Constrained Wiki: an Oxymoron?

Angelo Di Iorio
Department of Computer Science
University of Bologna
Mura Anteo Zamboni, 7
40127 Bologna, ITALY
diiorio@cs.unibo.it

Stefano Zacchiroli
Department of Computer Science
University of Bologna
Mura Anteo Zamboni, 7
40127 Bologna, ITALY
zacchiro@cs.unibo.it

ABSTRACT

In this paper we propose a new wiki concept — *light constraints* — designed to encode community best practices and domain-specific requirements, and to assist in their application. While the idea of constraining user editing of wiki content seems to inherently contradict "The Wiki Way", it is well-known that communities of users involved in wiki sites have the habit of establishing best authoring practices. For domain-specific wiki systems which process wiki content, it is often useful to enforce some well-formedness conditions on specific page contents.

This paper describes a general framework to think about the interaction of wiki system with constraints, and presents a generic architecture which can be easily incorporated into existing wiki systems to exploit the capabilities enabled by light constraints.

Categories and Subject Descriptors

I.7.2 [**Document Preparation**]: Hypertext/hypermedia; H.3.5 [**Online Information Services**]: Web-based Services

General Terms

Design

Keywords

Wiki System, Validation, Assisted Editing

1. INTRODUCTION

The main factor of success of wiki sites is what inventor Ward Cunningham called "The Wiki Way" [4]: an open editing philosophy that allows users to freely write and collaborate on web content, without any restriction. In a sense, a wiki site can be considered a state-of-mind, an inclination shared by the users, rather than a collection of scripts and pages. This free notion of web editing, strengthened by some careful technical choices (direct editing in the browsers, minimality, versioning, ...) has made wiki systems commonly useful tools for single users, universities and firms.

Although authors are free to change and produce new content at will, we cannot help noticing that even the wiki editing process is often bound by some (implicit) rules. For instance, writers frequently create sets of pages (often explicitly grouped in wiki site areas) that share a predefined structure. Surprisingly, the most widely used approach to create and manage such structures is based on copy&paste and manual refinement and checking. Some solutions based on a partially constrained editing model have been investigated, mostly exploiting powerful yet flexible templating languages (see [8] for instance).

Templating control is only an example of a more general trend we observed: wiki users tend to agree on sets of non-written conventions that one or more pages must adhere to, and they then need ways for ensuring, or at least checking, that those requirements are really met. In a sense, the existence of *WikiGnomes* [20] (who work behind the scenes to fix minor nuisances) show this need, since much of their work is based on monitoring and adjusting to community best practices. Moreover, in the context of the success of grassroots information encyclopedias based on wiki technologies (like WIKIPEDIA [21] or World66 [23]), the issue of the quality of wiki site content is increasingly becoming relevant.

Apart from such spontaneous and implicit rules developed by the community, some wiki sites need to satisfy requirements that depend on the context they are being used in. Consider for instance the wiki systems that supply *in-lining*, i.e. mixing content written in varied formats within a document written in wiki syntax: SnipSnap [11] allows users to in-line a text representation of mind maps, organigrams and UML diagrams; OpenWiki [16] users can enrich pages with mathematical formulas written in MathML [15], and TWiki [18] users can include LaTeX mark-up commands, if a plug-in is installed. It is very useful to require that such in-lined text is correctly parsed and rendered by the wiki system. Yet, the existence of ill-formed documents does not cause irreversible problems (actually, inconsistent statuses are accepted for intermediate savings) but it would be preferable having integrated and automatic mechanisms to validate such content.

WikiSym'06, August 21-23, 2006, Odense, Denmark.

In this paper we analyze scenarios where non-written conventions over content are set up by the community as well as scenarios where inherent and context-based requirements affect wiki sites. Despite apparent differences, these issues have a common denominator: the need to express and enforce rules over the wiki content, without sacrificing the wiki editing paradigm. All these scenarios can be helped by what we have called *light constraints*, to emphasize their non-mandatory nature.

All the solutions we know are essentially *ad hoc* proposals for specific domains, hard-coded within systems, rather than instantiations of a general mechanism. No wiki system we are aware of offers support for representing and exploring different classes of light constraints and no generic model has been proposed yet. This paper has a double goal: on the one hand, to emphasize the strong relationship between wiki sites and content constraints, in order to foster such discussion in the wiki community; and on the other hand, to propose a solution based on a strong distinction between wiki engine and validation tools used to verify whether relevant light constraints are respected or not.

The framework we present can be instantiated whenever a form of validation on wiki content is required. The usual wiki workflow changes somewhat, since the VIEW operation is enriched with a validation report and the SAVE operation become a conditional step controlled by an intermediate validation process. It's worth remarking that validation respects the lightness of constraints, since users may still read and save invalid pages. Validation is meant to be helpful for both readers and authors, without sacrificing "The Wiki Way".

The rest of the paper is structured as follows. Section 2 discusses related works and various scenarios where light constraints need to be accounted for. Section 3 introduces our generic framework, describing the underlying data model and the role of validation. Section 4 focuses on actors (which may be either users or system components), discussing how their actions interact with the validation process. Section 5 presents a proof of concept implementation of our model in MoinMoin [9]. Section 6 concludes the paper and discusses future work.

2. LIGHT CONSTRAINTS SCENARIOS

Drawing the word "wiki" close to the word "constraint" sounds as an oxymoron. We noticed however that specific forms of constraints are surprisingly compatible with wiki systems and they can be fruitful exploited by them. Generally speaking, we define as an *informal constraint on wiki content* any kind of rule that a wiki page ought to satisfy. Our focus is thus on constraints which do apply to the *content* of wiki pages, not to other entities like URLs, metadata, keywords, user profiles, ...

Two different classes of informal constraints can be distinguished:

Hard Constraints: constraints that wiki pages must satisfy at any given time instant to be practically useful. An example is the need of having syntactically correct pages (i.e. pages which can be parsed by the wiki engine);

Light Constraints: constraints that can be (temporarily or not) violated, without inhibiting proper wiki runtime behaviour.

Light constraints are particularly relevant to wiki systems, since they can give fruitful help to the authors without sacrificing the wiki open editing philosophy. The lightness of such constraints plays a leading role: they can be verified providing detailed error reports, but users can ignore them. On the other hand, when verified, they improve the wiki authoring process.

In this section we present various scenarios where existing wiki systems have already dealt with light constraints (either in an implicit manner or by adopting ad hoc solutions), and new scenarios where the light constraints concept can be recognized. These scenarios taken all together emphasize that the relationship between wiki systems and light constraints is already pervasive within the wiki community. We believe that relationship deserves deeper investigation.

2.1 Scenarios from existing wiki systems

A very basic example of light constraints is the verification of the spelling of words. Some wikis are supplied with an internal tool that spell checks content on request. For instance, MoinMoin [9] integrates a Python module that validates a document against a dictionary and a list of exceptions. New words can be easily added modifying the exceptions using the wiki. Note that such spell checking is an optional operation that users can activate only on demand. Still, the lightness of the constraints is preserved, since users can save pages without caring about correct spelling. Similarly, DokuWiki [7] implements a 2-phase editing process that allows users to edit a page, to switch in correction mode and fix misspelled word (by invoking a server-side spell checker) and then save the final document. Moreover, DokuWiki is designed to help users, teams and work groups in producing documentation: verification can be particularly interesting in such kind of wiki applications, since pages are subject to some rules about correctness and well-formedness.

The correct management of intra wiki links is another field of application for light constraints. The "broken links" do not represent a real problem, since they are practically used to create new pages. Actually, two classes of dangling links have to be distinguished: those intentionally created to add new pages, and those created (often unintentionally) when deleting fragments or whole pages. A very interesting example in such area is PurpleWiki [12]. PurpleWiki is a wiki-clone that implements a fine-grained linking mechanism, through *purple numbers*: paragraphs, heading, lists and other text fragments are labeled with a number used to reference that elements. It's worth ensuring that any referenced purple numbers really exists, in order to avoid dangling links. Delete and move operations, as well as addition of new content, need to be carefully managed and can be once again bounded with light constraints. Moreover, it can be useful to express constraints about the non-existence of unreachable pages. Some wikis can retrieve a list (usually called *orphan pages*) of those pages, which are usually re-connected to the rest of the wiki site by a manual intervention. No wiki system we are aware of provides users a direct way for preventing the creation of orphan pages, after deleting a page or a fragment. However, their absence is a common and implicit requirement that should be fulfilled.

Light constraints can be also used to ensure minimum capabilities. In TWiki [18] whenever a new user registers, a page with the corresponding profile is created according to a given master page. Such master page allows users to auto-

matically set their profile, which can be erroneously modified preventing users to modify their own page and permissions. A light constraint is implicitly defined on TWiki pages, in order to prevent the cancellation of such access control data.

In addition, a new class of light constraints can be identified considering *properties shared* among pages belonging to the same group. Many times wiki users define spontaneously conventions on pages in order to ensure uniformity on wiki subsections: they define the structure of specific pages, the type of content, the order of the elements and so on. These requirements are often non-written and manually checked or simply ignored. For instance, in [5] authors discussed the adoption of wikis within an italian academic community, reporting examples of repeated pages, structures and patterns developed in that context.

Templating mechanisms ease the enforcement of such uniformity. WIKIPEDIA implements a powerful templating engine: to provide a page users simply instantiate a template assigning values to its variable. For instance, a summary table in the "oak" page is described with the following markup:

```
{{Taxobox
| color = lightgreen
| name = Oaks
| image = Quercus robur.jpg
| image_width = 240px
| image_caption = Foliage and acorns of
    ''[[Pedunculate oak|Quercus robur]]''
| regnum = [[Plant]]ae
| divisio = [[flowering plant|Magnoliophyta]]
| classis = [[dicotyledon|Magnoliopsida]]
...
}}
```

In this case, expressing constraints on the final structure of a page does not make sense, but it can be interesting to enforce the instantiation of a core set of template variables. Such constraints has not to be forced and always valid (in a page under construction or after an intermediate saving, it is fair having an invalid state) but they can be used to notify users of the need of more information (the same role played by "stub pages").

WIKIPEDIA gives us the opportunity of outlining a different form of light constraints, which ensure consistency among lists of elements shown in different pages. Consider for instance the set of states described in WIKIPEDIA: many different lists of these states can be found, ordered by name or by population, grouped by continent or by timezone and so on. All these lists are manually maintained and no check is performed to ensure they contain the same sets of elements. Similarly, the correctness of the order of elements in a list is not checked. Once more, such controls do not interfere with the editing process which remain spontaneous and free.

The dilemma between unstructured wiki pages and structured ones has been investigated in [8]. The authors stressed the need of structured information, considering it a way to help users in stating their ideas and comments. They introduced the concept of wiki templates, 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". The apparent contrast with The Wiki Way is solved using a tailoring process: users can freely modify the templates, so that no limitation is provided over the editing. Wiki templates embodied a different form of light constraints: instead of validating them after an editing session, they are enforced a priori. The possibility of freely edit templates makes such constraints light.

2.2 New scenarios

Before discussing how light constraints are expressed and validated in our model, it is worth introducing two possible new scenarios: WikiFactory and Miki. Light constraints management turned out to be generic enough to address domain-specific issues we found in two unrelated projects.

2.2.1 WikiFactory

WikiFactory [6] is a framework designed to automatically produce domain-oriented wiki sites. The idea came by examining how users use to create similar pages and structures, in wiki sites for a specific domain. What we observed is that most of that work is completely manual, time-consuming, and error-prone. On the other hand, each domain suggests a set of pages, links and data structures that each wiki site used in that domain should have. An example is a wiki site for a university department, which is supposed to have a page with the list of professors, for each of them a brief description and a list of courses, and for each course information including an enrollment page for the exam. Instead of manually creating such pages, we proposed to automatically produce them from an ontological description of the departments and a set of instance data.

WikiFactory is a Java application based on semantic web technologies which takes in input an OWL document describing a domain-oriented wiki site, and produces pages for a specific wiki engine. Even if the very early implementation produces only content for MediaWiki, the architecture is independent from the final wiki engine. The core of the application is the ontological description created by an ontology expert, in charge of writing on OWL document about a specific domain, and a domain expert, in charge of completing such OWL document with data about a specific instance of that domain.

At the first installation, such ontology is actually transformed into a set of consistent pages. An important open issue of WikiFactory is how to preserve the consistency between the ontology and the wiki pages upon page editing. We do not want to prevent users to freely modify content, in order to preserve The Wiki Way. Still, it would be desirable updating the ontology according to user requests. Light constraints can be really helpful in such scenario: consistency can be described as a light constraint, so that whenever a user changes a page she can be notified about the consequence of the change. As expected, many more issues need to be investigated about the techniques for updating the ontologies, for versioning changes, for solving conflicts and so on, but even this scenario shows the flexibility of a model based on light constraints.

2.2.2 Miki

Miki is the codename of an ongoing effort for creating an infrastructure for collaborative authoring of formalized mathematics on the web. In formalized mathematics, mathematical concepts like definitions, lemmata, and theorems

are encoded in some formalism — e.g. higher order logics or set theory — that enables mechanical checking of proof correctness. Software artifacts like proof assistants and automatic theorem provers are used to produce (respectively interactively and automatically) the formalized versions of mathematical concepts. Such concepts are stored in machine understandable formats which are more and more frequently made available through the web. Examples of web-enabled digital libraries of formalized mathematics are [1, 2, 19]. Despite the web-friendliness of such libraries for browsing purposes, their authoring process is far from The Wiki Way and is often centralized and managed by the developers of a given proof assistant or theorem prover. Miki aims to import The Wiki Way in the authoring process of libraries of formalized mathematics.

Interesting challenges to the wiki community are posed by Miki. Some of them are related to the usability of wikis for editing content which requires high interactivity, like *scripts* (the list of commands which are fed into a proof assistant to create a proof), but are not relevant to our discussion. Others are related to the logical consistency of what is shown to the user. Concepts from the library are not isolated, but can be linked together by a *requirement* notion; for example: the proof of a theorem on algebraic ring structures is likely to require the availability of a definition of rings in order to be properly proof checked. If the definition of groups changes, it is likely for the proof to need changes as well, or it will probably fail a proof checking test. A user working on such broken pages needs to be aware of their brokenness to avoid him trusting wrong mathematical results.

Since libraries of formalized mathematics are also often used for presenting formal mathematics, they also support free form pages (sometimes called *theories*) which are used to describe mathematical results and which contain references to formalized concepts. During rendering those references are inlined and showed to the user. This poses the additional need of verifying the logical consistency of a set of concepts, giving feedback to users who are watching theory pages.

In Miki we encoded the logical consistency of mathematical concepts as a light constraint, and we were able to address both the above issues. The design of Miki, which is still in early stages of development, is an instantiation of the generic architecture we present in this paper.

2.3 User experience: requirements

In Sections 2.1 and 2.2 we gave evidence of the existence of light constraints in real world collaborative editing tasks. No wiki system we are aware of support them in any way. For this reason, even what does "supporting them" mean is an unanswered question. In this work we give one answer, hoping to foster discussion on this subject in the wiki communities.

We claim that an authoring system is said to support light constraints if:

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

(b) it helps the work of tailors (the users which coordinate the collaboration on set of pages) enabling the description of constraints and their association to pages.

Instantiating such a system in the setting of wiki system poses additional requirements on the way users should interact with it. All boils down to respecting the wiki way of working and is expressed by the following set of requirements:

(1) Unconstrained Saving: *authors should not be forced to resolve all constraint violations in order to save a page.* In apparent contrast with the purpose of light constraints support, this requirement stresses the fact that constraints are meant to help authors while not diminishing their freedom.

(2) Freedom of Constraints Definition: *tailors should be able to work on constraints and associate them to pages using classical wiki techniques.* This include providing simplified markup for constraint definitions, and versioning of both constraints and of their association to pages.

(3) Constraints Visibility: *information on constraints should be visible to all users.* This requirement is meant to provide visibility of all information relative to constraints (which are associated to a given page, which are violated and which aren't, ...) to all users, i.e. not only to authors during page editing. This would help diminishing the gap between page producers and page consumers (the more is visible that something need to be fixed, the more is likely that someone will fix it), and ease WikiGnomes' lives.

We claim all these requirements are fulfillable in a wiki system and in the next sections we describe the skeleton of such a system.

3. DATA MODEL

Our proposal to the wiki community for encoding light constraints in wiki system is to represent them as *validators*: computational entities able to decide whether a wiki page fulfills a given light-constraint. Validators will be associated to pages. VIEW, SAVE, and other actions on pages will be changed to exploit validation 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.

This section and the next one are devoted to describing a generic architecture which implements this proposal. Here we present the concepts and the static entities which characterize it (what we call the *data model*) while in Section 4 we focus on the actors which compose it and on how the behaviour of the usual wiki actions is changed to exploit validation.

Figure 1 is an Unified Modeling Language (UML) sketch of the data model. The basic entity of all wiki system is the page, which is reported on the left of Figure 1. At the very minimum it is characterized by three properties:

Markup: a text string containing the actual wiki markup the user see 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;

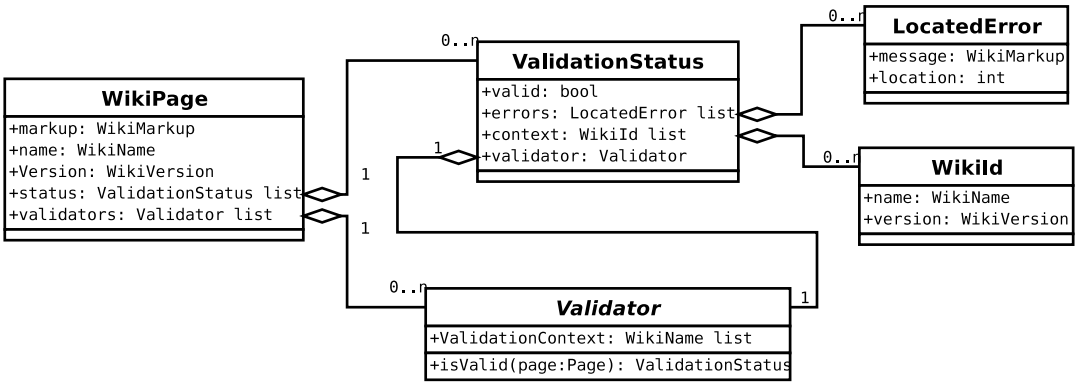

Figure 1: UML sketch of the data model.

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

Let's consider the spell checking scenario of Section 2.1 (an example that will follow us in this section), a sample page might be represented in the minimal data model as follows (using a syntax inspired by the object as record metaphor):

```
Page about = {
 markup = "This peper rocks, follow TheWikiWay";
 name = "AboutThisPaper";
 version = "314";
}
```

A validator is intuitively a function encoding a single light constraint. A validator can be applied (via the `isValid` method) to pages and returns either a statement that the page is valid (with respect to the light constraint encoded by the validator itself) or a statement that it is not. This computational aspect of validator is needed in order to be able to take decisions based on its outcome as it will be done, for instance, for conditional saving. In case a page turns out not to be valid, the validator returns a list of localized error messages: textual messages which are bound to particular characters in the wiki markup. This choice is motivated by the need of guiding authors toward fulfilling constraints: localized errors are easier to spot than global ones and hence faster to solve (at the very minimum the spotting time is reduced). Note 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.

In order to respect requirement (2) (see Section 2.3), the association among a page and its validators should be part of the information which are editable by users, hence the following property on pages:

Validators: a list of validators, one for each light constraint which should be enforced on the page.

In the frequent case of wiki systems supporting hierarchical structuring of the page namespace, the validators property is likely to be inherited to enforce a common set of light constraints to a particular area of a wiki site. Addition and removal of constraints on large page sets, for example, can then be performed changing a property in a single (root) page.

Requirement (3) is implemented in the data model by keeping track of the *validation status* with the following property:

Status: a list of validation statuses, one for each validator, which were associated to the owning page when the last validation attempt has been performed. The key properties of a status are `valid` (a boolean value denoting the success of the `isValid` invocation), `errors` (the list of located errors, which is meaningful only if `valid == false`), and `context` (a validation context, discussed below).

From several of the scenarios discussed in Section 2 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.[1] In the spell checking scenario for instance, the validation is parametric on a dictionary external to the wiki site and on an extra page containing additions to the dictionary. That page is likely to be editable by users. A *validation context* represents the set of wiki pages (referenced by their names) on which a validator is parametric. Note that pages referenced from validation contexts are non-versioned, since to better ensure liveness of wiki content validation will always be attempted using their more recent versions.

Validators are parametric in their validation contexts. In a sense, we think about validators as taking in input both the page they should validate and the validation context. This way we can for instance have parts of a wiki sites spell checked using a list of geeky words as exceptions and other parts using a list of biological terms.

In order to fulfill requirement (3), all information about the validation status should be available to users. That's explain the `context` property of validation statuses. 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

[1]The latter form of additional information should however be minimized in order to preserve the ability of users to influence validation.

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 need to be available to users, that's way we record the *actual validation status* as a property of validation status. The actual validation status is a list of page references corresponding to the validation context, together with their version. This way it will always be possible to retrieve the exact set of pages which leaded to a particular validation outcome.[2]

4. ARCHITECTURE

The presence of validators changes substantially the workflow 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 state (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.

We designed a general-purpose architecture that allows users to associate and run sets of validators on wiki pages.

Various entities compose such architecture:

Roles Played by Users:[3]

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.

Software Components:

Wiki Engine: the actual wiki engine working as any other wiki clone do, but also in charge of invoking validators.

Validators: entities that actually validate pages content, as discussed in 3.

Batch Validator: stand-alone component which verifies whether or not changes on a single page affect validation of other pages.

In order to explain the purpose of each entity in our architecture, we discuss individually the two main operations of a light-constrained wiki, VIEW and SAVE, focusing on both their differences with the corresponding wiki operations and the architectural choices behind them. Later we discuss how validation affects other wiki operations, like versioning and diff-ing.

[2]Note that changes to external information used by validators can't, in general, be captured in the same way: yet another good reason to keep as much validation information as possible represented as wiki pages

[3]as often happens, the same user can play different roles at different times

VIEW. Our architecture transforms the view operation in *annotated viewing* whenever a user accesses a page. Its 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.

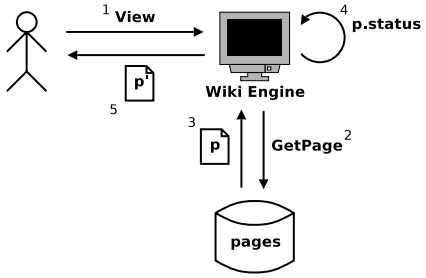

Figure 2: Runtime behaviour of the VIEW action.

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[4] databases (as WIKIPEDIA) to plain text file (as most wikis), from RDF tuples [17] to Subversion[5] repository [10]. 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.

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 outcome 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 example of Miki discussed in Section 2.2.2. A page can be invalid because some lemmas referred by that page are not correct and, in turn, even these lemmas can be inconsistent because of other related properties. It is very useful to show users such a chain of relationships and consequences. Moreover, errors should be localized, as discussed in 3. 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.

A final remark is worth: such rich information and, above all, their availability for the whole wiki community even sim-

[4]`http://www.mysql.com`

[5]`http://subversion.tigris.org`

plifies and speeds up the production of pages that fulfill light constraints, since any user can easily discover errors or imperfections and can spontaneously fix them. In a sense, providing a validation feature even for the VIEW action simplifies the life of WikiGnomes and paradoxically improves both content and sharing habits among wiki users.

SAVE. While the actual text editing is not affected by the presence of light constraints, saving is. 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 (similar to that shown in Figure 5). Then the user can choose whether saving that page or not. Figure 3 summarizes the runtime behaviour of the SAVE action.

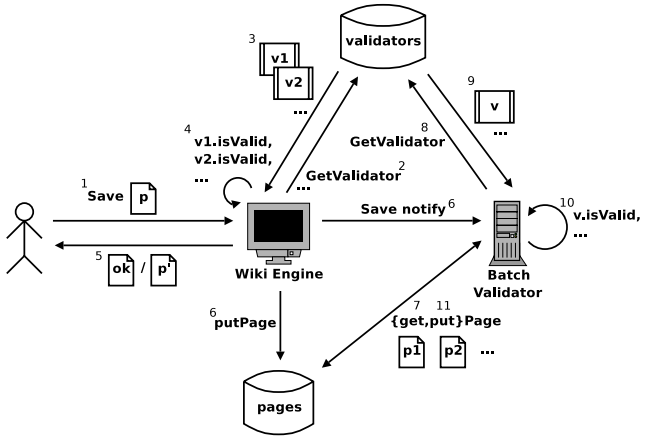

Figure 3: Runtime behaviour of the SAVE action.

After submitting a page (step 1 in Figure 3), the wiki engine gets all the validators associated to that page (steps 2–3) and runs each of them on the submitted content (step 4). The internal structure of validators, as well as the language used to implement them and/or configure the validation itself, are not relevant at this point. What is relevant here is the strong separation between the validation process and the common wiki workflow: 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. 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, and different communication protocols and validation engines can be exploited.

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 "forcedly 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 and a notification is sent to a component we call *batch validator*. Note that storing a page in the repository does not mean only storing its content, but also its validation status. The batch validator is a process running in background we introduced in order to address 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 a listening daemon and letting it running validators over each page included in the current context. Details about the communication protocol between the wiki engine and the batch validator are not relevant here.

The batch validator proceeds as follows: gets all validators associated to all pages in the context (steps 7–9), executes them (step 10), and updates pages accordingly to the validation outcome (step 11). The latter action of updating does not trigger any further validation. Note that the batch validator 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 batch validator 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.

Consider the spell checking example: it can be very possible that, adding a new word to the dictionary, an existing page becomes valid; the two statuses of that page, before and after having added the new word, are worth be traced and reported to the users. A new version of a document should be created either after an editing session or after an automatic update done by the batch validator. Yet, these two kinds of versions are conceptually different: in a sense, there exist two overlapping and intermixing version trees and users should be able to see both.

Inevitably even the DIFF operation changes, since a DIFF between two versions does not mean comparing only their content, but even their validation states. At first glance, it means simply producing a more complex DIFF page composed by two parts: one devoted to show changes in validation state and another one dealing with content (as expected, one of these part can be empty). However, a more complex issue need to be addressed, once again because of the validation context. Such a DIFF should provide users precise references to the content modifications that causes that local change, even if they occur in other pages. Obviously the richness and granularity in the DIFF output opens the doors to a series of complex issues related to usability and cognitive overhead, but we consider these aspects out of the scope of this paper.

Last but not least, we need to add details about the association between pages and validators, as well as the configuration and encoding of the validators themselves. We introduced a specific user role called *tailor*. The term "tailor" indicates the ability of cutting out validators and configuring them for specific (class of) pages. In [13], 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: although that work primarily focused on software customization, the same observation can be extended wherever a specific and quite difficult configuration task has to be accomplished, and highly-skilled users or domain experts need to be involved. On the contrary, in [14] authors claimed that tailorability should be extended to all the users: yet, differences among user expertises exist and are required to exist, but the customization itself is improved by involving average users too. We are still investigating which level of tailorability is suitable for light-constrained wiki systems, also considering that boundaries among roles are blended in the wiki setting.

Actually, considering the generality of our architecture, a wide spectrum of tailors can exist: on the one edge, a validator can be hard-coded within the system, so that a tailor can at most select validators; in the middle, validators can be parametrized so that a tailor can set parameters (besides associating pages); on the opposite edge, a validator can be completely programmable, and a tailor can completely decide its internal behaviour. The extreme solution consists of coding directly the behaviour of validators through a wiki syntax and allowing any user to describe such behaviour.

5. IMPLEMENTATION

We wrote a proof of concept implementation of the architecture described in Section 4 which adds validation capabilities to MoinMoin [9]. Its aim is not to extend MoinMoin into a fully general constraint-enabled wiki system, but rather to show the non-invasiveness of a similar extension to a popular wiki system.

The main component of the implementation is a new parser, which in MoinMoin terminology defines one of the possible formats a wiki page (or fragments of it) can be written in. To use the parser, our being called `validator`, is enough to add a processing instruction at the beginning of a page (or of a delimited fragment). It receives as additional arguments a list of validators, each of which can be in turn be passed a list of validator-specific arguments. Figure 4 shows a snippet of MoinMoin markup of a page which uses our parser. It represents the markup of a wiki page on an hypothetical wiki site used to coordinate paper submissions to a conference on the wiki topic. Line 1 requires the page to be validated by two validators. The former (`abstract_length`) checks that the abstract is no longer than 200 words, while the latter (`spellcheck`) ensure correct spell checking using a page named `WikiWords` as its exceptions list.

The validation status is stored, together with the list of validators associated to a page, encoded in the `extra` field of the `edit-log` file associated to each page. `edit-log` is the place where MoinMoin stores the metadata associated to a page. When a page which uses the `validate` parser is accessed its validation status is retrieved and used to annotate the page markup. Annotations come in two flavours: a validation summary and a set of located errors. The summary is added at the end 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. Figure 5 is a screenshot of a MoinMoin page rendered via `validate` showing both the validation summary and one located error (at the beginning of the abstract). A similar feedback has been returned to the user who last edited that page, before he chose to forcedly save.

After markup annotation, the `validate` parser acts as a "proxy" invoking again the internal MoinMoin machinery to discover the appropriate parser for the annotated markup and render it using the abstract formatter which gets passed to parser.

Validators are stored as Python scripts server side, are loaded using the `importPlugin` mechanism of MoinMoin, and are invoked when a page is saved, possibly requiring forcing by the user in case of validation errors.

The batch validator has been implemented in Python as a daemon with XML-RPC interface. Once notified of a save, it starts digging MoinMoin pages to discover which pages have the saved on in their validation context.[6] Each of them is then retrieved (using wiki RPC interface [22] `getPage`), (re-)validated, and stored in case of validation status change (using wiki RPC `putPage`). In order to push toward the wiki the information about validation status changes, the implementation of `putPage` should support the `attributes` argument. In MoinMoin that was not the case, we patched it and reported the bug upstream.

Yet being "proof of concept", our implementation shows that adding support for light constraint to existing wiki systems is far from being challenging. The peculiarities of MoinMoin we exploited are just a few: the extensibility of its markup and its metadata, and its wiki RPC interface. All are features available in the implementation of many existing wiki systems. The only parts of MoinMoin code we actually had to patch are the reaction to a "save" request (adding user notification if she attempted to save an invalid page without explicitly forcing the save) and a save time hook which notify the external batch validator. Taken together they sum up to less then 100 lines of code. The batch validator is fully reusable for other wiki systems (assuming they implement the wiki RPC interface).

6. CONCLUSIONS AND FUTURE WORK

The openness and freedom of the wiki editing process has a strong impact on the final pages: they are frequently updated, rich, and continuously improved, but also under-controlled and flawed. We noticed that wiki pages improve their correctness and clearness, when a set of rules are enforced by the community or by the wiki system. These rules cannot be strict prohibitions that prevent users from freely expressing their ideas and comments, rather they should help them in doing work that would be otherwise done later or never. In this paper we referred these rules as light constraints and we proposed a general framework to manage them. Our goal is awakening the community to the existence of a strong connection between wikis and constraints,

[6] this is actually implemented naively using wiki RPC interface method `getAllPages`, of course this can be optimized having the batch validator keeping an internal record of validation contexts

```
#format validate abstract_length(200) spellcheck(WikiWords)

#format wiki

= Constrained Wiki: an Oxymoron? =

'''Author(s)''': ["Angelo Di Iorio"] and ["Stefano Zacchiroli"]

'''Abstract''':

  ["The Wiki Way"] is in apparent contrast with any kind of editing-time constraint. Nonetheless it
  is well-known that communities of users involved in wiki sites have the habit of establishing best
  authoring practices, and it is a frequent need of domain-specific wiki system to enforce some kind
  of well-formedness on page content. A general framework to think about the relationship of \WIKI{}
  system with constraints is missing.
  ...
```

Figure 4: MoinMoin markup of a page equipped with validators.

and provide a first general model that can be applied to heterogeneous scenarios.

Basically our solution 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*. The proposed solution does not change the user editing experience, as opposed to other solutions like [8].

In the case of spell checking, each page can be associated to an external validator that actually spell checks content, looking for a dictionary page, whenever that page is saved. In the case of inconsistent and unordered lists of WikipediA, each page can use a validator who knows which other pages have to be consistent with the current one: such validator verifies whether all those lists contain the same elements. Even the new scenarios we described can be addressed: for WikiFactory, a validator associated to a page recognizes the template for that page and verifies whether that page matches it or not. Similarly in Miki the consistency among mathematical concepts is checked by external validators that run over the whole mathematical repository and give a response back to the wiki engine. A point-to-point description of the remaining scenarios is as useless as boring, but it might not be difficult to instantiate such observations to each of them.

An analysis of our architecture can be completed verifying whether all the requirements for a constraint-enabled wiki are fulfilled. In Section 2.3 we identified three such requirements: (1) unconstrained saving, (2) freedom of constraints definition, and (3) constraints visibility. Many times in this paper we stressed (1) and (3), showing how no strict rules are really imposed on the editing process and showing that all validation information can be given to the users (for instance through the View and Diff operations and through the localized errors list).

On the contrary, (2) is tricky and not properly addressed. The Wiki Way would suggest us to allow any user (or better tailor) to freely program validators, directly 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). Scenarios like Miki shows how complex can be validation needs. 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 be easily provide denial of services. Actually, such approach has already been faced by the so called Community Programmable wikis [3], which allow any user to modify the code of the wiki engine itself, without arriving at satisfying solutions. A compromise solution can be achieved by "tailoring", that is allowing only a subset of trustworthy users to configure and actually write validators. As discussed in Section 4, limiting users' tailoring can be, on the one hand, a very good solution to capitalize skilled users work but, on the other hand, a restricted approach which limits average users potentialities. Note that we do not claim that such a customization is so dangerous to be impossible, rather we think that a more detailed and deeper discussion is required.

Our next step will be investigating such relationship among average users, tailors, languages for programming validators and open editing philosophy. In particular two future directions seem to be equally valid. On the one side, we will try to figure out a general language simple, safe, but enough expressive to allow user to define validators in frequent occurring scenarios. On the other side, we will try to figure out small (different) languages suitable for specific domains (for instance, we are discussing a language to define and verify templates for scenarios similar to WikiFactory). Our point should be clear now: at first glance constraints and wikis seem to be incompatible, but after a more careful analysis, they can coexist in an interesting and fruitful oxymoron.

7. ACKNOWLEDGMENTS

We would like to thank David G. Durand for the fruitful discussions we had with him on the topic of constrained wikis and for its feedback on this paper.

8. REFERENCES

[1] S. Allen, M. Bickford, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. Fdl: A prototype formal

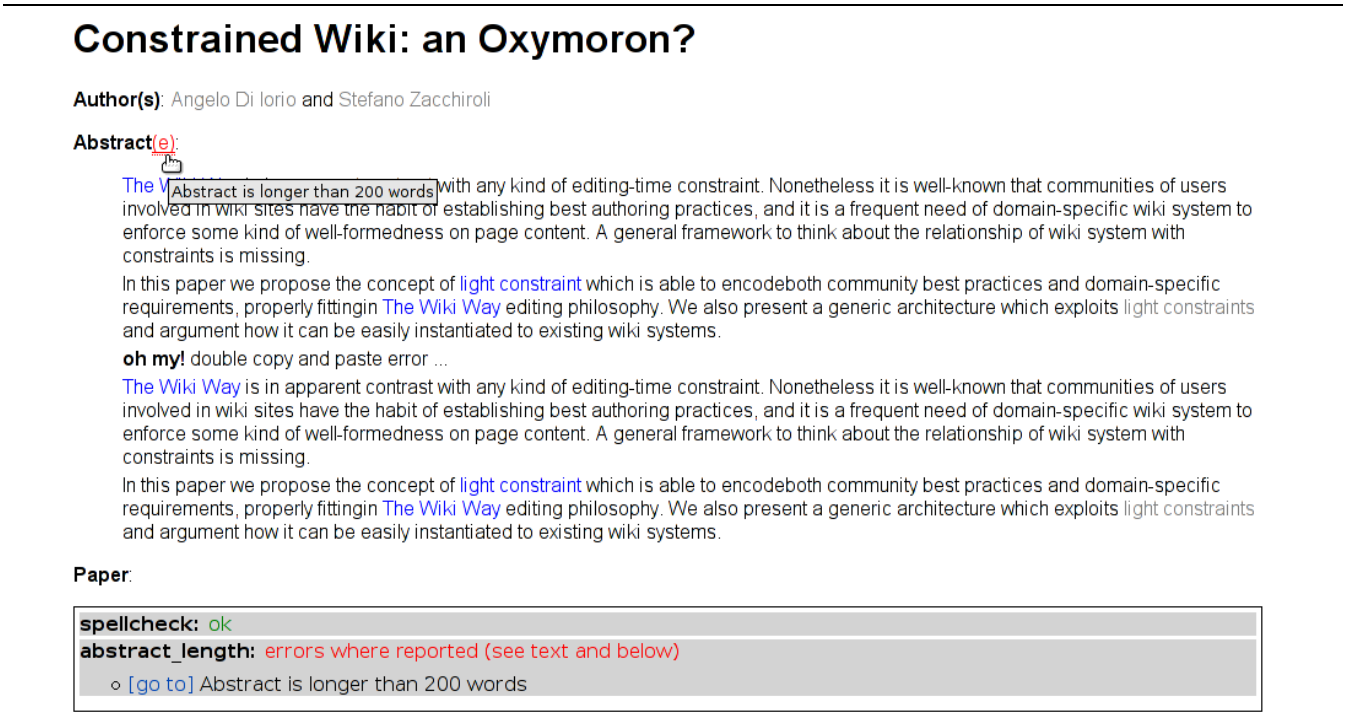

Figure 5: Screenshot of MoinMoin's VIEW action extended with validation support.

digital library. Technical Report TR2004-1941, Cornell University, 2002.

[2] A. Asperti, F. Guidi, L. Padovani, C. Sacerdoti Coen, and I. Schena. Mathematical knowledge management in HELM. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):27–46, May 2003.

[3] Community programmable wikis. `http://purl.net/net/cpw`.

[4] W. Cunningham and B. Leuf. *The Wiki way*. Addison-Wesley, New York, 2001.

[5] E. Da Lio, L. Fraboni, and T. Leo. Twiki-based facilitation in a newly formed academic community of practice. In *Proceedings of WikiSym 2005*.

[6] A. Di Iorio, V. Presutti, and F. Vitali. WikiFactory: an ontology-based application to deploy domain-oriented wikis. In *Proceedings of the European Semantic Web Conference*, 2006.

[7] Dokuwiki. `http://www.splitbrain.org/projects/dokuwiki`.

[8] A. Haake, S. Lukosh, and T. Schummer. Wiki templates: Adding structure support to wikis on demand. In *Proceedings of WikiSym 2005*.

[9] P. Herman. Moin Moin Wiki. `http://twistedmatrix.com/users/jh.twistd/moin/moin.cgi/`.

[10] J. Hess. Ikiwiki. `http://ikiwiki.kitenet.net/`.

[11] M. L. Jugel and S. J. Schmidt. Snipsnap: the easy weblog and wiki software. `http://www.snipsnap.org/space/SnipGraph`.

[12] E. E. Kim. Purplewiki. `http://purplewiki.blueoxen.net/cgi-bin/wiki.pl`.

[13] W. E. Mackay. Patterns of sharing customizable software. In *CSCW '90: Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 209–221, New York, NY, USA, 1990. ACM Press.

[14] A. MacLean, K. Carter, L. Lövstrand, and T. Moran. User-tailorable systems: pressing the issues with buttons. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182, New York, NY, USA, 1990. ACM Press.

[15] Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation 21 February 2001, `http://www.w3.org/TR/MathML2`, 2003.

[16] Openwiki. `http://www.openwiki.com/`.

[17] S. B. Palmer. Rdfwiki. `http://infomesh.net/2001/rdfwiki/`.

[18] P. Thoeny. TWiki: Enterprise Collaboration Platform. `http://twiki.org`.

[19] J. Urban. XML-izing Mizar: making semantic processing and presentation of MML easy. In A. Asperti et al., editors, *Post-Proceedings of the 4th International Conference on Mathematical Knowledge Management, MKM 2005*, volume 3863 of *LNCS*, pages 346–360. Springer-Verlag, 2006.

[20] WikiGnomes. `http://en.wikipedia.org/wiki/Wikignomes`.

[21] WikipediA, The Free Encyclopedia. `http://www.wikipedia.org/`.

[22] Wiki RPC interface 2, API version 2. `http://www.jspwiki.org/Wiki.jsp?page=WikiRPCInterface2`.

[23] World66. World66 home. `http://www.world66.com/`.