Cooperative Repositories for Formal Proofs*

A Wiki-Based Solution

Pierre Corbineau and Cezary Kaliszyk

Institute for Computing and Information Science
Radboud University Nijmegen, Postbus 9010
6500GL Nijmegen, The Netherlands
{corbinea,cek}@cs.ru.nl

Abstract. We present a new framework for the online development of formalized mathematics. This framework allows wiki-style collaboration while providing users with a rendered and browsable version of their work. We describe a prototype based on Coq, its web interface as implemented by the second author, and a modified version of the MediaWiki code-base. We discuss open issues such as dependencies and repository consistency. We explain limitations of the current prototype and we give a perspective towards a more robust solution.

1 Introduction

1.1 Motivations

Proof assistants are software tools used for expressing properties and checking proofs of those properties, be it about mathematical concepts or models of computer systems or software. Nowadays, most proof assistants follow the *interactive paradigm*: the user enters the statement of a theorem; the system checks the well-formedness of the statement. The user then enters a proof commands and the systems responds by validating the command and giving the remaining facts to be proven. This process is then iterated until the proof is complete. Thus, the resulting sequence of commands, called *proof script*, has barely any meaning without the succession of *proof states* it yields. However, most formal developments only consist of the bare proof script, maybe with some comments.

Two solutions are available for people who want to understand the proofs better: HTML rendering and local execution. With web rendering, the proof scripts are processed by a documentation tool that turns the files into HTML documents and provides some facilities such as hyperlinks from symbols to their definition, indexes of symbols and searching. Some even provide pretty-printing of comments, rendering of mathematical formulae.

But to understand the proof script itself, one has to first locate and download the files containing the proofs, then install the proof assistant, and finally run

* This work was funded by NWO Bricks/Focus Project 642.000.501 (Advancing the Real use of Proof Assistants) and partially funded by NWO FEAR Project.

M. Kauers et al. (Eds.): MKM/Calculemus 2007, LNAI 4573, pp. 221–234, 2007.

the proof assistant on the file to inspect the sequence of proof states. When doing this, one loses the ability to browse the code using hyperlinks, and it can sometimes be complicated to get the proof assistant to run on one's computer in the first place.

Recent work by Kaliszyk[1] shows that the *Asynchronous DOM Modification* web technology (sometimes referred to as *AJAX* or *Asynchronous Javascript and XML* [2]) can be used to build a web interface for interactive proof assistants: ProofWeb. This means that users can use their favorite web browser to run proof assistants sessions, so they can perform themselves the checking of the formal proof. However this work still lacks essential features: it is not designed to support multi-file developments properly, no proper HTML rendering is implemented and there are no tools to store and retrieve multiple versions of files.

A popular web architecture supporting all those features is called wiki. The wiki concept actually covers many implementations, but all are aimed toward a *cooperative authoring* of knowledge repositories. The key feature of a wiki system is the ability to follow an 'edit' link and be able to immediately modify and publish a new version of the page being viewed.

The popularity of wiki based solutions made us think of integrating the web interface for proof assistants within a wiki repository: the web interface would be used as the viewing and editing window for files containing proof scripts. The main difference between our work and common wiki usage is that our framework handles formal content that requires a consistent environment (i.e. file dependencies) to run interactive sessions. Thus (semi-)automated maintaining of cross-file consistency is crucial.

1.2 Related Work

Most proof assistants already have a more or less user-friendly way of *rendering* formalisations as a set of interlinked web pages. Some provide a standalone tool that allows users to render their own files: this is the case for Isabelle[3] and Coq[4]. Isabelle also provides a way to navigate the dependency graph of multi-file developments.

The Mizar[5] system has a proof repository called the MML (Mizar Mathematical Library) [6]. This repository is modified by human editors: duplicates are eliminated, results are moved to appropriate sections, new sections are created. This gives the MML a monolithic and consistent look[7]: it is handled as an encyclopedia, where new content is added with many authors but one central editing comitee. However, the rendering tool is not available for the common user to work with his local development. The MML (and its associated journal JML) is the *de facto* standard way to publish a Mizar proof.

The Logiweb System[8] provides a way to submit and retrieve articles from a network of distributed repositories. It allows reliable cross-references to fixed versions of already published articles. However it still relies on a locally installed checker to verify articles before submitting them.

The HELM[9] (Hypertextual Electronic Library of Mathematics) and the Whelp[10] search engine give users a good rendering of distributed formal libraries along with a powerful search engine. The Matita[11] proof assistant offers native support for queries and browsing of these libraries.

The Logosphere[12] project aims at presenting developments from different proof assistants (Nuprl, PVS, ...) using a unified framework.

The Mizar and Coq proof assistants already have wiki web sites for their documentation. The Mizar wiki[13] is an official, general purpose web site whereas the Coq wiki, called Cocorico[14], is a community website more dedicated to the sharing of specific knowledge about Coq usage, hints and tips, dirty tricks ...

1.3 The Future of Proof Interfaces

The aim of this new web-based cooperative proof environment is to provide — as an IT marketing representatives would say — a *complete solution* for the development of formalized mathematics or software verification. It brings together the availability of a web-interface with the accessibility of a web-rendered archive.

The unique feature of this environment is that, beyond the separation between the raw editable and rendered read-only versions of the files (a characteristic of wiki environments), both of those versions can be processed by the proof assistant at the request of the user, giving him more information as to how the proof script works. Where standard online formal libraries tend to treat proof scripts as minor, here their contents can give the user insights on how the proof was made: the proof script is not write-only anymore.

Therefore, this environment provides a useful tool for specialists to communicate about proofs with a broader audience: non-specialists, general audience. It provides a simple way for article writers to give referees easy access to their formal development.

The repository is also a convenient way for proof authors to work from everywhere simply using a network access, and to learn from others' proof idiosyncrasies.

The repository can also be used for education about proof assistants and formal logics. A permissions system can allow students to cooperate on multi-files projects and their supervisor to provide guidance.

1.4 Document Contents

In the rest of the paper we present the technologies relevant for creating a wiki for proof assistants (Section 2), and the components that are used. Then we present the global architecture of our system and discuss our library consistency policy (Section 3). We describe our current prototype (Section 4) and discuss performance and security issues. Finally, we give our road map towards a more stable system and present our conclusion (Section 5).

2 Web Technologies

2.1 Asynchronous DOM Modification

With the growing usage of the Internet, more technologies are available for designers of web services. Recently asynchronous DOM modification technology has allowed the creation of interfaces that are completely available in a web browser, but have similar functionality and responsiveness to local ones.

The *asynchronous DOM modification* technology (sometimes referred to as *AJAX* [2] or *Web Application*) is a combination of three commonly available web technologies. JavaScript is a scripting programming language interpreted by the web browsers. *Document Object Model* (*DOM*) is a way of referring to sub elements of a web page, allowing to modify it on the fly to create dynamic elements. XmlHttp is an API available to client side scripts, that allows sending requests to the web server without reloading the page.

The asynchronous DOM modification technique consists in creating web pages that capture events and processes without reloading the page. Events that can be processed locally modify the web page in place. For actions that require interaction with the server, the data is sent using an asynchronous XmlHttp transaction and the page is modified by the script when receiving the answer, therefore making the interface as responsive as an application run locally. For a more detailed description see [1].

One application of this technology is an architecture for the creation of web interfaces for proof assistants, that are completely available in a web browser, but resemble and behave as local ones [1]. ProofWeb, an implementation of this architecture, keeps a prover session for every user on the server (Fig. 1). It allows similar interaction as ProofGeneral [15] does, but using a web browser.

2.2 Wikis

Wikis are dynamic web sites that behave as static ones: they contain a number of fixed pages that can link to each other. The unique feature of wikis is that each of those pages includes an 'edit' link that displays the contents of the page in an editable textbox (or in a more fancy WYSIWYG box in advanced implementations). This page allows the user to actually modify the contents of the box and publish the new version on the web site, simply using a web browser.

This is what makes wikis dynamic: this online edition feature requires files to be served in a more fancy way than just static HTML files. Usually they are stored in a database system rather than in a filesystem. Unfortunately, current proof assistants do not include the functionality to access such databases.

The wiki technology is now very popular, especially for documentation of software tools: it allows to start with a very small, general (and somewhat imprecise) documentation which is then improved by visitors when finding inconsistencies, errors and missing items. The most famous wiki is obviously the Wikipedia web site, which aims at being an online encyclopedia where information is added by visitors. In each of the 14 most popular languages on Wikipedia, more than

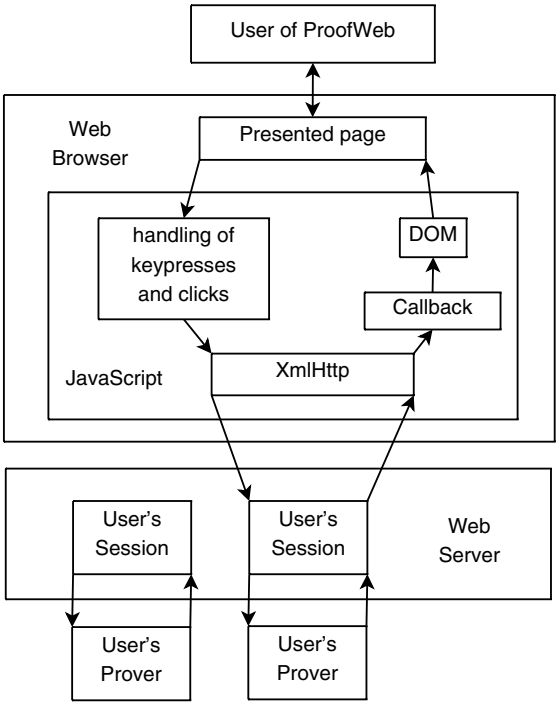

Fig. 1. ProofWeb architecture

100,000 articles are available. This shows that the wiki architecture can support large amounts of data and heavy activity.

The file format used by wikis is usually a simplified markup and formatting language, tailored to make references to other pages simple to add. Wikis usually have a permissions system to forbid reading or writing for particular users. Most of them also allow modification by unregistered users. In that case the IP address of the client is used as an identifier.

Wikis also offer the possibility to explore the history of any article: what was modified, when. They allow renaming of pages, and provide indexes of available pages. They usually include tools that allow searching the page database.

Those features match our requirements for a content management system to be usable with the ProofWeb framework.

3 Architecture

3.1 Main Components

The system we propose uses proof assistants with some of their companion utilities and some web serving utilities. The first element we need is the interactive toplevel of a prover. It is required on the server side to be able to verify the

input of the user in an incremental manner and to go to particular positions in them.

For efficiency reasons some provers allow compilation of their input files. Such files can then be quickly reused, without verifying all proofs contained in them again. For such provers we want to use the compiler on the server to generate a compiled version of proofs that are saved.

Many provers include documentation generators, that process raw prover input files and generate rendered output. The output of a documentation generator is usually HTML or PDF format. Links between files are created, different conceptual elements of the prover input are colored in different color, and sometimes mathematical formulas are rendered in a graphical way.

We need to keep a history of versions for every prover file. Usually, collaborative developments are done using version control systems. The source files are kept in the repository and each user has to build compiled or rendered versions him/herself.

Not only would we like the source prover files to be stored for all versions, but also the rendered and the compiled files (for provers that include this concept). This way users can see a rendered version of older versions of files. Referring to older versions of compiled files will be discussed in section 3.3.

Wikis already include some kind of versioning of the files they contain. Generally file versions are numbered in sequence. The user that made every change is stored with the file, and viewing changes is possible. The history mechanism is more limited than the ones provided by file versioning systems, but the simplicity can also be an advantage: in particular wikis do not include branches, tags, etc and the casual user does may not have a good understanding of it.

A wiki infrastructure will be used for tracking changes done by users and allowing them to see the history of files and changes. It needs to store files in a way that is accessible by the prover toplevel. The wiki should allow generating indexes and searching for terms. Most wikis generate text indexes and allow searching for text only, whereas prover scripts are highly contextual.

Finally we need a web part that allows interactive edition of a proof script in a way that resembles local work, to allows efficient work. Additionally we would like to be able to step interactively over the proof regardless of whether we are in view or edit mode. The ProofWeb framework can be modified to allow those two modes.

3.2 Global Design

Our architecture is composed of a web server running a modified version of a wiki that redirects some requests to a ProofWeb server (Fig. 2).

Editors of most wikis are standard HTML textboxes, and the flat text includes special markup for marking links and elements that should be formatted in a special way in the read-only version. Recent wikis allow WYSIWYG editing in an editable IFrame. The HTML formatting introduced by user's browsers is combined with wiki links to create the read-only version. In our architecture we embed the ProofWeb editor as the editor of the wiki. This way, the user can

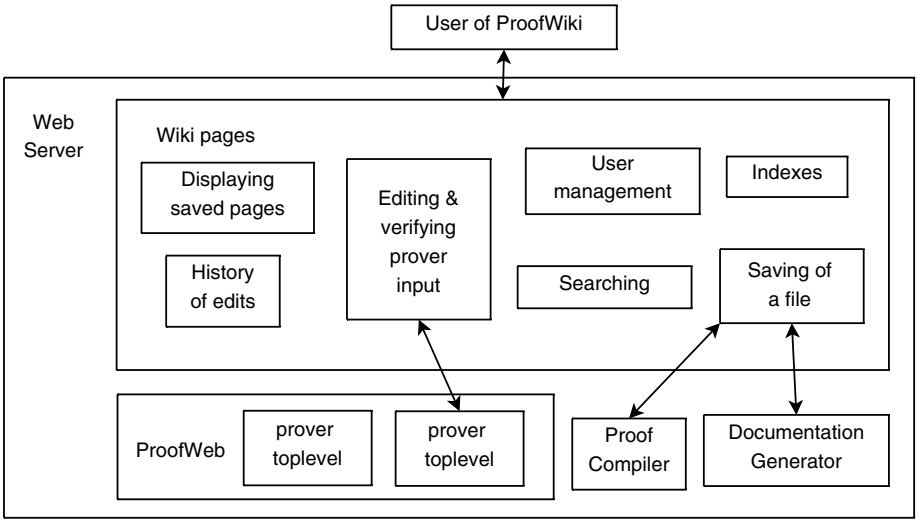

Fig. 2. Our architecture

edit the script in an interactive way seeing the output of the prover. Additionally we include a readonly version of the ProofWeb interface for the read-only version to allow examining the prover state at any point in the buffer.

The next necessary change is the way the wiki stores users files. For every saved file the wiki tries to compile it and to make a rendered version of it. The rendered version should be linked with the original, and is therefore stored in the same way the wiki stores the original in its database. Whenever the user requests the file for viewing or editing one of those two versions is used. The compiled version is stored as a standard file in the filesystem in order to make it available to the compiler and to the toplevel used in prover sessions on files that refer to this compiled file.

This change in the wiki behavior should not prevent users from storing and editing standard wiki pages in the repository. Those would include textual description of the formal content, discussions, tutorials with hyper-links to formal content.

The documentation generators of provers have to be able to generate a wiki compatible output. The format that a wiki displays is usually very close to HTML which many prover documentation generators already support. The important difference with respect to HTML is that since we will process the rendered version of the script we need to be able to distinguish active parts of the file from comments.

3.3 Consistency Issues

In usual wikis links to nonexistent pages lead to a new editable page. This is perfectly acceptable for the usual informal content but not for a formal development referring to another: think of it as a program missing libraries.

Moreover, we need to make sure that dependencies are always consistent. Files in the database can depend on each other, sometimes in an indirect (transitive) way. First of all, we want to require all saved files to be valid (compilable); they can still contain incomplete proofs terminated with the Coq `Admitted` keyword or its equivalents for other provers. For a valid saved file we want to ensure that the current version remains valid after changes to the files it relies on. Some provers already include compatibility verification mechanisms. Coq stores the checksums of files to ensure binary compatibility between compiled proofs. To solve the problem, we have to consider the static and the dynamic approach.

The dynamic approach is the convention that a file always refers to the latest versions of other files. It means that saving any change to a file will induce a costly recompilation of all files it depends on. Another problem is that changing definitions deep inside a library will make many developments incompatible and thus correct files will stop working. Saving only valid files does not solve this problem since the objects they contain (their interface) might be modified too. This approach also makes older versions of existing files immediately obsolete.

The opposite approach is static linking, where a saved file always refers to the same version of other files. In other words, we never change a file, but rather add a new version of that file, with a fresh name. This means that the user will have to manually update the version number of files that are referred to if newer versions of those become available. The main advantage of this approach is that of integrity: provided you can safely assign new version numbers, you can enable concurrent access. Moreover, changing a file will never *break* another file. However, when changing a file deep in the library, one has to manually modify all the files in the dependency chain between that file and the files in which the changes should be reflected, which can sometimes be quite heavy.

3.4 Towards a Hybrid Approach

We believe that the static approach is a more adequate way to store older (historical) versions of a given file, whereas up-to-date files should use the dynamic approach towards dependency. This way, older versions of files still make sense by statically referring to older versions of files they depend on. The latest versions can remain up-to-date with their immediate dependencies by being dynamically linked to them, i.e. recompiled when new versions of those files are saved. It might happen that such a file might not be valid anymore because of changes made to its dependencies: to keep validity we have to make it link statically to the suitable previous version.

To help with this version compatibility issue, we propose a three-colour scheme:

- A file is labelled as *red* (i.e outdated) if it depends statically on an older-than-latest version of another file.
- A file is labelled as *yellow* (i.e tainted) if it depends only on the latest versions of other files, and one or more of those files have a yellow or red status. Yellow status thus tracks the files which are indirectly lagging behind.

– A file is labelled as *green* (i.e. up-to-date) if it depends only on the latest versions of other files, and all those files are also labelled as up-to-date (green status).

The separation between the yellow and red files comes from the fact that red files have to be manually updated to become green again (i.e. by creating a new version of them), whereas yellow files might be fixed by updating the red files that taint them.

The switching to red status can be automated by rewriting `Require` statements on-the-fly to make them refer statically to the last suitable version of the file depended on. This means that fixing a red file can give red status to yellow files that it was tainting, thus pushing the problem upwards in the dependency tree.

If the user wants to export a file together with its dependencies from the repository, a mechanism can be used to convert long file names (with version number) to short ones. The case might arise where a file would refer, directly or by transitivity, to an old version of itself. We can either forbid this or generate fresh file names using standard suffixing techniques.

The procedure of updating the prover itself, although intended to be rare, will be critical. Here a decision will have to be made whether to port all possible versions or only the newest versions of each files and their dependencies. The current system is clearly not yet designed to enable such updates without putting it offline and porting files manually, but such a feature should definitely be designed and implemented.

4 Prototype

4.1 Implementation

To experiment with our idea, we have created a prototype implementation based on off-the-shelf components as much as possible. We chose Coq as our target proof assistant. We used the MediaWiki code-base, the `coqdoc` documentation generator for Coq and ProofWeb for Coq (Fig. 3). The `coqdoc` tool was modified to generate wiki format rendered pages.

When the user opens a page of our wiki, he/she is presented with a viewing page where the usual contents area is replaced by three subframes. One frame shows the rendered version of the current document, the second one shows the current proof state and the third one displays the Coq error messages.

The user may press the 'up' and 'down' buttons to step over the proof and examine both the proof state and Coq messages. A background coloring scheme allows the user to keep track of the part of the script that was already processed.

The proof is rendered, that is identifiers are colored and linked to their definition, mathematical LaTeX comments are rendered, links to internal wiki pages lead to those wiki pages and links to Coq standard library objects lead to the documentation on the Coq website (Fig. 4).

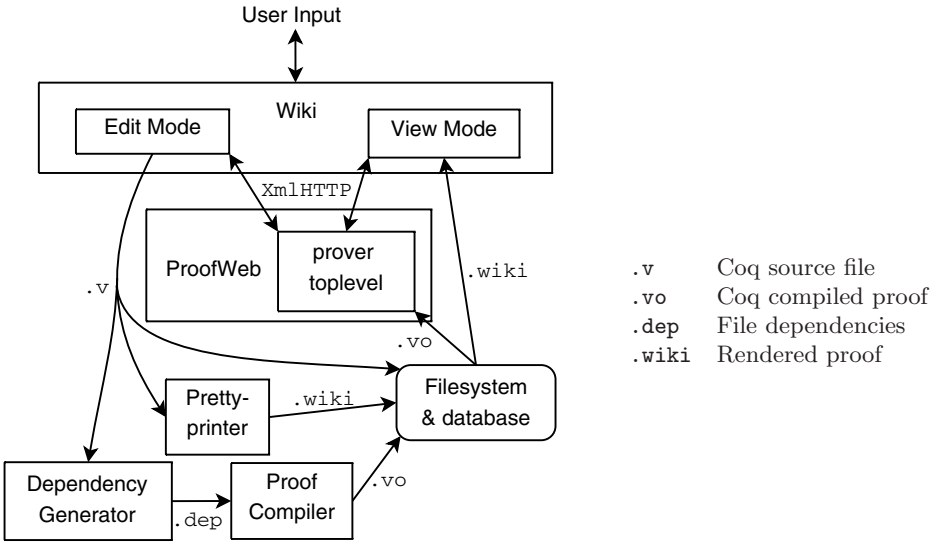

Fig. 3. Data flow diagram for our prototype wiki

The page also includes standard wiki elements, one of which is the 'edit' button. When the user starts editing the page, a similar page is presented, but with the raw proof script (no rendering) in a modifiable text box. The user may modify the script and use the 'up' and 'down' buttons to step over the proof in a similar way as in the view mode (Fig. 5). The processed part of the buffer is frozen.

When satisfied with his work, the user can save the proof. The contents of the buffer are processed in three ways:

- The raw script is saved in the database, to be used by following edits.
- The file is compiled and the corresponding `.vo` file is stored in the filesystem for processing of files that would include it using `Require` statements.
- A rendered version is generated by `coqdoc` and saved in the database to be displayed in view mode.

The user can see the history of any page as well as display the differences between the sources of any versions, using built-in MediaWiki routines. The textual search mechanism allows to query the source Coq files for any terms.

4.2 Security and Efficiency

The security and efficiency of the server are crucial since unavailability of the proof wiki would make the users not only unable to work, but also unable to access their own files. The security and efficiency of the architecture relies on the security of ProofWeb, the underlying wiki, the compilation and rendering processes and the communication mechanism.

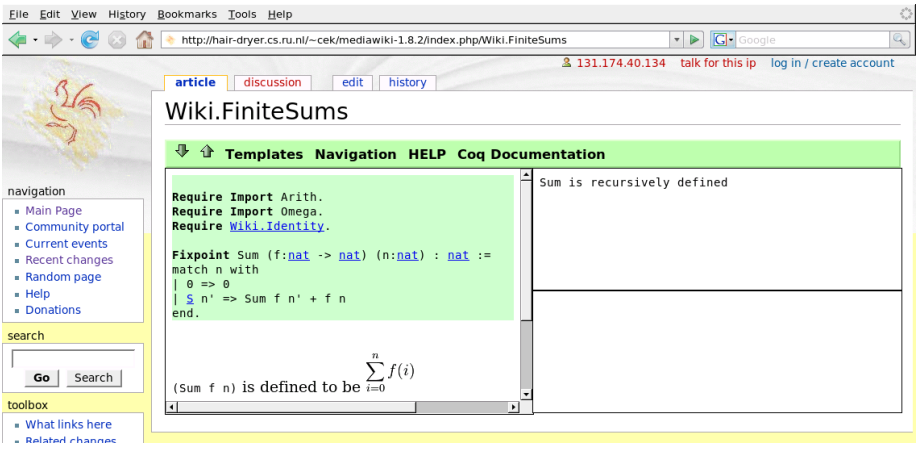

Fig. 4. Screenshot of the prototype showing the rendered version of a a Coq file. The verified part of the edit buffer is colored. The state buffer shows the state of the prover, there are no Coq warnings.

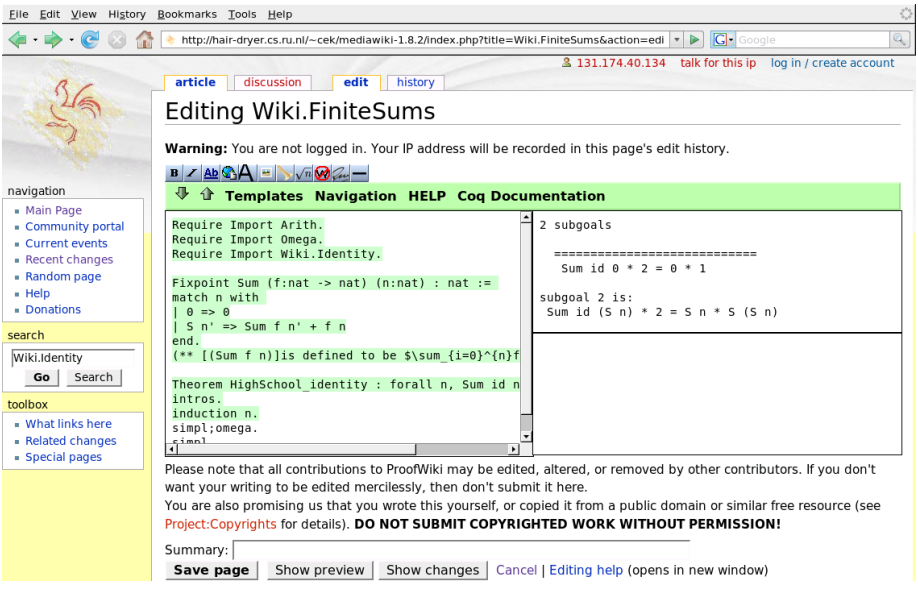

Fig. 5. Screenshot of the prototype showing the editing of the corresponding source file. The verified part of the edit buffer is colored and frozen.

The security and efficiency of ProofWeb are described in detail in [1], we remind here the most important issues. The solution adopted is *sandboxing*: the ProofWeb server process is run in a chrooted environment as a non privileged user without network access. The permissions include only reading server files

and executing prover toplevels. Provers are run as different users with a modified scheduling policy and have rights to read only the prover libraries and to write in designated subdirectories. For provers that are based on programming language toplevels issuing toplevel commands can be disabled. Finally to disallow storing overly large amounts of data filesystem quota is used.

The sandboxing which is a part of the ProofWeb architecture makes it reasonably safe. The efficiency is divided by the number of users, but it is straightforward to distribute prover sessions over a set of machines. We are additionally running a Coq dependency generator, compiler and renderer. We run these processes in the same sandbox as the prover toplevel, so we expect them to be comparably secure. However for big formalizations performing the compilation can be costly. Specially when many files depend on each other, modification of one of them may require recompiling numerous proofs. We expect this to be the main bottleneck of a wiki for proofs. Although this can also happen with local proof interfaces, here multiple parallel sessions might overload the system for a longer period.

The proof text verification that ProofWeb does is independent from page serving performed by the web server and MediaWiki, so we can analyse the latter separately. Wikis are quite secure and efficient. At the time we are writing this article, Wikipedia provides servers that have more than 3 million users and 1.6 million articles without significant efficiency issues. An issue that is often a problem in wikis is vandalism. Disallowing edition by particular users or IP addresses is a common practice, and is already supported in MediaWiki. Discovering vandalism in our framework may sometimes be easier than in standard wikis, since incorrect proofs no longer compile.

The data that is being transferred to and from the wiki is usually public, still the communication mechanism can be secured by configuring the web server that serves the wiki to use HTTPS.

If the wiki is secured properly we do not expect crackers to be an important issue. However the efficiency seems to be quite fragile, in particular it seems that our architecture is quite vulnerable to denial-of-service attacks.

4.3 How to Integrate Other Provers

Although our prototype has been implemented for Coq, we do not rely on any specific Coq feature. We think that extending the wiki to other provers is feasible provided the following functionalities are available: wiki compatible documentation renderer, dependency generator, ProofWeb support and optionally an index generator.

The renderer does not need to be sophisticated, the only mandatory feature of the renderer is distinction between active proof script from comments. Other features like syntax highlighting and links are not necessary, although they allow a more wiki-like interaction.

The wiki needs to know how to call the dependency generator of the prover, to know what files need to be updated if a particular file is modified. If the prover has a compiler, the wiki needs to know how to compile proofs. The wiki

also needs to be able to identify statements that refer to other files during the interactive session.

An optional element is an index generating utility. It is needed for the wiki to distinguish concepts from the new prover's language. This allows not only nice index pages in the wiki, but also searching for particular prover objects, like only definitions or theorems.

Finally ProofWeb needs to be able to interact with the prover. It already supports some provers. To extend it to a new one, the client part needs to know how to find the ends of complete prover commands and the server part needs to know how to interface with the prover process, in particular it needs to know how to check if commands succeed and how to undo. The details of extending ProofWeb to a new prover are described in [1].

5 Conclusion and Future Work

5.1 Future Work

The current architecture of the prototype is not satisfying since it relies on a double storage of files: in the database, and on the disk. We are also limited by the way MediaWiki handles its name space. If we adopt the static system where files are never modified, it can be worthwhile to consider moving all the data to the file system, and adopting an architecture where we can have a better control of the name space.

The static naming will require to implement a versioning system for the substitution of `Require` statements and the distributed generation of version numbers, then the three colour scheme will be added. A mechanism for importing and exporting parts of the library will also be necessary, to allow users to have a local copy on which to work without Internet access.

A milestone in this development will be the ability to actually import the Coq standard library and official users contributions to our repository. Only then will we be able to get user feedback and report on the suitability of the repository for Coq users.

The `coqdoc` tool is able to generate index files that contain all constants occurring in the library. We could use such a a feature to generate such wiki pages.

The basic textual search is very limited and proof assistants users often need query types that are far beyond the scope of textual search: find theorems about a given object or do pattern matching on theorem statements. This might be achieved by adapting the Whelp search engine to search our database: it will require a customisation of the indexing technology.

We could also experiment with more advanced rendering tools such as Helm and consider using MathML instead of (currently) HTML with LaTeXimages.

5.2 Conclusion

Although our prototype is still at a very early stage of development, our idea of combining a wiki web site with the ProofWeb interface looks definitely

promising. Surprisingly, we could achieve the current result without many modifications neither to the wiki code-base, nor to ProofWeb. Most of the work was devoted to database modification and rendering.

We believe that formalising mathematics in a wiki system will foster more cooperation both within prover specific communities and between users of different provers, especially if we can make several provers coexist in the same repository. We also believe that such a project can act as a display of the work on formal proofs for a wider audience.

References

1. Kaliszyk, C.: Web interfaces for proof assistants. In: Autexier, S., Benzmüller, C. (eds.) Proceedings of the FLoCs Workshop on User Interfaces for Theorem Provers (UITP-06), Seattle, pp. 53–64 (To be published in ENTCS) (2006)
2. Paulson, L.D.: Building rich web applications with ajax. Computer 38(10), 14–17 (2005)
3. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
4. Coq Development Team: The Coq Proof Assistant Reference Manual Version 8.0. INRIA-Rocquencourt (January 2005) URL :`http://coq.inria.fr/doc-eng.html`
5. Muzalewski, M.: An Outline of PC Mizar. Fondation Philippe le Hodey, Brussels (1993)
6. Bancerek, G., Rudnicki, P.: Information retrieval in MML. In: Asperti, A., Buchberger, B., Davenport, J.H. (eds.) MKM 2003. LNCS, vol. 2594, pp. 119–132. Springer, Heidelberg (2003)
7. Rudnicki, P., Trybulec, A.: On the integrity of a repository of formalized mathematics. In: Asperti, A., Buchberger, B., Davenport, J.H. (eds.) MKM 2003. LNCS, vol. 2594, pp. 162–174. Springer, Heidelberg (2003)
8. Grue, K.: Logiweb - a system for web publication of mathematics. In: Iglesias, A., Takayama, N. (eds.) ICMS 2006. LNCS, vol. 4151, pp. 343–353. Springer, Heidelberg (2006)
9. Asperti, A., Padovani, L., Coen, C.S., Schena, I.: Helm and the semantic math-web. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 59–74. Springer, Heidelberg (2001) URL: `http://helm.cs.unibo.it/smweb.ps.gz`
10. Asperti, A., Guidi, F., Coen, C.S., Tassi, E., Zacchiroli, S.: A content based mathematical search engine: Whelp. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 17–32. Springer, Heidelberg (2006), URL: `http://www.bononia.it/~zack/stuff/whelp.pdf`
11. Asperti, A., Coen, C.S., E.T., Zacchiroli, S.: User interaction with the Matita proof assistant. Journal of Automated Reasoning (To appear 2007)
12. Schurmann, C., Pfenning, F., Kohlhase, M., Shankar, N., Owre, S.: Logosphere. A Formal Digital Library Logosphere homepage: `http://www.logosphere.org`
13. Mizar Development Team: Mizar wiki (2006) URL: `http://wiki.mizar.org`
14. Niqui, M.: Cocorico: a Coq wiki (2005) URL: `http://cocorico.cs.ru.nl/coqwiki`
15. Aspinall, D.: Proof General: A generic tool for proof development. In: Schwartzbach, M.I., Graf, S. (eds.) ETAPS 2000 and TACAS 2000. LNCS, vol. 1785, pp. 38–42. Springer, Heidelberg (2000)