or server-side ones can invoke validators through the same remote calls and run the same validation processes. Further details about such a common API are provided in Section 6.1.

There is actually a further externalization possible for validation services. Instead of considering a validator as a component that interacts with a client-side module of the wiki, a validator might be a fully external agent activated on-demand after a wiki page is delivered to the browser. In that case, the validation outcome is merged on-the-fly with the original unmodified content into the final annotated page [40]. Such an approach opens up interesting perspectives as it makes it possible to share validation rules stored in public repositories, to personalize validation processes for different classes of users and to validate any (wiki) page regardless of access permission. On the other hand, its requirements and goals are very different from the ones of this work: while fully external validators are meant to produce an extra-layer that added a posteriori on the top of unmodified wiki pages, light constraints validators impact the internal wiki work-flow and aim at supporting both readers and authors to handle constraints.

6.1. A Common API to Deploy Multiple Validators. Designing a common API that allows validators and wikis to communicate is useful to identify precisely the set of operations and parameters validators have to support and to simplify the integration of new validators into existing platforms.

This section proposes an API that is shared for either internal or external validators. The actual communication between wikis and validators might be implemented via direct function calls, SOAP web services, Java RPCs or any other protocol. As far as this work is concerned, these solutions only differ in syntactic and implementation details. Note also that some frameworks reify SOAP/WSDL methods as real language methods, so that using the API can actually be agnostic on where the validator is deployed. We then discuss this API through a java-like syntax, without loss of generality.

The API is actually composed by one method, exported by each validator, that implements the `isValid()` abstract method discussed in Section 5.1:

```
public validationstatus isValid (

   wikiMarkup content,

   idPage page,

   wikiAPIType* API,

   wikiType wType,

   idPage* context,

   hashmap params

   )
```

As expected the method takes as input the page to be validated. Two approaches are actually available to pass that content: (1) a *push* mode, where the content of the page is directly passed to the validator in the `content` parameter or a (2) *pull* mode, where references to get that content are passed to the validator (in the `page` parameter) that actually retrieves it. Different APIs exist to interact even remotely with a wiki such as WikiGateway [41], WikiRPC [42], MediaWiki API, [43] or SocialText REST API [44]. They all provide a rich set of methods to get a wiki page, to save a new one, to get information about changes, versions, metadata, and so on. Validators can exploit those methods to retrieve the actual data they need.

Both the push and pull modes are useful. The first approach is more efficient and suitable for validators deployed as internal modules. It can be used for external validators as well. Such a solution comes to play in the Ajax-based context too. In fact, it allows the client-side module to even send the actual page content to the validators while the user is editing the page (or when previewing it). The "pull" mode, that is useful for the client-side AnnotatedView, would not be enough, as the content is not stored in the wiki yet, but only in the client-side data structures. The "pull" approach is slower as at least two HTTP requests are required for any validation process. On the other hand, it allows validators to select which data they actually want to retrieve and how. Note also that such approach allows validators to perform caching and apply customized efficiency strategies. In order to enable caching policies on the validator side, the parameter `idPage page` must be a pair name/value.

The parameter `wikiAPIType∗API` has been introduced to specify the set of APIs exported by the wiki and available to the validator. It is actually a string chosen among a finite set, as few wiki APIs exist that can be extended:

```
public enum wikiAPIType {

    “WikiGateway”, “WikiRPC”, “MediaWiki
    API”, “SocialText API”, ...

}
```

We also introduced a `WikiType wType` parameter to indicate the wiki clone invoking the validator. That value is used to parse correctly the page source code and to specialize the validation process in order to check features of specific wiki clones. Consider, for instance, a Semantic Media Wiki installation: although the API is the same of a plain MediaWiki, further validation can be performed over the semantic data unavailable on the plain wiki. Even the value of this parameter is chosen among a finite set that can be easily extended:

```
 public enum WikiType {

    “MediaWiki”, “SemanticMediaWiki”,
    “TWiki”, “SocialText”, ...

}
```

The context parameter is a set of references to get the pages of the validation context according to the data-model described in Section 5.1. Once again, the content can be retrieved via internal functions or remote calls according to the actual position of the validator. Finally the hash-map `params` has been introduced as a general mechanism to pass further values useful for the validation process. Examples are the length to be verified for a section, the number of expected items in a list, the exact URL to be checked in a link, and so on. Notice that such parameters are different from the ones expressed in the `context` that are actual wiki pages.

The `isValid(...)` method returns a structured validation outcome that includes rich information available to the wiki and the final users:

```
Class ValidationStatus {

    URL validator,

    ProcessStatus status,

    Boolean result,

    LocatedError* errors-list,

    IdPage* context,

    InputFormat input

    }
```

Some data have been introduced for debug purposes only. The value `validator` indicates the name or the URL (when it is a remote service) of the validator. The value `status` contain diagnostic data about the validation process itself. This parameter does not indicate whether or not a page is valid, rather whether or not the validation process was successful. The process could fail because of internal errors of the validator, errors in the network communication, missing data in the request, and so forth. For this reason, the output is further structured into:

```
Class ProcessStatus {

    ValidationCode code,

    HTML message

    }
```

The `Integer status code` is a code indicating the status of the process, while the `HTMLType message` is a human-readable explanation about that status, coded in HTML. Further details about their possible values are out of the scope of this paper.

The actual validation outcome is contained in the values `result`, `errors list`, and `context` that express rich information about *localized errors* and validation context. They do not need further explanation here as they follow straightforwardly the data model discussed in Section 5.1.

The localization of errors deserves some more discussion. Each localized error is in fact a pair

```
Class LocatedError {

    Integer location,

    HTML message,

    }
```

The value location indicates the offset to pinpoint the error in the page source. Since validators can access the source file in different formats (from a specific wiki syntax to XML, from WikiCreole to HTML) this value alone would not be enough. We then introduced the value `InputFormat input` (in the `ValidationStatus` object) that indicates the format of the source page used for validation. It will be used by the wiki module to locate errors and correctly display messages. Since all localized errors share the same input format, this parameter is global instead of being repeated for each error. The set of its possible values is again extensible:

```
public enum InputFormat {

    “WikiCreole”, “WikiCreoleXML”, “HTML”,
    “MediaWiki”, “TWiki”...

}
```

Finally, the value message contain a human-readable explanation of the error, for the final user. Error messages could be in the specific wiki syntax too. That solution would require validators to know details of multiple wiki syntaxes and to further convert messages. Using HTML messages makes possible a single strategy for all wikis and reduce complexity. Note also that similar considerations can be extended to the debug status message discussed earlier on.

7. Wiki Content Validation

There is no universal answer to the question of “how to code validators”. In fact, light constraints vary radically for their nature. A general classification is equally possible taking into account (i) how many pages are involved in the validation process and (ii) how much semantic analysis is needed to validate content. Section 3 identified two orthogonal axes to, respectively, cover these two aspects: scope and expressivity.

While the *scope* mainly impacts on the interaction between the validator and the wiki (single or multiple pages need to be retrieved, and in both cases it can be done through the API discussed in the previous section) the *expressivity* has great impact on the way the validator for a given constraint is implemented. The goal of this section is to classify the most feasible strategies to implement validators, dealing with different levels of expressivity of the light constraints.

Before going on, we need to answer a very related question: "who actually code the validators". The WikiWay would suggest us to allow any user to freely program validators. That would be even more appropriate if the coding itself could be done on the wiki site. Such a solution raises several issues. First of all, it is very difficult to find a language suitable for this purpose, due to the tension among language expressiveness and simplicity (in term of both syntax and semantics). A second issue is of course security: assuming that a silver bullet language can be found, we need to prevent malicious uses of validators which can easily provide denial of services. Actually, such approach has already been faced by the so called Community Programmable wikis [45], which allows any user to modify the code of the wiki engine itself, without arriving at satisfying solutions.

The most relevant issue here is a third one: average wiki users are not capable of coding validators and, more important, they are not expected to.

We in fact introduced in Section 4 a specific role called *tailor* to indicate those users in charge of managing validators. In [8], the authors noticed that, even when the whole community is affected by system customization and tailorability, it is very common that a restricted set of users actually perform such task. On the contrary, in [9] authors claimed that tailorability should be extended to all the users: yet, differences among the expertise of users exist and are required to exist, but the customization itself is improved by involving average users too. Though wikis are inherently closer to the latter approach, we believe validators management is still a task meant for expert users.

In our view, however, the tailors are not expected to know the internals of each validator. They are only required to know validators' interfaces and functionalities and to associate validators to pages, by using wiki-specific mechanisms. The actual implementation and deployment are delegated to programmers and sysops.

The distinction between common users and tailors proposed in Section can be then refined in four roles: common users, who access validation facilities, tailors who configure validators and associate them to the pages, implementers who actually code validators and make them respect the above mentioned API, and administrators who take care of the final deployment.

The implementers code validators in multiple ways according to the expressivity of the light constraint they are required to meet. Our preliminary analysis identifies three categories of validators: procedural, schema language, and semantic.

7.1. Procedural Validators. The first category includes validators implemented through procedural languages such as C, Java, or PHP. Obviously these validators allow implementers to code any validation process and give them great flexibility and control over the validation outcome. The validator for the spell checking scenario is a very simple example of such a category: an integrated Python module (or a spell-checker written in any other language) can be invoked in order to validate a page against a dictionary and a list of expectations, retrieved by the validator itself from the wiki.

Procedural validators are particularly useful to check interpages constraints (opposed to intrapage ones that only involve one single page) as they make it easier to analyze the overall wiki content space. Just think about the detection of (the lack of) broken links mentioned in Section 3 that requires the validator to access the wiki and check if the links being verified refer to existing pages or not.

One issue about procedural validators might be their interoperability: a module developed in a language may require some rework and intermediate processing to be used in a different environment. Code maintenance has also to be taken into account: any (even slight update) to a procedural validator requires recompiling and reinstalling it.

These problems can be mitigated by clearly distinguishing the actual process of validating the content from the process of retrieving that content. This second part can, and in most cases need to, be procedural but the first one can be done in a more declarative way, as in the case of schema-language validators.

7.2. Schema-Language Validators. The basic idea of such class of validators is to exploit validation languages for XML [27]: light constraints are expressed as rules of the schema language and wiki pages are validated against those rules by a legacy schema processor.

Such an approach provides a lot of benefits to different users, compared to the procedural one. First of all, implementers are not required to code procedures from scratch as the validation is delegated to another module: they have to write schema rules and let the wiki interact with the schema processor. Schema editors could also be very useful in writing these rules. Note also that updating such a validator means updating a schema document and does not require any recompiling or installation. Last, but not least, schema documents can be shared across different wikis and different users.

The problem, on the other hand, is that schema languages work on XML documents while most of the wiki content is stored in a plain text syntax. Recent developments in the world of wiki syntaxes solve such issue. WikiCreole [46] is a common markup language for multiple wikis. Implementation-dependent wiki syntaxes can be easily translated from/into WikiCreole, so that content can be transferred across a broad range of wiki clones. Recently, the WikiCreole committee also proposed an XML format for WikiCreole [47] and released some stylesheets to perform bidirectional conversions between XML and the plain text syntax. Thus, most of the wiki content can be easily expressed in XML and validated against XML-Schema or RelaxNG schemas.

Just a few of applications of such an approach are listed here, by using XML-Schema operators: the restrictions on the length of a section can be verified through simple types (in particular the `<xs:minLength>` facet), the correct format of a date can be checked through native data types (`xs:date`); the presence of sections or the order of subsections can be verified through complex types (`<xs:sequence>`, `<xs:choice>`); even the coexistence/coherence of some data can be checked via assertions and conditional type assignments introduced in XML-Schema 1.1 [48].

As expected, schema language validators do not cover all the wiki constraints users might be interested in. In fact, these validators can only process the content that wikis are able to export in XML WikiCreole. While solving this issue might be only a matter of time, a lot of other constraints cannot be verified even if the XML file contained all the required data: those constraints have a "higher level" of semantic expressivity.

7.3. Semantic Validators. Semantic validators are meant to be implemented when some semantic analysis of the wiki content is needed to verify a light constraint. Different and incrementally complex levels of semantic expressivity can be identified.

The first example we discussed in Section 3 was about template matching. When checking whether a page respects a given template, some knowledge about the role of each structural object is required. The validator has to recognize headings, sections, paragraphs, and in-line elements. The basic check consists of verifying if the objects in the page are all present and exactly in the same position of the template. That control can be implemented via a schema language validator too. A "smarter and more semantic" validator could also check whether the page contains the same information but uses different constructs: for instance, users might be interested in checking that a page about a book contains the name of the author and the publication date, regardless of the fact that they are organized in a table, a list or multiple paragraphs. Yet, such an approach requires some extra-information embedded in the page source code about each relevant data fragment. Several semantic wikis [16, 19, 24] provide final users with these semantic editing facilities (see Sections 3 and 2 for more details about them).

Semantic wikis allow users to create rich knowledge bases and provide them with advanced functionalities for surfing and reasoning about these data. Those community-driven knowledge bases express facts that are very often stored as RDF statements. The further level of semantic validators is then to verify light constraints on top of these statements. For instance, users might be interested in checking that in SemanticWikipediA [16] specific kinds of pages (e.g., those belonging to the category "countries") contain a specific statement (e.g., "... has currency"), or that a page about a paper contains [a statement with] the complete list of authors, and so on. Since the entire pool of statements is available, such a verification can be easily performed to spot missing data throughout the wiki.

The semantic verification can also be extended to multiple pages, in order to check semantic interpages constraints. This can be exploited for example to ensure that pages express a consistent set of assertions (for instance, they all indicate correctly the date of a given event, or the capital of a given nation, etc.). Semantic Web reasoners, such as Pellet (http://pellet.owldl.com/) come into play at this stage. Launching a reasoner on the set of RDF and OWL statements derived from a semantic wiki, in fact, it returns inconsistencies (besides new inferred statements) that can be translated into localized-errors in the LC validation report.

8. Implementation

We wrote a proof-of-concept implementation that showcases the techniques discussed so far to deploy and code multiple validators and to make them interact with multiple wikis. It is a distributed environment including one instance of MoinMoin [49] and one instance of MediaWiki [50] enhanced with LC validation capabilities (the source code is available at http://vitali.web.cs.unibo.it/lcwikis). Our goal is not to extend these wiki clones into full-fledged constraint-enabled wiki systems but rather to show the peculiar aspects of the framework, proving that

(1) validators can coexist with wikis without impacting the main work flow and without burdening noninterested users.

(2) validators can be deployed in multiple ways (internal modules, external web services, client-side extensions, see Section 6 for more details) without requiring invasive modifications to wikis.

(3) there is no interdependency between the strategies to deploy validators and to code them. A common API can be exploited to make one wiki invoke multiple validators.

(4) there is no inter-dependency between validators and specific wiki clones. Multiple wiki clones can then share the same validator by exploiting the above mentioned API.

The overall architecture is shown in Figure 4. We implemented three validators and made them communicate to either MoinMoin and MediaWiki through the API discussed in Section 6.1.

(i) V_1 is an internal module of MoinMoin checking whether or not a page is compliant to a given template. MoinMoin allows users to create new pages by copying the same structure and initial content of a template. Such a model, called *creational templates*, has been widely studied and compared to its counterpart *functional templates* in [23]. Wiki pages are weakly connected with creational templates as they can be completely modified, up to loose relevant information for that page. It is then useful to check if a page is still valid against a template even after its initial creation, in order to

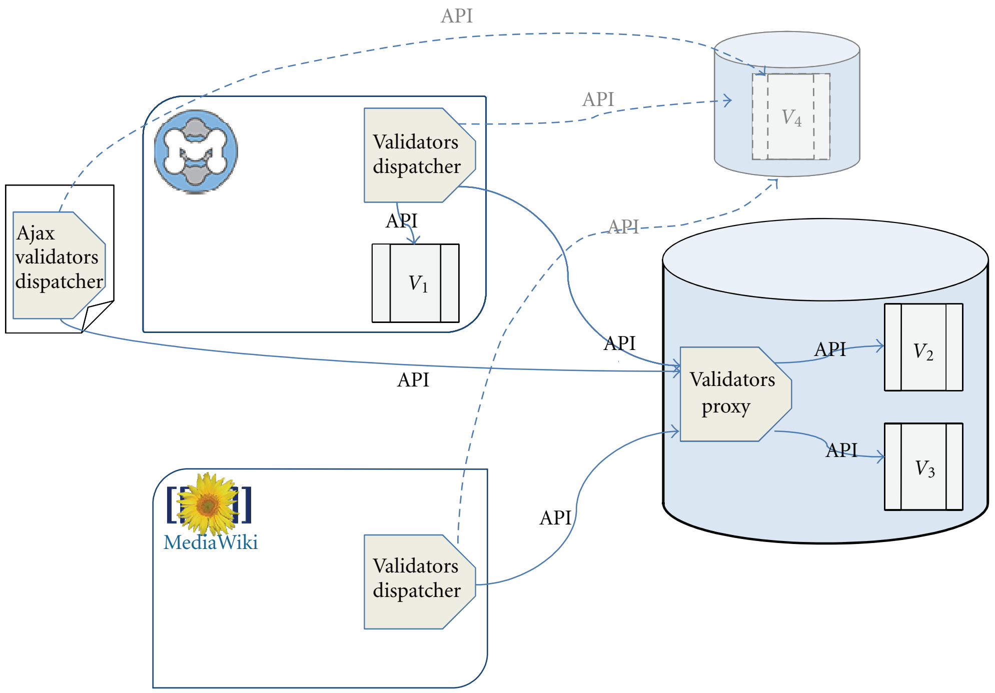
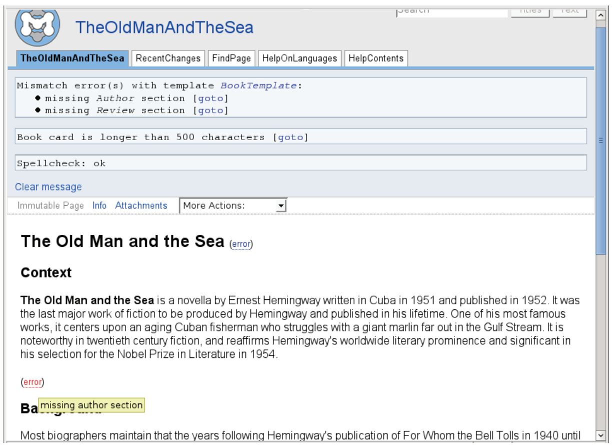


FIGURE 4: The overall view of the LC proof-of-concept implementation.



```
Mismatch error(s) with template BookTemplate:
    • missing Author section [goto]
    • missing Review section [goto]

Book card is longer than 500 characters [goto]

Spellcheck: ok
```

Clear message

The Old Man and the Sea (error)

Context

The Old Man and the Sea is a novella by Ernest Hemingway written in Cuba in 1951 and published in 1952. It was the last major work of fiction to be produced by Hemingway and published in his lifetime. One of his most famous works, it centers upon an aging Cuban fisherman who struggles with a giant marlin far out in the Gulf Stream. It is noteworthy in twentieth century fiction, and reaffirms Hemingway's worldwide literary prominence and significant in his selection for the Nobel Prize in Literature in 1954.

(error)

Ba missing author section

Most biographers maintain that the years following Hemingway's publication of For Whom the Bell Tolls in 1940 until

FIGURE 5: Screenshot of our LC templating implementation in the MoinMoin wiki engine. The shown tool tip is associated to the first of the template mismatch errors summarized at the top of the page, localized at the second (error) marker in the page.

check the presence of some data or the uniformity with other content. The validator checks if a page contains all required sections and subsections, as defined in the template. The verification is performed through regular expression matching on the page source.

(ii) V_2 is an external validator written in python and invoked via SOAP, in charge of checking the length of a given section in the page. The control can be easily performed by parsing the page source, extracting the required section (whose title is passed as a parameter) and running very common string functions.

(iii) V_3 is a second external validator invoked via SOAP, in charge of spell checking the content of a page. The actual spell check is a command-line utility installed on the server deploying the validator.

We actually combined validators V_2 and V_3 in the same web service, as shown in the right part of the picture. Such an optimization reduces the number of remote calls without requiring modifications to the overall framework. In fact, the

```
#format validate check-template (BookTemplate)
section_length (abstract, 500) spellcheck (WikiWords)

= The Old Man and the Sea =

= Context =

"The Old Man and the Sea" is a novella by Ernest Hemingway
written in Cuba in 1951 and published in 1952. It was [......]
in his selection for the Nobel Prize in Literature in 1954.

= Background =

Most biographers maintain that the years following Hemingway's
publication of For Whom the Bell Tolls in 1940 until 1952 were...
```

Algorithm 1: MoinMoin markup of a page equipped with validators.

wiki uses the same API to invoke a "proxy validator" that in turn invokes the actual validators by passing them the right parameters. Notice also that the picture shows a fourth and not yet implemented validator. The dashed arrows indicate that V_4 could be accessed through the same API (and from multiple wikis) without the mediation of a proxy.

The ValidatorsDispatcher is a component worth discussing more in detail. This is the core module that interacts with the wiki and invokes validators. In particular, it performs four main actions: (i) intercepts wiki events to trigger Annotated View and Conditional Save, (ii) finds the validators associated to a given page, (iii) invokes them through the API and (iv) enriches the page with the validation outcome.

This is the only part that needs to be added to a wiki in order to get LC facilities, unless of course the validator is implemented as an internal module. Most existing wikis provide standard *noninvasive* mechanisms to add extensions. The ValidatorsDispatchers for both MediaWiki and MoinMoin were in fact developed by exploiting such standard mechanisms. In particular, Mediawiki defines a set of Hooks that allow developers to override the wiki behaviour for specific events. The hook `wgHooks ["CustomEditor"]` is used to interpose the Conditional Save action after any editing session and to activate validation. Similarly, MoinMoin defines a set of EventHandlers, among which the `PagePreSaveEvent` and `PagePostSaveEvent` are used for implementing the Conditional Save, and an extensible set of Parsers invoked when a page or a fragment is written in a specific format. The MoinMoin dispatcher is in fact implemented as a parser that identifies the validators associated to a given page, and invokes them internally or remotely. More details about the actual syntax and rules to find validators are provided below.

Notice that in presence of a proxy validator some operations of the wiki dispatcher could even be delegated for optimization purposes. For instance, the dispatcher could pass the whole page to the proxy and let it to parse it, identify validators and invoke them separately. Such a solution makes the wiki dispatcher very simple and fast, reduces the number of remote requests, and externalizes the whole complexity into the proxy. On the other hand, it requires the proxy to be aware of the syntax used to associate validators and makes it dependent on the wiki. An aspect worth remarking is that it can be again implemented on top of the same API.

We also implemented an alternative approach for enabling MoinMoin with LC facilities, shown in the left side of the picture. The Ajax ValidatorsDispacther is a Javascript module added to the MoinMoin skin that performs very similar actions of a server-side dispatcher: intercepting events, invoking validators through the same API (the Javascript `XMLHttpRequest` is used for this purpose), and enriching the page with the validation outcome. As highlighted in Section 6, such a solution provides users a more interactive environment as the validation outcome is shown when they preview the page and even while they are editing it (DOM manipulations functions are exploited to update periodically a special page section). On the other hand, such a solution required us to patch the MoinMoin installation by overriding the hard-coded definition of the editor (basically we had to add some JavaScript declarations and an handler for the `onSubmit` event). Since very few lines of code were actually required and limited to a specific part of the original code, we consider it as an acceptable trade-off to get LC facilities.

Let us now briefly explain how users interact with the framework, using the MoinMoin installation as example. To associate validators to a page users are simply required to add a processing instruction at the beginning of that page. Validator and context are in fact stored as part of the page markup using MoinMoin `#pragmas`, the preferred place where to keep page metainformation. Algorithm 1 shows a snippet of markup of a page (about a review of Hemingway's book "The Old Man and the Sea") associated to all three validators V_1, V_2, and V_3. The first line contains the list of validators, each of which is passed a list of context pages and arguments.

The latter `(spellcheck)` validator ensures correct spell checking using a page named `WikiWords` as its exceptions list, the second (`section_length`) checks that the abstract is no longer than 500 characters, and the former one (`check-template`) checks the page against the template `BookTemplate`. This template organizes a page as a sequence of a title and four sections "Abstract", "Author", "Background", and "Review".

Figure 5 shows the Annotated View of the page after the validation process. The validation outcome is composed of two parts: a validation summary and a set of located errors. The summary is added at top of the page and reports, for each attached validator, whether the validation has been successful or not and the description of each error. Located errors are reported as links in the markup with (CSS) pop-up descriptions of the errors, pointing to the corresponding error entries in the summary. In fact, validators V_2 and V_3 show that the page meet the requirements on the length of the abstract and the spell checking. Instead V_1 spots two localized errors, namely, that sections "Author" and "Review" are missing in the page markup. Users are then notified that parts of the original templates are missing and are encouraged to fill in the appropriate information. A similar feedback has been returned to the user who last edited that page, before she chose to forcibly save.

Notice that the overall validation outcome has been assembled by the ValidationDispatcher from the outcome of the three validators, deployed as internal or external modules and invoked through the API. The dispatcher uses a configuration file to locate external validators and to compose remote requests.

In conclusion, the proof-of-concept implementation showed that adding support for light constraint to existing wiki systems is far from being challenging. The availability of standard extensions mechanisms for the most common wiki platforms can be exploited towards this goal, along with the general API presented in this paper. In fact, new and more powerful/complex validators can be added to this architecture following the same schema. At the time of writing we have not yet implemented any specific semantic or schema-language validator but we believe this to be a rather simple part of the problem at hand. More important, such sub-problems that can be addressed as independent units and their solutions can be easily "plugged" in the general LC architecture.

9. Conclusions

Drawing the word "wiki" close to the word "constraint" sounds as an oxymoron. The success of wikis derives from the openness and freedom of the wiki editing process: wiki users are entitled to modify any page at any time, apart from specific well-controlled cases. On the other hand, wiki users tend to define implicit constraints to improve content correctness and clearness. Those rules are usually expressed as community guidelines and manually enforced within the community. The goal of this research is to transform such an oxymoron into a novel fruitful synergy. This paper presented a general framework, called constrained wiki, to make that possible for different wiki platforms, without disrupting "the WikiWay" and without changing the way users interact with their wikis.

An aspect is crucial about the overall vision: the lightness of the constraints. Constraints are not meant to be strict prohibitions that prevent users from freely expressing their ideas and comments, rather to help them in creating uniform and high-quality content. The solution proposed here relies on a strong distinction between the actual wiki engine and a set of validators, in charge of verifying the respect of light constraints associated to the pages: by exploiting validators wiki systems can provide conditional saving and annotated viewing. Particular attention to the strategies for deploying and implementing validators was given in the central sections of this paper.

The framework does not mandate neither a specific wiki clone nor a specific validation process. On the contrary, we propose a general-purpose architecture that can be instantiated for multiple wikis and multiple validators, following the models and rules discussed in this paper. Our goal is to foster the discussion about lightly constrained editing within the wiki community, in order to create a free marketplace where users can easily share, improve, and add new validators.

References

[1] B. Tancer, "Look who's using wikipedia," Time 2007, http://www.time.com/time/business/article/0,8599,1595184,00.html.

[2] B. Leuf and W. Cunningham, *The Wiki Way: Quick Collaboration on the Web*, Wesley Longman, Boston, Mass, USA, 2001.

[3] F. B. Viégas, M. Wattenberg, and K. Dave, "Studying cooperation and conict between authors with istory ow visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*, pp. 575–582, ACM, 2004.

[4] Wikipedia, "WikiGnomes," 2010, http://www.en.wikipedia.org/wiki/Wikipedia:WikiGnome/.

[5] J. Grudin, "Computer-supported cooperative work: history and focus," *Computer*, vol. 27, no. 5, pp. 19–26, 1994.

[6] K. Y. Lai, T. W. Malone, and K. C. Yu, "Object lens: a 'spreadsheet' for cooperative work," *ACM transactions on office information systems*, vol. 6, no. 4, pp. 332–353, 1988.

[7] T. W. Malone, K. R. Grant, and F. A. Turbak, "The information lens: an intelligent system for information sharing in organizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*, pp. 1–8, ACM, 1986.

[8] W. E. Mackay, "Patterns of sharing customizable software," in *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW '90)*, pp. 209–221, ACM Press, 1990.

[9] A. MacLean, K. Carter, L. Lovstrand, and T. Moran, "User-tailorable systems: pressing the issues with buttons," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*, pp. 175–182, ACM Press, 1990.

[10] A. Haake, S. Lukosch, and T. Schummer, "Wiki-templates adding structure support to wikis on demand," in *Proceedings of the Conference Proceedings of the International Symposium on Wikis (WikiSym '05)*, pp. 41–51, ACM Press, 2005.

[11] C. Arnold, T. Fleming, D. Largent, and C. Lüer, "DynaTable: a wiki extension for structured data," in *Proceedings of the 5th International Symposium on Wikis and Open Collaboration (WiKiSym '09)*, pp. 26:1–26:2, ACM, Orlando, Fla, USA, October 2009.

[12] P. Thoeny, "TWiki: enterprise collaboration platform," 1998, http://www.twiki.org/.

[13] L. Argerich, E. Polidor, and G. Foster, "TikiWiki: CMS/Groupware," 2002, http://www.tikiwiki.org/.

[14] C. Trattner, I. Hasani-Mavriqi, D. Helic, and H. Leitner, "The Austrian way of wiki(pedia)! Development of a structured wiki-based encyclopedia within a local Austrian context," in *Proceedings of the 6th International Symposium on Wikis and Open Collaboration (WikiSym '10)*, pp. 9:1–9:10, ACM, July 2010.

[15] P. Buneman, J. Cheney, S. Lindley, and H. Müller, "DBWiki: a structured wiki for curated data and collaborative data management," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1335–1337, 2011.

[16] M. Völkel, M. Krötzsch, D. Vrandecic, H. Haller, and R. Studer, "Semantic Wikipedia," in *Proceedings of the 15th International Conference on World Wide Web*, pp. 585–594, ACM, May 2006.

[17] Y. Koren, Semantic forms 2008, http://www.mediawiki.org/wiki/Extension:Semantic_Forms.

[18] T. Kuhn, "Acewiki: a natural and expressive semantic wiki," *Proceedings of Semantic Web User Interaction at CHI*, vol. 543, 2009.

[19] M. Buffa and F. Gandon, "SweetWiki: semantic Web enabled technologies in Wiki," in *Proceedings of the International Symposium on Wikis (WikiSym '06)*, pp. 69–78, ACM, Odense, Denmark, August 2006.

[20] N. Kong, B. Hanrahan, T. Weksteen, G. Convertino, and E. H. Chi, "VisualWikiCurator: human and machine intelligencefor organizing wiki content," in *Proceedings of the 16th international conference on Intelligent user interfaces (IUI '11)*, pp. 367–370, ACM, 2011.

[21] A. Batliner, D. Seppi, S. Steidl, and B. Schuller, "Segmenting into adequate units for automatic recognition of emotion-related episodes: a speech-based approach," *Advances in Human-Computer Interaction*, vol. 2010, Article ID 782802, 2010.

[22] J. A. Bargas-Avila, O. Brenzikofer, A. N. Tuch, S. P. Roth, and K. Opwis, "Working towards usable forms on theWorldwideWeb: optimizing multiple selection interface elements ," *Advances in Human-Computer Interaction*, vol. 2011, Article ID 347171, 6 pages, 2011.

[23] A. Di Iorio, F. Vitali, and S. Zacchiroli, "Wiki content templating," in *Proceedings of the 17th International Conference on World Wide Web*, pp. 615–624, ACM, April 2008.

[24] S. Schaffert, "IkeWiki: a semantic Wiki for collaborative knowledge management," in *15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '06)*, pp. 388–393, ACM, June 2006.

[25] E. D. Lio, L. Fraboni, and T. Leo, "TWiki-based facilitation in a newly formed academic community of practice," in *Proceedings of the International Symposium on Wikis (WikiSym '05)*, pp. 85–111, ACM, October 2005.

[26] Dokuwiki, Constrained Wiki: the Wiki Way to validating content 33, http://www.splitbrain.org/projects/dokuwiki/.

[27] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, "Taxonomy of XML schema languages using formal language theory," *ACM Transactions on Internet Technology*, vol. 5, no. 4, pp. 660–674, 2005.

[28] G. Castagna, *Object-Oriented Programming: A Unified Foundation*, Birkhauser Boston Inc., Cambridge, Mass, USA, 1997.

[29] S. B. Palmer, "Rdfwiki," 2001, http://infomesh.net/2001/rdfwiki/.

[30] J. Hess, "Ikiwiki," 2012, http://ikiwiki.info/.

[31] J. Yu, B. Benatallah, F. Casati, and F. Daniel, "Understanding mashup development," *IEEE Internet Computing*, vol. 12, no. 5, pp. 44–52, 2008.

[32] V. Hoyer and M. Fischer, "Market overview of Enterprise Mashup tools," in *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC '08)*, pp. 708–721, Springer, 2008.

[33] Q. Zhao, G. Huang, J. Huang, X. Liu, and H. Mei, "A web-based mashup environment for on-the-y service composition," in *Proceedings of the IEEE International Symposium on Service-Oriented System Engineering*, pp. 32–37, IEEE Computer Society, 2008.

[34] H. N. Talantikite, D. Aissani, and N. Boudjlida, "Semantic annotations for web services discovery and composition," *Computer Standards and Interfaces*, vol. 31, no. 6, pp. 1108–1117, 2009.

[35] N. Zang, M. B. Rosson, and V. Nasser, "Mashups: who? what? why?" in *Proceedings of the Extended Abstracts on Human Factors in Computing Systems (CHI '08)*, pp. 3171–3176, ACM, 2008.

[36] J. Wang, H. Chen, and Y. Zhang, "Mining user behavior pattern in mashup community," in *Proceedings of the 10th IEEE International Conference on Information Reuse & Integration (IRI '09)*, pp. 126–131, IEEE Press, 2009.

[37] J. Kuuskeri and T. Mikkonen, "Partitioning web applications between the server and the client," in *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC '09)*, pp. 647–652, ACM, March 2009.

[38] P. Fraternali, S. Comai, A. Bozzon, and G. T. Carughi, "Engineering rich internet applications with a model-driven approach," *ACM Transactions on the Web*, vol. 4, no. 2, article 7, 2010.

[39] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: vulnerability-driven filtering of dynamic HTML," *ACM Transactions on the Web*, vol. 1, no. 3, article 11, 2007.

[40] A. Di Iorio, D. Rossi, F. Vitali, and S. Zacchiroli, "Where are your manners? Sharing best community practices in the Web 2.0," in *24th Annual ACM Symposium on Applied Computing (SAC '09)*, pp. 681–687, ACM, March 2009.

[41] B. Shanks, "WikiGateway: a library for interoperability and accelerated wiki development," in *Proceedings of the International Symposium on Wikis (WikiSym '05)*, pp. 53–66, ACM, October 2005.

[42] J. Jalkanen, "Wiki RPC interface 2, API version 2," 2006, http://www.jspwiki.org/Wiki.jsp?page=WikiRPCInterface2/.

[43] R. Kattouw, V. Vasiliev, B. T. Minh, and Y. Astrakhan, MediaWiki API, 2007, http://www.mediawiki.org/wiki/API.

[44] M. N. Van Ert, W. R. Easterday, L. Y. Huynh et al., "Global genetic population structure of Bacillus anthracis," *PLoS ONE*, vol. 2, no. 5, article no. e461, 2007.

[45] Community programmable wikis, 2006, http://www.purl.net/net/cpw/.

[46] M. Junghans, D. Riehle, R. Gurram, M. Kaiser, M. Lopes, and U. Yalcinalp, "An ebnf grammar for wiki creole 1.0," in *Proceedings of the ACM SIGWEB Newsletter*, vol. 4, ACM, 2007.

[47] M. Junghans, D. Riehle, and U. Yalcinalp, "An xml interchange format for wiki creole 1.0.," in *Proceedings of the ACM SIGWEB Newsletter*, vol. 5, 2007.

[48] S. S. Gao, C. M. Sperberg-McQueen, and H. S. Thompson, "Xml schema 1.1 part 1: structures," W3C Working Draft, 2009, http://www.w3.org/TR/xmlschema11-1/.

[49] T. Waldmann, "The MoinMoin Wiki Engine," 2000, http://www.moinmo.in/.

[50] WikiMedia, "MediaWiki," 2002, http://www.wikipedia.sourceforge.net/.