Proviola: a Tool for Proof Re-animation

Carst Tankink[1], Herman Geuvers[1,2], James McKinna[1], Freek Wiedijk[1]
{carst,herman,james,freek}@cs.ru.nl

[1] Radboud University Nijmegen
Institute for Computing and Information Sciences
Heyendaalseweg 135, 6525 AJ Nijmegen
[2] Technical University Eindhoven
The Netherlands

Abstract To improve on existing models of interaction with a proof assistant (PA), in particular for storage and replay of proofs, we introduce three related concepts, those of: a **proof movie**, consisting of frames which record both user input and the corresponding PA response; a **camera**, which films a user's interactive session with a PA as a movie; and a **proviola**, which replays a movie frame-by-frame to a third party. In this paper we describe the movie data structure and we discuss a prototype implementation of the camera and proviola based on the ProofWeb system [7]. ProofWeb uncouples the interaction with a PA via a web-interface (the client) from the actual PA that resides on the server. Our camera films a movie by "listening" to the ProofWeb communication.
The first reason for developing movies is to uncouple the reviewing of a formal proof from the PA used to develop it: the movie concept enables users to discuss small code fragments without the need to install the PA or to load a whole library into it.
Other advantages include the possibility to develop a separate **commentary track** to discuss or explain the PA interaction. We assert that a combined camera+proviola provides a generic layer between a client (user) and a server (PA). Finally we claim that movies are the right type of **data** to be stored in an encyclopedia of formalized mathematics, based on our experience in filming the Coq standard library.

1 Introduction

Interaction with modern theorem-proving tools, Proof Assistants (PAs), still imposes heavy (temporal, spatial, computational, cognitive) resource demands on users. It is hard to write a proof, but typically even harder to write a formalized version with a PA. This additional overhead arises from at least the following:

– It is necessary to become (at least somewhat) familiar with the technical details of the PA, since the user needs to install and configure it before use;
– The user needs to understand the tools the PA gives her, in terms of libraries and commands, and how to use them to achieve a formalization of the proof;
– The user needs to know the proof and its implicit assumptions in far greater detail than required to communicate the main ideas to another person.

Much of the effort in interaction design for PAs has focused on the second and third of these issues from the point of view of defining a language of suitable basic proof steps, augmented with automation layers which are either fully programmable, or else encapsulate well-defined larger-scale proof steps, together with an editing model of how to soundly maintain a partially completed proof.

That is to say, the basic PA use case of "writing a proof" has received most attention, while those of "reading a proof" (written by someone else) or "browsing a library" rather less so: the narrative or explanatory possibilities afforded by a formal proof text have been largely overlooked in favour of (variously prettily rendered) static digests of named definitions and theorem statements. These are necessary prerequisites for these use cases, but hardly sufficient for gaining insight into how such proofs "work" (or even: how partial proof attempts may *fail*, as when trying to discuss such examples on a mailing list).

In each use case, however, there still remain the computational and cognitive bottlenecks arising from the first and second problems. For example, as illustrated by Kaliszyk [8], before being able to formalize a theorem in Isabelle, a user needs to install the system, comprising the program itself, an HOL heap and a version of PolyML. This does not include a user interface, provided by the Emacs-based ProofGeneral. Having installed the PA, the user is then left with understanding it: digging through a tutorial or manual, and finding out what contributions and libraries are necessary for the formalization of the proof.

Our work contributes to addressing the first and second issues, reducing the overheads of installing and configuring a PA and simplifying access to, and communication about, existing proofs. The third issue we do not address here.

Previous work at Nijmegen has partially addressed the first issue. By providing a generic web-interface for PAs, the ProofWeb system [8] removes the computational load on users by uncoupling interaction with a PA via a dedicated webserver. This relieves the user from the installation problem: she can just visit a website and use a web-interface to access the PA. However, when trying to understand a (part of a) formal proof script, it is often necessary to see how the proof state changes through execution of a specific tactic. This requires first to bring the PA into that specific state — finding and loading the required libraries and files — a significant overhead, not addressed by ProofWeb.

Similarly, when explaining a tactic or a part of a proof script, with the current technology, a proof author has to publish the proof script, sometimes with an explanation of the output the PA returns to her. This is not very satisfactory and often not informative enough, especially when the publication of the script is intended to show the intricacies of the proof: if the reviewer is uninitiated in the specifics of the PA, she might not understand the proof script.

This paper discusses the design of a further uncoupling layer, providing a client-side Proxy to the server-side PA output. What we do is send a script piecewise to the PA and record the response: that is, we "film" the interaction with the PA. So by analogy with film-editing, we have dubbed our system a "proviola": a playback and editing suite for "proof movies" created by submitting formal proof texts, "scripts", to the PA.

In the present paper we describe the basic ideas behind the movie-camera-proviola concept by discussing two use cases and a prototype implementation based on ProofWeb, which can be inspected at `http://mws.cs.ru.nl/proviola`.

Then we further examine the versatility of the concept and observe that we can further use our movies as a drop-in replacement for the existing ProofWeb interaction model for editing proofs: the proof movie then becomes the "file in the middle" that receives interactive updates from the user and the PA.

1.1 Contributions

In this paper, we describe the design of an architecture for capturing the interaction between a PA and its users. Specifically, we:

– articulate two roles and associated use cases: *creating* and *reading* a proof;
– define a **proof movie** datastructure that encapsulates interaction between a proof author and a PA; such wrapping affords a reviewer fast access to details of this interaction;
– have built tools for creating and viewing movies: **camera** and **proviola**;
– identify the set of actions an author needs to create and modify a movie on-the-fly, and the gestures that give access to these actions;
– extend the model to allow arbitrary tools to operate on movie content; and
– design a concrete architecture for implementing such a system.

2 Background and Use Cases

We identify two roles involved in communicating a proof script: the **proof author** and a **proof reviewer** (the reader). The proof author is a user of a PA that creates a (formal) proof. The interaction with the PA is encoded in a **proof script**, which contains the commands used to build the proof. The author can communicate the proof to any reviewer by publishing this script, and the reviewer can look at the script by loading it in his local version of the PA.

These two roles both have their own well-defined activities, but an actor (an actual user of a system, instead of just an abstract role) can play both the author-role and the reviewer-role: when writing a proof, an author might want to review what he has done before. To make the activities of both roles explicit, we identify a single use case for each of them. These use cases are *creating a proof* for the author (Figure 1), and *reading a proof* for the reviewer (Figure 2).

A third use case is *browsing a library*, in which a user takes on a role similar to that of the reviewer, but also searches for useful lemmas in the library, and tries to understand how they are used. While we do not treat this use case here (we are grateful to one of our referees for drawing attention to it), we nevertheless claim that such "advanced" elaborations of the reviewer's use case would benefit from the proviola technology presented here. Current technology does not give users fast access to other developments, which is a prerequisite for this use case.

Legend The diagrams below are almost, but not quite, standard UML:

- A stick figure represent a user role (proof author and proof reviewer).
- A "package" represents a tool/program instance (here: the PAs).
- The cloud represents the Internet.
- A folded page is a file (here: the proof script).
- Arrows represent data flow; a double arrow, interaction between two parts.

First Use Case: Creating a proof To create a proof, an author writes commands to be interpreted by the PA. In response, the state of a proof changes by decomposing the theorem to be proved, generating new proof obligations or discharging goals as proven. A proof script stores a a transcript of the commands issued.

In Figure 1, we display the traditional implementation of this use case, in which the PA is locally installed, and creates a local copy of the proof script.
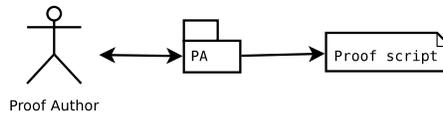
Figure 1. Creating a proof script

Second Use Case: Reading a proof To read a proof, the reviewer obtains a (copy of a) proof script, possibly via the Internet. Subsequently, he can load it in his copy of the PA, and "replay" the proof: many PA interfaces have a notion of stepping through a script, by sending commands one-by-one to the PA or by undoing the last command sent. Because the PA does not know that the commands it receives are extracted from a script, it responds to commands in the same way as in the creation use case: by sending the new proof state.
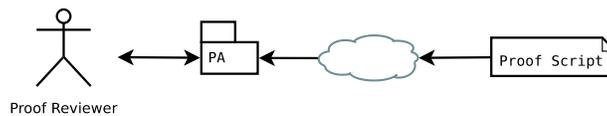
Figure 2. Reading a proof script

Discussion This form of communication has a number of problems:

1. If only a small part of the script is relevant, it might still be necessary to send the entire script to the reviewer: the parts of interest might require definitions defined previously in a script, or might use lemmas proved earlier.
2. Before a proof can be reviewed, the reviewer needs to install (ideally the same version of) the PA the author used, or be so familiar with its technology to interpret the script mentally[3]. Especially when the script is used to communicate a proof to a reviewer who is not part of the PA community, this can be a large handicap.

A possible solution to the first problem, frequently exercised on the Coq-club mailing list [2], is to simplify a proof script to a minimal example, which focuses on the problem in the script, and the definitions directly necessary to obtain this problem. But such simplification might be too drastic, abstracting away crucial details. Furthermore, abstraction is not an option when the main purpose of the communication is not to point out a problem, but to display a (partial) formalization of a proof to an outsider, because it is necessary to stay close to the vocabulary and methods of the target audience.

The ProofWeb system developed by Kaliszyk [7] places the PA on a central server that is accessible through an AJAX-based web application. This means that to review a proof, a reviewer needs only a web browser and the proof script, which could be hosted on the same server as the PA.

ProofWeb is an Internet-mediated realisation of the "creation" use case, which mitigates the second problem of having to install and configure a PA, but does not yet allow partial communication of the proof to a reviewer. It does not provide fast access to an arbitrary proof state: to obtain the state after a given command, all preceding ones need to be resent to the PA for reprocessing.

3   The Proof Movie

In the previous section, we identified two problems with the current method of distributing a proof script from a proof author to a proof reviewer. In summary, these problems are that reviewing requires the proof reviewer to resubmit the proof script to the PA, and that the proof script cannot be reviewed in fragments.

To solve these problems, we propose to enrich the proof script data structure. In the new data structure, which we call the **proof movie**, we record the commands sent to the PA coupled together with the response of the PA to each command. Such a pair of a command and a response is a **frame**. The exact Document Type Definition for the movie data structure can be found at `http://mws.cs.ru.nl/proviola/movies/film.dtd`.

The proof movie is designed to be *self-contained* and *generic*:

---

[3] For a PA that uses a procedural proof style (tactics), it is hopeless to try to interpret a proof script purely mentally, without seeing it executed.

Self-contained Making the movie self-contained means that a proof reviewer only needs a movie and a tool capable of displaying it to replay the proof: no other tools are necessary for this. Aside from this, the frames themselves contain the exact state of the proof at the point represented, meaning that it is possible for an author to publish a proof partially, omitting or reordering frames before publication.

Generic By making the movie generic, creation and display do not depend on a specific PA or PA version: this does require that one can specify a transformation from the PA's interaction model generating discrete frames.

```
<frame frameNumber="2">
  <command>
    intros A x.
  </command>
  <response>
    1 subgoal

    A : Type
    x : A
    ============================
     A
  </response>
</frame>
```

Figure 3. An example frame of a Coq movie

To implement the movie data structure, we used the eXtensible Markup Language (XML). An example frame in this implementation can be found in Figure 3. This example contains the notion of a "frame number": a sequence number identifying the order in which the commands were submitted to the PA.

We do not consider the movie to be a complete replacement for a script. Instead, it is a container of a part of the script, together with the output of the PA. This output does not need to be correct, but this does not interfere with our intention of the movie: we see a movie as an explanation of a proof, not as checked proof script *per se*. If a movie contains a complete script, the concatenation of all the command segments of all frames in the movie reproduces the script, which can then be (re-)checked by a PA.

The movie introduces a new use case, *creating a movie*, which we describe in Section 5. Having the movie also changes the use case of reviewing a proof script, and we describe this modified use case first, in Section 4.

4 The Proviola: Watching a Proof Movie

When the reviewer has obtained a proof movie, he wants to access the data within it to review the proof, much like when he obtained a proof script. In effect, the "reading a script" use case described in Section 2 and illustrated in Figure 2 has not changed, only the data structure supporting it has changed.

We call the system used for displaying a movie a **proviola**. Just as in film-making, where an editor might wish to review a film while editing, and moreover quickly fast-forward and rewind the movie to see individual shots, we wish to achieve similar access speed and portability. Indeed, our proviola is not a separate tool: rather, we realize it through the use of HTML and very simple JavaScript.
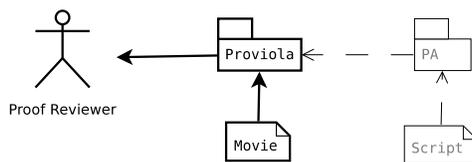
Figure 4. Watching a movie: proviola and movie proxy PA behaviour

Reading a proof script, revisited: watching the movie The movie is self-contained, and can be distributed like a script. Unlike a script, the contents of a movie can be inspected without any external tools except a web browser: the movie can be located anywhere, and inspected from this location. In particular, a PA is not required to compute the proof's state and the movie can be watched offline, at any time. After the reviewer loads a movie in the proviola, he wants to step through the proof much like when a PA was loaded with a script: by indicating for which command he wants to see the response.

As illustrated in Figure 4, the proviola and the movie together **proxy** [5, Chapter 5] the behaviour of a PA: the responses shown to the proof reviewer are stored (or cached) in the movie, after having been computed by the PA.

A prototype implementation of the proviola We implemented a prototype of the proviola as an XSLT transformation from the movie into an HTML file containing embedded JavaScript. This page initially shows the commands within the movie, much like a proof script. Figure 5 illustrates this: when the reviewer places his cursor over a command, the corresponding response is revealed dynamically. The command pointed to is highlighted as a visual reminder. The prototype proviola can be inspected at `http://mws.cs.ru.nl/proviola/movies/movies.html`.
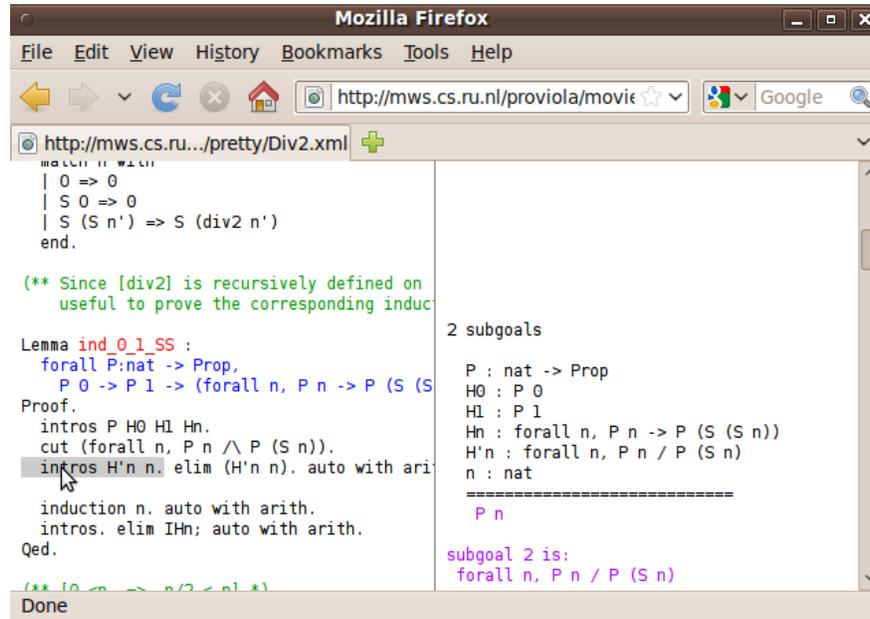
Figure 5. Screenshot of a proviola

5 The Camera: Creating a Proof Movie

Before a movie can be replayed, it must be created from a proof script by a tool we call a **camera**. Such *creation of a movie* is a new use case, shown in Figure 6.

Third Use Case: Creating a movie This is non-interactive: the user of the camera invokes it on a script, after which the tool does all the work, yielding a movie.

After it has been invoked, the camera parses the proof script given to it into separate commands. These commands are stored in a frame and sent to the PA. When the PA responds to a command, the response is recorded alongside the command. The frame is subsequently appended to the movie.

From the perspective of the PA, the camera and the script behave like an actual proof author. In other words: in creating the movie, the camera and script together *proxy* the behaviour of a proof author.

Because the author holds the original script, it seems natural that she invokes the camera on it to obtain a movie, for distribution to reviewers. However, a reviewer might also play the role of cameraman, given access to the script.

Prototype implementation We implemented the camera as a client to the ProofWeb system, available at `http://mws.cs.ru.nl/proviola/camera/camera.html`. Although it is possible for the camera to communicate directly with each PA, we believe that using a generic wrapper like ProofWeb has the
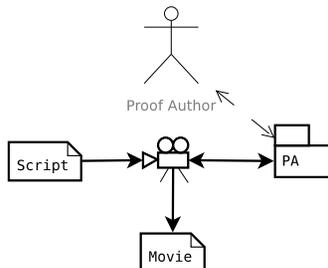
Figure 6. Creating a movie: camera and script proxy user behaviour

following advantage: ProofWeb provides a generic interface to different PAs: the communication protocol is the same for each PA, and the only PA-specific knowledge the camera needs to have is how to split a proof script into separate commands.

The main disadvantage to this approach is that to support new PAs with the camera, these need to be made compatible with ProofWeb, which imposes stricter requirements on the interaction model than the movie design needs. In our design, the camera behaves as a straightforward client to a ProofWeb server, wrapping the commands in the annotation expected by that server and unwrapping the responses. A step beyond this, which we investigate in Section 6, is creating a movie automatically and on-demand.

6 Proxying Movie-Making

Until now, we have created proof movies by submitting a complete script to a PA. In particular, we have filmed the Coq standard library [3]. We now wish to go beyond this simple case of filming completed scripts, and investigate how to create a movie dynamically, by observing the interaction between proof author and the PA. Based on these observations, we redesign this architecture to support the desired behaviour. This architecture will be implemented in future work.

As we have mentioned in Section 2, a user taking the role of a proof author can also take on that of proof reviewer. If she takes on these two roles simultaneously, we get the situation depicted in Figure 7. This figure is constructed by composing Figures 4 and 6, replacing the proxied components from each figure with their implemented counterpart in the other figure.

In this figure, the proof author writes commands into a script, which is submitted to the camera command-by-command. The command is then handled by the camera as described in Section 5. Because the proof author has the movie corresponding to the script loaded in the proviola, it updates whenever the script updates or when the PA responds to a command.

The camera as designed in the previous section requires an explicit action by the creator of the movie: the camera is a tool that needs to be invoked on a
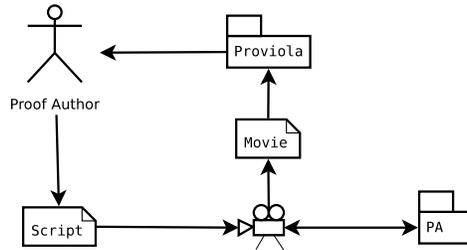
Figure 7. Interactively creating a movie: instantiating the proxies

proof script to create a movie from it. Such a design requires the proof author to constantly update the movie when updating the script, to keep them in sync. In fact, if we follow the information flow in Figure 7, the author cannot even see the result of her own changes to the script if she does not use the camera first.

As a solution for this disadvantage we merge the script and movie into a single data structure, that is manipulated by both the proof author (through the proviola) and the PA. To obtain this, we will need to modify the movie, the camera and the proviola in the following ways:

Movie The movie does not need to change in any radical way. The only change necessary is that a movie be editable after it has been created. This way, the proof author can write commands into the movie as she would do into the script.

Proviola The proviola already provides a display of the movie, giving the proof reviewer access to the script and the proof state at that point. To allow an author to update the movie, the proviola needs to be extended with an interface to update the commands in the frames in the movie. This extension is done by adding the notion of a **focused frame**, which can currently be edited. To manipulate this frame, we add gestures for the following actions to the proviola:

Create a movie This action creates a new, empty movie.
Focus frame The proof author can use this action to change the focused frame to be the frame she is interested in editing.
Edit frame The only frame that can be edited is the focused frame.
**Add frame to a movie** When the author finishes a command in the focused frame, a new frame is tacitly added to the movie and given focus. The previously focused frame is submitted to the camera for further processing.
Remove frame from a movie This is an inverse action to adding a frame.

We consider submitting a frame for further processing an implicit action: the proof author does not indicate in any way that she is finished with a frame, but the system recognizes a frame to be complete and processes it further, including rechecking commands later in the movie, if these depend on the frame that was changed. What "depends on" actually means depends on the script-management model (and more generally: the interaction model) of the PA used.

Camera To keep the proviola as lightweight as possible, the PA's manipulation of the movie is brokered through the camera. This includes periodically checking whether the author completed a command in the focused frame. Frames containing completed commands are then split off the focused frame and submitted to the PA. This means the camera evolves from batch-processing a script (as in Section 5) to continuously reading the movie, updating its contents as needed.

By merging script and movie, implementing the changes above, we obtain the architecture shown in Figure 8.
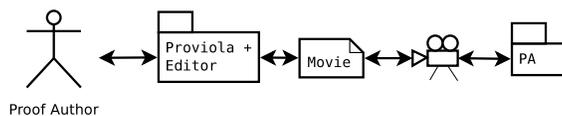
Figure 8. Interactively creating a movie: cutting out the script

Design and implementation of the system We have embarked upon a prototype implementation of the architecture described above. We decided to base it on ProofWeb's architecture, placing the camera on a server, with the proviola as a client to the camera, as an interface to the services offered at the server.

The movie is kept both as a local copy by the proviola and as a remote copy by the camera, and the protocol between camera and proviola is meant to keep these copies synchronized: each action of the author is executed on the local copy and then communicated to the camera, which communicates the change to the PA. As such, the movie becomes a *proxy* for the PA's state, and the interaction of proof author with PA is proxied by editing the movie.

By being placed between client and PA, the camera plays a similar role as the Broker in the PG Kit framework [1]. The main differences between that system and our proviola-based system, are based on the design decision to make the movie the main entity of the architecture. This has the following implications:

Cached history The history of the interaction with the PA is stored as a movie. Because the client has a local copy of the movie, it provides instant access to the history of the proof development.

Implicit proof navigation The author can freely choose the frames of the movie to edit instead of having to request the PA to unlock or process a line.

To synchronise the movie between camera and proviola, we follow a version of the *publisher/subscriber* [5, Chapter 4] design pattern:

– The proviola holds the local copy of the movie, and records when a user focuses on a specific frame.
– When a frame's content changes, for example, when a PA responds to a command, the camera notifies the proviola of this.

- If this frame has not been loaded before, or has been updated, it is requested from the camera, and cached locally.
- If the frame has been requested before, and has not been updated, it is loaded from cache.
- When the user updates the focused frame, it is sent to the camera.

7   Applications

7.1   The Camera as a Service Broker

In our design of interactive movie-making, the camera takes a centralized place at the server, containing the "master copy" of the movie and providing write access to it. This access is not only usable by the proviola or the PA, but could also be used by other tools that work with a formal proof.

To provide access to the movie, the camera needs to allow arbitrary tools to register as either a *subscriber* or as a *publisher* for a movie.

Subscribers A **subscriber** to a movie can obtain the frames of the movie in which they are interested. When a frame is changed in the camera-side movie, it notifies each subscriber of the changed frame. Subscribers can then update the frame. Our intention is that subscriber access be read-only.

Publishers By contrast, when registered as a **publisher**, a tool can write in the movie. We do not believe there should be any restriction to the types of content that publishers can write, but do require that the movie should be *extended*: if the tool produces information of a different type than already exists in the movie, it should be added alongside the existing data, not replace it. So, although it is possible for multiple tools to write into the command section of a frame, for example a tool processing a script and the proviola+editor of Section 6, a tool that does not produce commands should not write in the command section.

The idea of tools listening for changes in proof state is not new: it was previously mentioned by Aspinall, Lüth and Winterstein [1] as future work for their broker architecture. Using the camera as a broker is similar, with the advantage that history does not need to be kept by the individual tools but can be requested from the camera: an additional benefit is that subscribers coming late to the party still have access to the full history of a proof, which we consider especially interesting for our ongoing work towards a repository of formal proofs.

7.2   Towards an Encyclopedia of Formal Proof

With some modifications, the proof movie can be used as the data structure underlying an encyclopedia that we envisage containing formal proofs together with an informal narrative explanation, and provide a toolbox for using and manipulating such composite "articles", as originally sketched in [4].

The movie provides a generic structural view of different kinds of script, so that for a tool wanting to manipulate a proof on a structural level, it is not necessary to know the exact details of the PA concerned, and the history contained in the movie implies that tools developed later are not required to replay the entire script through a PA. Furthermore, because the camera provides a central interface for accessing the movie, tools do not have to be on the server hosting the encyclopedia, allowing others to use the proofs without undue restrictions.

To implement the encyclopedia, we do need to create several tools ourselves, that together form the backbone of the encyclopedia.

First, we need to store and retrieve the movie. Possible candidates for storage are the file system, a database or a version control system which keeps track of the history of a proof. Retrieval cannot easily be expressed by communicating with the camera, and will most likely warrant an extension to that concept.

The second addition is aimed at adding informal explanation to a proof: a "commentary track", where an author can comment on the proof script and the output that the PA produces. This requires a revision of the movie-concept. Because several frames in a movie might be explained in a single, continuous narrative and commands might be repeated several times in the narrative, we will add *scenes* to the movie. A **scene** contains a single, informal description of a group of frames, and a reference to these frames. A scene can refer to zero or more frames, and a frame may belong to zero or more scenes.

As well as linear narrative, scenes could be used to store alternative problem-solving approaches, such as using different lemmas or automated proof search.

As previously mentioned, we have investigated the movie's ability to capture a large body of proofs by filming the Coq standard library [3]. The filming took less than one hour, including a two-second sleep between processing each file, which was inserted to prevent overloading the server running ProofWeb. The resulting films can be inspected at `http://mws.cs.ru.nl/proviola`, which serves the films quickly, even for large developments such as that of the Riemann integral, at around two MB in size. Movies typically are much smaller, however, up to five hundred kB. The movie size scales in the size of the original script: a long script that uses tactics producing many subgoals also creates a large movie.

In further work [12], we elaborate the concepts of scenes and commentary, and describe tooling for creation and display of a commented movie, by investigating movies of course notes on Software Foundations by Pierce *et al.*[11], available at `http://mws.cs.ru.nl/proviola/movies/sf`.

8 Related work

We have already mentioned the PG kit system and its protocol, PGIP [1]. Interactive movie-making is similar to its broker architecture. While PGIP focuses on the message structure for the dynamics of communication between author and PA, a movie freezes a sequence of such messages into a static data structure. In that respect, our approach is orthogonal to that of PGIP: the movie stores the result of an interactive session while PGIP deals with the messages and protocol

needed to generate it. Instead of a prototype based on the camera and ProofWeb, we could just as well have filmed an interaction with the PG kit broker.

Wenzel's Jedit editor [13] also considers a finer grained analysis between editor and Isabelle. But this is more oriented towards parallelisation of proof steps than the human-orientated details of interaction and representation of proofs considered here.

The movie file format is an XML-based representation of formal mathematical documents. The best-known and most versatile XML-based format for mathematical documents is OMDoc [9]. We decided not to use this format internally in our system. OMDoc is about document *structure*, while the content of a movie is unstructured system input/output. Still, if needed, movies could be easily converted to OMDoc, so our choice of a simpler format is not a limitation.

The history stored in a movie is not the same as history in, say, a web browser: the movie stores both the messages sent and the responses received, while a browser only stores a reference to pages visited, and needs to obtain the page again when the user requests it. This approach of "recalculating" a result is similar to what happens when a user sends undo commands, both when using a normal program (such as a text editor) and when using a PA. In our proviola there is no recalculation, because all the "history" is stored in the movie.

A command in a PA can consist of a combination of more primitive tactics, for example two tactics sequentially composed with the ";" tactical in Coq. One might wish to see the intermediate state after the first tactic in such a sequence. Coq does not support this small step execution model, so in our implementation, the movie cannot provide this information. A system like Matita [10] does, so a proviola based on Matita could potentially expose this refined execution.

Integrating a "commentary track" into the movie, by collecting frames into "scenes" and adding a narrative text to it, goes in the direction of earlier work on tmEgg [6]. There we started with a mathematical document, written in the editor TEXmacs, as the backbone with a Coq proof script (a `.v` file) underneath it. At any point one could consult the formal proof by opening a Coq session within the document and executing Coq commands. This means Coq is started up and brought into the required state, and then the selected commands are executed. This works quite well for small scripts, but can take a lot of time for larger proofs. Also, it requires the PA to be available, with the right libraries, which makes the mathematical document less "self-contained". Another hindrance is that tmEgg relies heavily on TEXmacs, which means that one is dependent on yet another application to run smoothly. The present work allows movies to play the role of the Coq interaction (as a proxy) within an interactive mathematical document. This can basically be done within any editor and thus relieves the interactive mathematical document from being tied to either TEXmacs or Coq.

9 Conclusion

In conclusion, we claim that our refactored interaction model and its associated data structure are an important contribution in their own right. But our in-

terest is in how this data structure may be further extended to support richer interaction and display as part of a MathWiki.

References

1. Aspinall, D., Lüth, C., Winterstein, D.: A framework for interactive proof. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) Towards Mechanized Mathematical Assistants. LNAI, vol. 4573, pp. 161–175 (2007)
2. Coq-Club Mailing List: The Coq-Club mailing list. Mailing List, archived at: `http://logical.saclay.inria.fr/coq-puma/topics`
3. Coq Development Team, T.: The Coq standard library. Library documented on `http://coq.inria.fr/stdlib`, obtained on March 5, 2010.
4. Corbineau, P., Kaliszyk, C.: Cooperative repositories for formal proofs. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM. LNAI, vol. 4573, pp. 221–234. Springer (2007), `http://www4.in.tum.de/~kaliszyk/docs/cek_p3.pdf`
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison–Wesley (1994), first edition.
6. Geuvers, H., Mamane, L.: A document-oriented Coq plugin for TeXmacs. In: Libbrecht, P. (ed.) MathUI workshop, MKM 2006 conference, Wokingham, UK. `http://www.activemath.org/~paul/MathUI06/` (2006)
7. Kaliszyk, C.: Web interfaces for proof assistants. In: Autexier, S., Benzmüller, C. (eds.) Proceedings of UITP'06, Seattle. ENTCS, vol. 174[2], pp. 49–61 (2007), `http://www4.in.tum.de/~kaliszyk/docs/cek_p2.pdf`
8. Kaliszyk, C.: Correctness and Availability. Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web. Ph.D. thesis, Radboud University Nijmegen (2009), `http://www4.in.tum.de/~kaliszyk/docs/ck_thesis_webdoc.pdf`
9. Kohlhase, M.: OMDoc – An Open Markup Format for Mathematical Documents [version 1.2], LNAI, vol. 4180. Springer (2006)
10. Matita Team: Matita interactive theorem prover. Web page, obtained from `http://matita.cs.unibo.it/`
11. Pierce, B.C., Casinghino, C., Greenberg, M.: Software foundations. Course notes, online at `http://www.cis.upenn.edu/~bcpierce/sf/` (2010)
12. Tankink, C., Geuvers, H., McKinna, J.: Narrating formal proof (work in progress). Submitted to UITP '10, `http://cs.ru.nl/~carst/files/narration.pdf`.
13. Wenzel, M.: Parallel proof checking in Isabelle/Isar. In: Reis, G.D., Théry, L. (eds.) PLMMS 2009. ACM, Munich (2009), `http://www4.in.tum.de/~wenzelm/papers/parallel-isabelle.pdf`