Hypertext Navigation of ACL2 Proofs with XMLEye

Antonio García-Domínguez
Escuela Superior de Ingeniería
Universidad de Cádiz
C/Chile 1, CP 11003
Cádiz, Spain
antonio.garciadominguez@uca.es

Francisco Palomo-Lozano
Escuela Superior de Ingeniería
Universidad de Cádiz
C/Chile 1, CP 11003
Cádiz, Spain
francisco.palomo@uca.es

Inmaculada Medina-Bulo
Escuela Superior de Ingeniería
Universidad de Cádiz
C/Chile 1, CP 11003
Cádiz, Spain
inmaculada.medina@uca.es

ABSTRACT

Difficult problems often require complex solutions, and the proofs checked by ACL2 are no exception. There are steep learning curves involved both in producing the proof script and analyzing its long and complex results. Existing tools, such as DrACuLa or ACL2s, tend to focus more on the first aspect than the second one.

We have developed XMLEye, a framework for creating viewers for complex structured documents. Upon this framework, we have created a tool which can reorganize ACL2 proofs and present them in a more accessible format. First, the plain text proof produced by ACL2 is converted into an intermediate form. Then, it is rendered as hypertext through a transformation described by a collection of external scripts.

We introduce the overall design and implementation of XMLEye as a framework and discuss the customizations required to reorganize and render ACL2 proofs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Graphical user interfaces (GUI)*; I.7.2 [**Document and Text Processing**]: Document Preparation—*Hypertext, markup languages*

General Terms

Design, Experimentation

Keywords

Formal methods, ACL2, proof presentation, XML, XSLT, XHTML

1. INTRODUCTION

As an industrial-strength theorem prover, ACL2 has been successfully used to prove properties about complex hardware and software. However, new users (both in academia and industry) are confronted with a steep learning curve because of its inherent complexity and its text-based interface.

Several projects [8, 24] are already underway to create a graphical interface to ACL2 (or restricted subsets thereof) which is more intuitive and aesthetically pleasing. Most of the new functionality is geared towards making proof scripts easier to write, though some facilities for browsing their ACL2 output are also added. Nevertheless, ACL2's output remains essentially the same: a long, linear and slightly formatted text describing a series of proofs with goals that can branch widely and deeply. This is difficult to read by humans and to process by external tools.

We created XMLEye with this sort of problem in mind. XMLEye is a generic framework for creating viewers for complex structured document formats. It is licensed under the GNU GPL and available at [13, 12]. We have used it to implement a tool which can process a considerable subset of ACL2's output and present it in hypertext form, highlighting the different parts of the proof, establishing links between its elements and deriving new information of interest to the end user. Our work on top of XMLEye is divided into two parts: a Perl program which converts the proof to XML, and a collection of XSLT stylesheets to reorganize the proof and render it as XHTML.

This paper is organized as follows. Section 2 briefly introduces the technologies needed to understand the rest of the text. Section 3 shows how ACL2 proofs are processed by XMLEye and discusses the work involved. After a qualitative evaluation of the current status of the tool in section 4, we compare our work with similar approaches and present some conclusions along with an outline of our future work.

2. TECHNICAL BACKGROUND

At its core, XMLEye rests on top of three technologies from the W3C (World Wide Web Consortium): XML (eXtensible Markup Language) [20], XSLT (eXtensible Stylesheet Language Transformations) [4], and XPath (XML Path language) [5]. In this section we will briefly introduce them and show some examples of their usage.

2.1 Markup Languages: XML and XHTML

Plain text documents by themselves lack information about their expected presentation or meaning. We can convey these by marking parts of the original text with *tags*. For instance, a tag could indicate that it should be rendered in a heavier font (presentation markup), or that some text is a heading (logical markup).

A *markup language* defines a set of tags with their own syntax and semantics. In most cases, markup languages are specializations or *vocabularies* of *metalanguages*, which describe entire families of markup languages. HTML [15] (HyperText Markup Language) itself, which most of the Web is based on, is an SGML [14] (Standard Generalized Markup Language) vocabulary.

However, SGML as a whole is overly complex. Most markup languages in use today are based on an ever-popular subset of it: XML 1.0 [20]. XHTML [21], for example, is an XML version of HTML. There exist several technologies to describe XML vocabularies and validate documents against them, but they lie beyond the scope of this paper. An example of a simple XML document describing a point in 3D space is shown in listing 1.

One important feature of XML is its extensibility. To avoid name collisions, all element and attribute names can be qualified with *namespaces*. Most of the time, we can freely mix and match elements and attributes from different namespaces, including user-defined ones.

XML tools and technologies (such as XSLT and XPath, described below) usually model structured documents as rooted trees including nodes of different kinds: element nodes, text nodes, and so on.

2.2 Transforming XML Documents: XSLT

We will often need to process XML documents and convert them to another format, be it plain text, HTML, XML or just about any sequence of bytes. Writing these transformations using a general-purpose programming language is, except for some trivial cases, unwieldy and inefficient.

XSLT 1.0 [4] (XSLT hereafter) is a specialized XML vocabulary for describing *stylesheets* that indicate how the input XML document should be processed. It favors a declarative and functional style which can be heavily optimized by most available XSLT processors. It is based on the concept of a *template*, which is applied to produce a subtree of the output when the current node from the input tree matches some predicate.

A sample XSLT stylesheet for the previous example can be found in listing 2. It copies the original document, overwriting the original y-coordinate value with the z-coordinate value.

2.3 Selecting XML Nodes: XPath

XSLT 1.0 uses another W3C standard: XPath 1.0 [5] (XPath from now on). This specification defines a language to declaratively construct queries on XML documents.

With XPath, we can select one or several nodes from the

```
<point>
  <x>1</x>
  <y>2</y>
  <z>0</z>
</point>
```

Listing 1: A sample XML document

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

  <!-- When an 'y' element is found... -->
  <xsl:template match="y">
    <!-- Replace y-coordinate with z-coordinate -->
    <y><xsl:value-of select="../z"/></y>
  </xsl:template>

  <!-- Otherwise... -->
  <xsl:template match="*">
    <!-- Copy elements and attributes -->
    <xsl:copy>
      <!-- Descend recursively -->
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Listing 2: A sample XSLT stylesheet

XML document tree and optionally compute values from them. XPath includes a set of predefined functions that can be extended in an implementation-dependent way. Let us present some simple queries on the sample XML document shown in listing 1:

- `"/point/x"` selects the x-coordinate element from the point.
- `"count(//text())"` returns how many text nodes exist.
- `"//x + //y + //z"` sums the x, y and z coordinates.
- `"//*[. > 1]"` selects all coordinates that contain values higher than 1.

3. ACL2 PROOFS IN XMLEYE

In the previous section, we introduced the main technologies behind XMLEye. We will now expand on how we used the XMLEye framework to create a viewer for ACL2 proofs.

XMLEye [13, 12] itself is a graphical Java application that, by default, can only browse XML documents. To open any other format, we need to integrate an *external converter* to XML. XMLEye supports *document format descriptors* for this purpose. They are simple XML documents which associate a set of file extensions (i.e., `.lisp` and `.acl2`) with the proper commands for XML conversion and editing.

To browse ACL2 proofs as such in XMLEye, we have defined our own document format descriptor, converter and XSLT stylesheets. Other formats (such as outputs from other theorem provers) could be adapted in a similar fashion. In fact, some [18, 23] already generate XML output.

XMLEye can now open our new document format, but it will be rendered using generic XML views, which do not produce good enough results. To illustrate our point, we will use a modified version of the proof script of the Towers of Hanoi tutorial [25] as our running example in this section. We have divided it across three files:

books/hanoi.acl2 Following the recommended approach to book compilation described in the ACL2 documentation [17], this file establishes the initial world before certifying the *hanoi* book. In this case, it just defines the HANOI package and starts the certification process. An abridged version is shown in listing 3.

books/hanoi.lisp An ACL2 book defining the MOVE and HANOI functions to be analyzed, and LEN−APPEND, a lemma that is later used in the proof of the main theorem. See listing 4.

hanoi-use.lisp Includes the *hanoi* book and uses it to prove the main theorem, a result about the number of moves required by the HANOI function defined therein to solve the problem. See listing 5.

In XMLEye, a collection of graphical widgets manages the document tree structure and, upon selection, presents the XHTML rendering of the current node to the user. Viewing the proof generated by the `hanoi-use.lisp` script as XML would look as in figure 1. Soon, one realizes that this structure is cumbersome to navigate and difficult to understand: the information is too scattered to be useful.

For this reason, users can install external XSLT stylesheets to define their own transformations. The stylesheets, which are able to reorganize the whole document and render nodes as XHTML, can be freely switched separately at run-time. Compare with figure 2, where the same information is applied a different stylesheet.

In short, XMLEye uses a four-stage process (depicted on figure 3) to browse any document: XML conversion, document reorganization, node rendering and navigation. We will talk in more detail about each of these steps in the following subsections.

3.1 Conversion

Had ACL2 produced the necessary XML output, no conversion would be needed. However, this is not the case: ACL2's plain text output is a mix of natural language descriptions and S-expressions which is difficult to parse automatically. We first considered modifying ACL2 to have it produce XML, but we discarded this option. A number of modifications would have to be done and subtle soundness bugs could be introduced. Additionally, it would be expensive to maintain these modifications over different versions of ACL2.

```
(defpkg "HANOI"
  (union−eq ∗acl2−exports∗ ∗common−lisp−[...]∗))

(certify−book "hanoi" 1)
```

Listing 3: `books/hanoi.acl2` **— initial world**

```
(in−package "HANOI")

(defun move (a b)
  (list 'move a 'to b))

(defun hanoi (a b c n)
  (if (zp n)
      nil
    (if (equal n 1)
        (list (move a c))
      (append (hanoi a c b (1− n))
              (cons (move a c)
                    (hanoi b a c (1− n)))))))

(defthm len−append
  (equal (len (append x y))
         (+ (len x) (len y))))
```

Listing 4: `books/hanoi.lisp` **— basic definitions**

```
(include−book "books/hanoi")

(defthm hanoi−moves−required
  (implies (and (integerp n)
                (<= 0 n))
           (equal (len (hanoi::hanoi a b c n))
                  (1− (expt 2 n)))))
```

Listing 5: `hanoi-use.lisp` **— main theorem**

Therefore, we need a converter that can be integrated into XMLEye. Only a few constraints are imposed by the framework: progress is reported through the standard error stream, results are sent back through the standard output stream, and success is indicated through a zero status code. Our current approach uses a Perl program that takes the main proof script and produces an XML version of the proof generated by ACL2. It follows four steps:

1. *Dependency analysis:* we need to produce a tree listing, for each proof script file used in the proof, all books and certification scripts it depends on and the events defined in it. We start from the main proof script and analyze their S-expressions, descending recursively through their invocations to INCLUDE−BOOK and CERTIFY−BOOK.

   We require that users provide a certification script for every book (i.e., a `.acl2` file containing the initial certification world and the appropriate CERTIFY−BOOK command) or, at least, a `cert.acl2` file, according to the guidelines of the BOOK−MAKEFILES section in the ACL2 documentation [17].

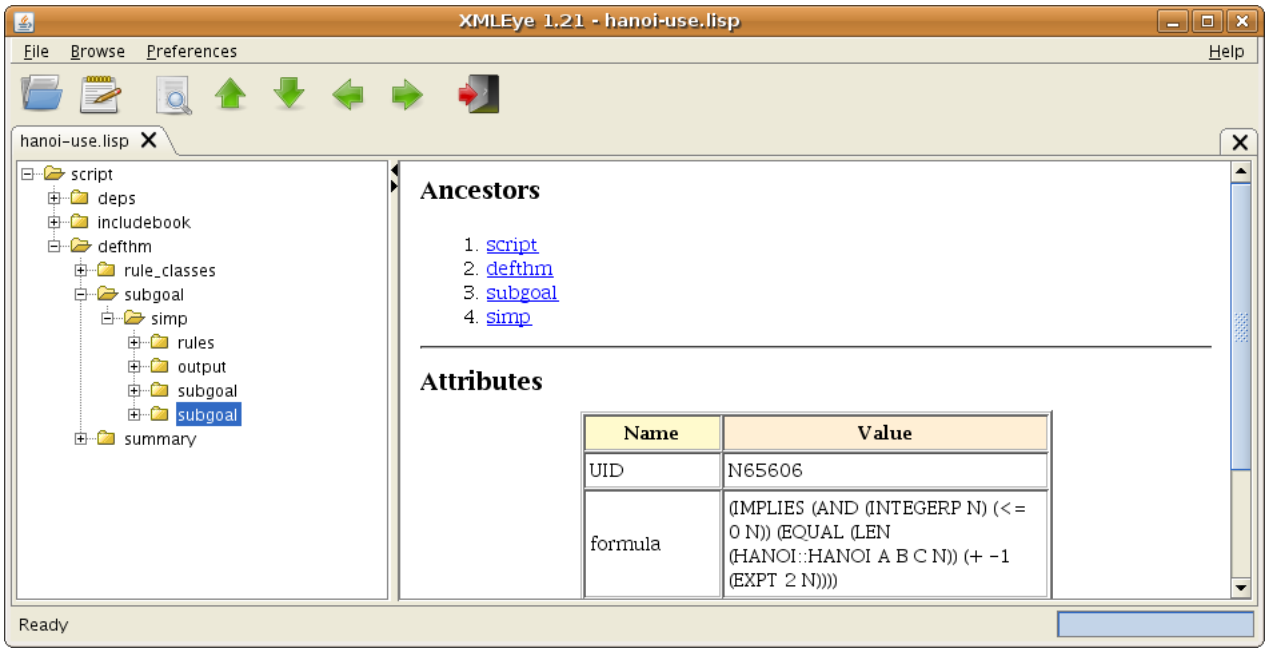

Figure 1: XMLEye browsing a proof: `xml` stylesheets

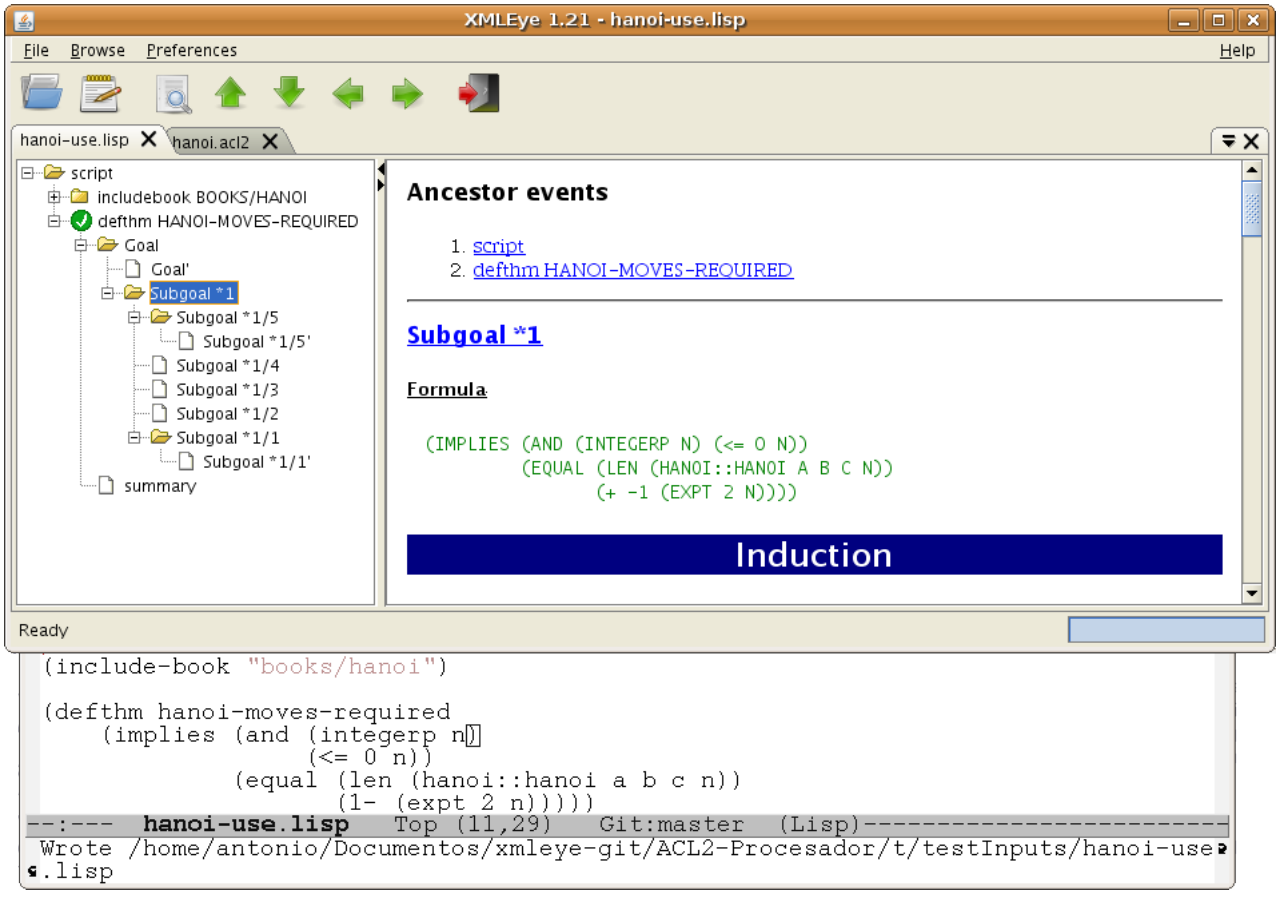

Figure 2: XMLEye browsing a proof: `ppACL2` stylesheets

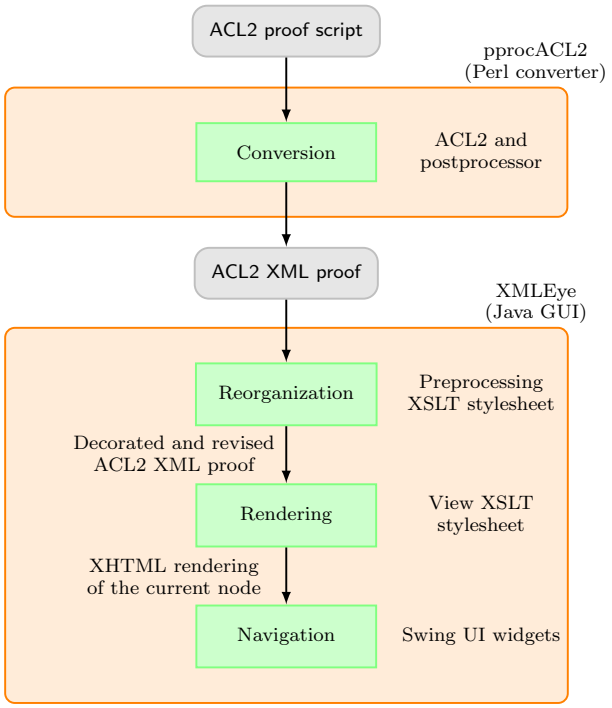

Figure 3: Overall architecture of the tool

```
<deps>
  <dep book_dir="books/" book_file="hanoi.acl2">
    <deps>
      <dep book_dir="" book_file="hanoi.lisp">
        <deps>
          <events>
            <event>HANOI::MOVE</event>
            <event>HANOI::HANOI</event>
            <event>HANOI::LEN-APPEND</event>
          </events>
        </deps>
      </dep>
      <events>
        <event>HANOI</event>
      </events>
    </deps>
  </dep>
  <events>
    <event>
      ACL2::HANOI-MOVES-REQUIRED
    </event>
  </events>
</deps>
```

Listing 6: XML dependency tree for `hanoi-use.lisp`

Before continuing to the next step, we convert each dependency to XML through a post-order traversal of the tree. This allows the converter to link each symbol reused by the main proof to the place where it is defined.

The XML dependency tree for our running example is shown in listing 6:

- `hanoi-use.lisp` defines the main theorem and depends on the certification script `hanoi.acl2`, contained within the relative path `books/`.
- `books/hanoi.acl2` defines the HANOI package and depends on the book that it certifies, `hanoi.lisp`, which is contained in the same directory.
- `books/hanoi.lisp` defines three events inside the HANOI package: MOVE, HANOI and LEN-APPEND. As we can see, it does not depend on any file.

2. *ACL2 invocation:* ACL2 is invoked on each modified proof script and results are saved to independent `.out` files in the corresponding directories.

   If the XML version of the proof saved from a previous conversion run exists and its modification time is more recent than the modification time of the proof script, then we can assume that the proof is unchanged. Thus, we can stop at this point and safely keep the existing version as is.

   This behavior is very similar to that of a *makefile*.

3. *Proof analysis:* every generated `.out` file is analyzed and an in-memory object-oriented representation is constructed. Roughly speaking, the proof is divided into a list of pairs with each S-expression and its corresponding output, and then converted into an object whose type depends on the ACL2 command used.

   If there is no specific code for analyzing a particular command, the analyzer will fall back to printing a list of paragraphs and S-expressions. If present, the summary will be processed as usual. This is useful, for instance, when evaluating functions or calling macros.

   On the other hand, if the specific code for analyzing some command fails to parse some part of its output, a fatal error will be raised. This is important for regression testing: there have been some subtle changes in ACL2's output over time, and it would be difficult, if not impossible, to check manually for them.

4. *Proof conversion:* using the in-memory representation of a `.out` file proof, an XML document which reflects all the extracted information is produced. It includes an XML representation of the dependency tree obtained in the second step. We have striven to keep intact as much of the original ACL2 output as possible, while making more information readily available to the XSLT stylesheets.

   For instance, whitespace in S-expressions is preserved, and the original text of the ACL2 proof is copied verbatim. However, XML markup has been added so the XSLT stylesheets can tell whether a paragraph contains regular text or an S-expression.

   Listing 7 shows the result from converting the output of the HANOI-MOVES-REQUIRED DEFTHM event. All information is contained in a *defthm* node, with attributes detailing the name of the event, the original S-expression (preserving whitespace) and whether the :OTF flag is active or not.

```
<defthm name="HANOI-MOVES-REQUIRED"
        otf="NIL"
        formula="(DEFTHM HANOI-MOVES [...])">
  <rule_classes>
    <rule_class type=":REWRITE"></rule_class>
  </rule_classes>
  <subgoal label="Goal"
           formula="(IMPLIES (AND [...]))">
    <simp>
      <rules>
        <rule name="NIL" type="case analysis" />
      </rules>
      <output>
        <paragraph>
          By case analysis we reduce the conjecture to
        </paragraph>
      </output>
      <subgoal label="Goal'" [...]>
        [...]
      </subgoal>
      <subgoal label="Subgoal *1" [...]>
        <induction plan="(AND (IMPLIES [...]))">
          [...]
        </induction>
      </subgoal>
    </simp>
  </subgoal>
  <summary>
    <warnings></warnings>
    <time total="0.00" prove="0.00" [...]/>
    <rules>
      [...]
      <rule name="HANOI::HANOI"
            type=":DEFINITION" />
      [...]
    </rules>
  </summary>
</defthm>
```

Listing 7: Abridged XML version of a proof

This node is the root of a subtree which includes a sequence of elements detailing the classes of rules to be generated, the main goal (labeled `Goal`) with its S-expression, and an XML version of the information contained in the summary (warnings, time elapsed and rules used).

Each goal includes a node describing the proof technique used. In turn, each proof technique contains the subgoals that it produced.

For example, in the main goal, we can see that simplification by case analysis (an unnamed built-in rule) was used to generate `Goal'`, which would later become `Subgoal *1` (as only induction could be applied).

After saving the XML document to disk for later runs, it is dumped to standard output.

It is important to note that the converter is a standalone program, which does not depend on XMLEye at all.

3.2 Reorganization

Thanks to the converter, we can produce ACL2 proofs in XML, a format which is more amenable to automatic processing and filtering by a computer program. First, we need to show the user a clear view of the overall structure of the proof (see XMLEye's left pane in figures 1 and 2).

We cannot use the XML proof yet as it is: the information is too scattered to be usable by a human. Data must be integrated into a higher level view which is easier to understand. We might want to compute derived information useful to the user, or provide several levels of detail.

These problems can be solved by applying a *preprocessing stylesheet* which modifies the whole document right after opening the proof and before anything is shown to the user. Besides adding, removing or reordering the elements of the original XML document, we can hide them or label them with new captions or icons. All the XSLT stylesheet needs to do is set the correct XML attributes to the proper values, and the navigation step will take care of the rest.

Currently, we have defined three stylesheets for browsing ACL2 proofs. It is important to note that stylesheets can inherit rules from others and that they can be localized to several languages, by using the facilities set up in place by XMLEye. The user can switch back and forth between them at any time: the document will just need to be reprocessed when that happens.

ppACL2 This is the main ACL2 stylesheet, from which the other two are derived. It hides unnecessary clutter, labels elements according to their content and sets icons to indicate success and failure of each command. It also caches results for some expensive computations in hidden nodes, to save on processing time during the navigation step. For instance, it computes direct and inverse dependencies between each rule used and the events where they are defined.

reverse Inherits all the functionality from `ppACL2`, but it reverses the order in which commands are shown (last command is shown first). This is useful if we are handling long proofs and are mostly interested in the last few events.

summaries Also inherits its functionality from `ppACL2`. It strips away all information from the proof except for the summaries, which list the rules used and processing time required. Users might be only interested in the relationships between the different elements of the proof and the rules that they use.

We can also use the predefined `xml` stylesheet to view the original, unformatted XML document. This is mostly useful for debugging and for developing new XSLT stylesheets.

3.3 Rendering

The user can now navigate through a clear view of the structure of the ACL2 proof. The next problem to be solved is deciding how to present the details of the particular element that they have selected. Just as when we were reorganizing the proof, there is no single best way to do it.

However, this time we want a graphical representation of the information contained in the selected element. We could use regular GUI components, but then we would have to write Java code for each case and lose XMLEye's application independence. We will use *view XSLT stylesheets* instead.

These stylesheets produce XHTML documents which the user will be able to browse as hypertext in the final navigation step. Links to other elements of the proof are established through hyperlinks that follow an addressing scheme based on a subset of XPointer [6] restricted to XPath's functionality. These hyperlinks are generated through an extension XPath function that computes XPath expressions which uniquely identify any node.

The accepted syntax for these links is $F\#\text{xpointer}(E)$, where F is an optional absolute path to another XML file, and E is an arbitrary XPath expression which uniquely identifies one of its elements. If F is unspecified, the current document will be used. Custom XPath functions have been defined to easily obtain links to other elements in the current document.

Currently, only one view stylesheet has been developed for ACL2: `ppACL2`. It has the same name as the main preprocessing stylesheet used for ACL2, as they are meant to be used together. Likewise, there is an `xml` stylesheet which shows the raw information from the XML document. If we wish to examine reformatted versions of the original XML source code, we can use the `xmlSource` stylesheet.

Some of the useful features of the `ppACL2` view stylesheet are:

- From a specific goal, the user can see at a glance the S-expressions of its direct descendants. This is useful when the current goal splits into several simpler ones.

  For instance, while applying induction at `Subgoal *1` in our running example, ACL2 produces five subgoals called `Subgoal *1/5`, `Subgoal *1/4` and so forth. If any of them required a reasonably complex proof, we would have to scroll back and forth a long way to view all of their S-expressions. With XMLEye, however, these S-expressions are shown together with links to their nodes, where we will be able to examine them in more detail.

- The proof method used in each goal is clearly displayed and its most relevant information is shown in a separate table. For example, when induction is applied, the induction plan and the rules used to infer it (with hyperlinks to their definitions) are displayed in addition to the original text from ACL2.

- The summary is decorated with hyperlinks to the origin of each rule, which might be defined in external books.

  In the particular case of HANOI−MOVES−REQUIRED, there would be links to HANOI::HANOI, HANOI::MOVE and HANOI::LEN−APPEND. If an user clicked on any of these, a new tab would show the converted version of `hanoi.lisp` (the *hanoi* book) and automatically select the correct element.

- Views for the DEFTHM and DEFUN events list all events in the same ACL2 proof script that use their definitions. We already computed this information using the `ppACL2` preprocessing stylesheet (or any of its descendants).

  Suppose we joined into a single file the contents of the *hanoi* book with the HANOI−MOVES−REQUIRED theorem from `hanoi-use.lisp`. If we opened it with XMLEye, HANOI, MOVE and LEN−APPEND would display a link to the above theorem. This is because they belong to the list of rules used included in the summary for HANOI−MOVES−REQUIRED.

3.4 Navigation

Once the final document structure and the XHTML rendering of the current node have been computed, the final step is to let the user interact with them. To this effect, a Java GUI was implemented using slightly customized Swing widgets (see figures 1 and 2). The interface allows for quickly switching between the preprocessing and view stylesheets, customizing on a user-by-user basis the document format descriptors and performing searches on the document.

One important restriction which has been imposed from the beginning in XMLEye is that it would be strictly a viewer. There are no plans for integrating an editor into the framework, as each format and each user have their own requirements. Instead, XMLEye can invoke the user's favorite editor for that document format on the source file from which the XML document is derived (the main proof script in ACL2's case). It will optionally monitor the source file in the background and automatically reconvert and reload the document when the modification date changes.

This supports a common workflow in the ACL2 community, which uses an Emacs window divided into two buffers. One buffer is used for editing the proof script, and the other is a shell running ACL2, in which commands are pasted or loaded through some Emacs-Lisp code. With XMLEye, the user would open her proof script and then invoke the editor (i.e. Emacs, as in figure 2). XMLEye would invoke ACL2 with the new contents of the proof script every time that the user saved her changes. As resubmitting the whole proof script to ACL2 can be a lengthy process, this feature can be disabled from the GUI.

4. CURRENT STATUS

The current design of the converter was obtained iteratively, by adding increasingly complex proofs to the automated test suite. The design and implementation were improved every time a new proof could not be analyzed. Previous proofs served as regression tests, making sure that no errors or unexpected changes in the converter's output were introduced. These automated tests have also been useful in adapting the converter to the changes in newer versions of ACL2, while staying compatible with older versions. So far, every ACL2 version between 2.9 and 3.4 (latest version at the time of writing this paper) running on GCL passes all test cases.

At present, a considerable subset of ACL2's output is supported, such as that generated by the exercises in [16] or the tours in the ACL2 website. Proof scripts must consist of S-

expressions interleaved with single line comments: comment blocks are not supported yet. Multifile ACL2 projects (as the running example in section 3) including and certifying books can be converted, though further testing with more complex proofs is required.

Parsing the output from the following commands and options (if any) has been tested to work: DEFUN, DEFMACRO, DEFABBREV, DEFTHM (:OTF−FLG, :RULE−CLASSES including cliques), ENCAPSULATE (using LOCAL, but not including any books), CERTIFY−BOOK, INCLUDE−BOOK, DEFPKG, IN−PACKAGE, :PROGRAM, :LOGIC and THM. By design, if some part of the output for these commands could not be parsed as expected, a fatal error would be raised. Not all options for the above commands have been tested: hints or forcing rounds, for instance.

We have not implemented specific support for other commands yet. The analyzer will fall back upon finding an unknown command on printing a list of paragraphs and S-expressions and extracting information from the summary, if there is any. This allows the analyzer to parse macro and function calls to a limited degree.

We plan to extend the range of accepted document formats. We have recently added an XML converter for the YAML and JSON metalanguages and some generic stylesheets. Some candidates which would not require much additional work include Mozilla Firefox 3 bookmark backup files and Prover9 proofs.

Reorganization and rendering are mostly feature-complete on the side of the XMLEye framework. We expect that their design will remain mostly the same for the next versions, except for an upgrade to XSLT 2.0 and XPath 2.0, which offer significantly more functionality than their 1.0 revisions. If anything, it is the stylesheets which will need to be kept up to date against future versions of the converters. Links to ACL2's online help could be added, among other things.

We consider navigation to be quite usable, but it could be improved further. The default rendering engine included with Swing is rather limited: it only supports HTML 3.2 and a very restricted subset of the CSS standard. It would be interesting to test some more advanced engines, such as FlyingSaucer [10] or Cobra [22].

We are considering developing an entirely Web-based interface for XMLEye. This interface could use an embedded lightweight web server such as Jetty [19], taking advantage of the fact that the view stylesheets already produce mainly standard XHTML documents. For added speed, the web interface could remain behind an HTTP cache. This would allow for easy exporting and browsing of ACL2 proofs through the World Wide Web.

This would be a major step ahead in the spirit of *mathematical knowledge systems*. For example, HELM [1] can translate Coq and (experimentally) NuPRL proof scripts to an internal XML representation and apply XSLT transformations to obtain MathML (content and presentation), and HTML.

5. RELATED WORK

We have already referred to most of the related work in previous sections. However, we would like to use this section to collect and analyze them in more depth. We will discuss each topic related to our tool in the following subsections.

5.1 Graphical interfaces for ACL2

Some graphical interfaces for ACL2 already exist, but their focus is the opposite to ours: they attempt to be alternative interfaces to work with ACL2, instead of the usual Emacs and ACL2 read-eval-print loop combination, which many newcomers find hard to master.

ACL2s [8] is an Eclipse-based environment which hooks into a modified version of ACL2 and uses Eclipse's facilities for displaying ACL2's output and proof tree, and editing the proof script. In this way, it implements "The Method" [16]. The user can switch between different modes, which offer varying subsets of the full capabilities of ACL2. ACL2s extends ACL2 with some new features, such as a powerful termination analysis algorithm.

DrACuLa [24] integrates most of ACL2 into DrScheme, a Scheme environment which offers additional features such as an integrated GUI and static analysis. It has been used with considerable success in university courses. Students can reason about graphical interactive programs by loading the proper custom module or *teachpack*. It has been reported that this increases their motivation for learning formal methods.

Our tool happens to be mostly orthogonal to ACL2s and DrACuLa. Nevertheless, it would be worthwhile to try and integrate it as one more pane in ACL2s. It would require some work, though: the interface would have to be reimplemented in SWT, the GUI toolkit used by Eclipse. However, most of the backend code and the XML converter could be used with few changes.

5.2 XML-Based Tools for ACL2

Ruben Gamboa [11] proposed using XML documents to write proof scripts. In the spirit of the literate programming approach proposed by Knuth, they would be augmented with information that would help readers. He also suggests that, by using XML, it would be easier to implement external tools for processing proof scripts.

We could say that our work covers the other end of the spectrum: rather than helping users read proof scripts, we are helping users read the proofs themselves. In fact, he proposes rendering the list of theorems used in a proof as XML. That is precisely one of the things that our XML converter performs. Our XSLT stylesheets use these lists to compute direct and inverse dependencies between the available rules and the events using them.

5.3 Theorem Provers with XML Support

Finally, it is noteworthy that theorem provers with support for XML output already exist, such as Mizar [2] or Prover9 [18] (Otter's successor). The motivation [23] behind Mizar's switching to XML as the main output format was easing its integration with external tools for semantic

processing and presentation. Some tools have already benefited from this approach, such as the Alcor interface for Mizar [3].

We argue that ACL2 could benefit just as well from an XML version of its output. We are concerned about keeping ACL2 reasoning process sound through such a change, though, which is why we think the approach of using a converter is a safer bet. One particular concern that the Alcor developers raised was that the terms in the reported proof are not exactly those that the user reported, but rather those that the Mizar tool used internally.

We have tried to avoid this sort of problem as much as possible in our converter: in fact, the original proof script (including whitespace) can be derived from the converted XML proof.

We are currently considering how to properly handle macros, as they can expand to just about any S-expression and cannot be analyzed statically. It might be necessary to invoke ACL2 to expand these macros first, but no information should be lost in the process: the original unexpanded call to the macro should remain.

6. CONCLUSIONS AND FUTURE WORK

ACL2 is an industrial-strength system that has been successfully used to prove important properties about complex software and hardware systems. However, it is not easy to learn, and the proofs it produces can be hard to read. Although proofs are organized as trees, the proof itself is a long, linear and slightly formatted text. This format hinders the creation of new external tools, as it is difficult to parse in detail and lacks a formal description.

Other tools already exist, but they focus mostly in providing a friendlier interface to ACL2. We have developed a preliminary version of a tool capable of browsing ACL2 proofs as hypertext. Links can be established between different proof elements and rich formatting can be used to highlight the most important pieces of information.

To do this, we have developed an XML converter for ACL2's output and added the required customizations to XMLEye, a framework of our making for creating viewers for complex and structured documents. The converter was developed through an iterative process with automated regression testing, which ensures that it will continue working as expected with newer versions of ACL2 and the converter itself. The XMLEye extensions consisted mostly of a set of XSLT stylesheets to reorganize ACL2 proofs and render their elements as XHTML.

In this work, we covered the most important aspects behind the design and implementation of both the XMLEye framework and its ACL2-specific customizations. XMLEye follows a four-step process to browse any document: XML conversion, reorganization, XHTML rendering and navigation. XML conversion is performed by an external standalone converter satisfying some simple constraints. Reorganization and XHTML rendering are performed by separate sets of XSLT stylesheets. Finally, navigation is done by using a tab-oriented Java GUI.

The tool supports a considerable subset of ACL2's output and offers a clear view of the structure and individual elements of its proofs. The user can switch between different ways to reorganize the proof and render its nodes, and can define her own without having to modify XMLEye's code. Theorems and functions in the proof include links to their direct and inverse dependencies. XMLEye can open the ACL2 proof script with the user's preferred editor and reload it when changes are detected.

XMLEye and our ACL2 customizations can still be improved, however. Not all commands or options are supported, and the XHTML rendering engine does not implement some useful features. We will need to add new test cases with more complex proof scripts, and continue refining the converter's design. We are also considering developing a Web-based interface to XMLEye, since XHTML code is already being produced in the rendering step.

Finally, we plan to upgrade XMLEye to support XSLT 2.0 and XPath 2.0, and add stylesheets to navigate through the output of other theorem provers, such as Prover9 or Mizar. We believe that these will be easier to support, as they already produce XML output.

7. REFERENCES

[1] A. Asperti, L. Padovani, C. Sacerdoti Coen, F. Guidi, and I. Schena. Mathematical Knowledge Management in HELM. *Ann. Math. Artif. Intell.*, 38(1–3):27–46, 2003.

[2] Association of Mizar Users. Mizar homepage. `http://www.mizar.org`, Dec. 2008.

[3] P. A. Cairns and J. Gow. Integrating searching and authoring in Mizar. *J. Autom. Reasoning*, 39(2):141–160, 2007.

[4] J. Clark. XSL Transformations (XSLT) Version 1.0. Recommendation, W3C, Dec. 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`. Latest version available at `http://www.w3.org/TR/xslt`.

[5] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Recommendation, W3C, Dec. 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`. Latest version available at `http://www.w3.org/TR/xpath`.

[6] S. DeRose, E. Maler, and R. Daniel. XPointer xpointer() Scheme. `http://www.w3.org/TR/xptr-xpointer`, Dec. 2002.

[7] P. C. Dillinger, P. Manolios, and D. Vroon. ACL2s homepage. `http://acl2s.peterd.org/acl2s/doc`, Jan. 2009.

[8] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore. ACL2s: "The ACL2 Sedan". *International Conference on Software Engineering Companion*, pages 59–60, 2007. Software available at [7].

[9] C. Eastlund, D. Vaillancourt, C. Schonwald, and K. McGrady. DrACuLa homepage. `http://www.ccs.neu.edu/home/cce/acl2`, Feb. 2009.

[10] FlyingSaucer homepage. `https://xhtmlrenderer.dev.java.net`, Mar. 2009.

[11] R. Gamboa. Writing literate proofs with XML tools. In *Proceedings of the Fourth International Workshop*

on the ACL2 Theorem Prover and Its Applications (ACL2-2003), Boulder, Colorado, USA, July 2003.

[12] A. García Domínguez. XMLEye Wiki. `http://wiki.shoyusauce.org`, Oct. 2008.

[13] A. García Domínguez. XMLEye Forge at RedIris. `https://forja.rediris.es/projects/csl2-xmleye`, Mar. 2009.

[14] C. F. Goldfarb. ISO 8879: Standard Generalized Markup Language (SGML). Technical report, International Standards Organization, 1986.

[15] I. Jacobs, A. L. Hors, and D. Raggett. HTML 4.01 Specification. Recommendation, W3C, Dec. 1999. `http://www.w3.org/TR/1999/REC-html401-19991224`. Latest version available at `http://www.w3.org/TR/html401`.

[16] M. Kauffmann and J. S. Moore. A Brief ACL2 Tutorial. `http://www.cs.utexas.edu/users/moore/publications/tutorial/rev3.html`, Nov. 2002.

[17] M. Kaufmann and J. S. Moore. *ACL2 Version 3.4: BOOK-MAKEFILES*. University of Texas, Austin, USA, Nov. 2007. Available at `http://www.cs.utexas.edu/users/moore/acl2/v3-4/BOOK-MAKEFILES.html`.

[18] W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/mace4`, Mar. 2009.

[19] Mort Bay Consulting. Jetty WebServer homepage. `http://www.mortbay.org/jetty`, Mar. 2009.

[20] J. Paoli, C. M. Sperberg-McQueen, T. Bray, F. Yergeau, and E. Maler. Extensible Markup Language (XML) 1.0 (Fifth Edition). Recommendation, W3C, Dec. 2008. `http://www.w3.org/TR/2008/REC-xml-20081126`. Latest version available at `http://www.w3.org/TR/xml`.

[21] S. Pemberton. XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition). Recommendation, W3C, Aug. 2002. `http://www.w3.org/TR/2002/REC-xhtml1-20020801`. Latest version available at `http://www.w3.org/TR/xhtml1`.

[22] The Lobo Project. Cobra: Java HTML Renderer and Parser. `http://lobobrowser.org`, Jan. 2009.

[23] J. Urban. XML-izing Mizar: Making semantic processing and presentation of MML easy. In M. Kohlhase, editor, *MKM*, volume 3863 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2005.

[24] D. Vaillancourt, R. Page, and M. Felleisen. ACL2 in DrScheme. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 107–116, New York, NY, USA, 2006. ACM. Software available at [9].

[25] B. Young. The Towers of Hanoi Example. `http://www.cs.utexas.edu/users/moore/acl2/v3-4/TUTORIAL1-TOWERS-OF-HANOI.html`, Aug. 2008.