(doc)Tools for a new generation of Tcl package documentation

Andreas Kupries ActiveState Corp 580 Granville Vancouver, BC CA
andreask@ActiveState.com

ABSTRACT

As part of general work on the ActiveTcl and Tcl Dev Kit distributions we invented a new documentation system for Tcl packages and use it to generate the nroff, HTML, and CHM documentation delivered with these distributions.

This paper describes the formats provided by this system, the packages handling them, especially their plugin interface, our experiences with these formats, their limitations, and ideas we have had for the possible evolution of the whole system in the future.

1. OVERVIEW

To have documentation for Tcl packages and applications in a single universal format is one of the recurring topics on comp.lang.tcl [13]. Especially given the high variety formats currently employed, like the *roff family [8], HTML [7], and even plain text.

The latest concrete attempt at creating such a format I am aware of was done for the Tcl-Blast! CD. At that time D. R. Hipp created TMML, the Tcl Manpage Markup Language [2], now maintained by Joe English.

The problem I had with TMML is that it is of XML-ancestry [14] and thus heavyweight in the sense that while

1. processing of such markup in pure Tcl is possible (See [15], [16]) it is not very efficient, good performance requires compiled extensions to the Tcl language.

2. editing the markup manually, just with an editor is possible it also causes the viewer to nearly drown in the markup versus the actual text. Reducing this problem requires special editors tailor-made for XML-based markup languages. It should be noted that compared to other DTD's TMML is relatively lightweight in this regard.

Thus the motivation for a more lightweight format was born. Lightweight as in easier to process in pure Tcl, and easier to edit without special tools. And a more tclish appearance would not hurt either.

The result of that motivation are the three doc* formats we describe here.

The remainder of the paper is structured as follows. In chapter 2 we explain the formats, their ancestry and where to find (the Tcl sources for processing) them. In the next chapter, 3 we present the API between the generic core processor and the plugins for the generation of the actual output.

Chapter 4 then discusses advanced topics, namely the automatic creation of keyword indices, tables of contents, parameterization, plugin hooks, etc.

Then we discuss the limitations of the system in chapter 5, provide conclusions in chapter 6, and at last chapter 7 talks about possible future work in this area.

2. THE DOC* FORMATS

As stated in the introduction the goal was to create a lightweight, easy to use and still powerful language for the creation of documentation, mainly manpages.

The result of this effort is the `doctools` system. The sources of this system are freely available under the same license as Tcl itself, as a module of the `tcllib` bundle of packages [11]. An application using the `doctools` packages is the doctools processor `dtp`, also freely available [12].

The system defines actually not a single format, but three. The primary format has the same name as the overall system, `doctools` and is intended for the creation of manpages. The other two formats are `doctoc` and `docidx`, for the creation of tables of contents, and of keyword based indices. While all of them can be written manually it is anticipated that documents in the secondary formats will most likely be generated automatically out of a set of documents written in the primary format.

Brief documents written in the defined formats can be seen in the figures 1, 2 and 3. They were taken from the documentation for tcllib itself, all of which is written in doctools, thus not only providing the means of processing the formats, but also serving as a pool of examples.

The result of converting the sha1 documentation in figure 1 into plain text can be seen in figure 4. Other output formats with predefined formatting engines[1] are nroff, HTML, TMML, LaTeX [6], and Wiki format [24].

The conceptual ancestor of the three formats is LaTeX and its metaphor of having text as the main part of the documentation, interspersed with markup commands.

[1]In other words coming as part of doctools in tcllib

Tcl '2003 Ann Arbor, Michigan USA
.

Figure 1: sha1 manpage in doctools

```
[manpage_begin sha1 n 1.0.3]
[moddesc   {sha1 hash}]
[titledesc {Perform sha1 hashing}]
[require Tcl 8.2]
[require sha1 [opt 1.0.3]]
[description]
[para]

This package provides commands to compute a SHA1 digests of arbitrary
messages.

[section COMMANDS]
[list_begin definitions]
[call [cmd ::sha1::sha1] [arg msg]]

The command takes a message and returns the SHA1 digest of this message
as a hexadecimal string.

[call [cmd ::sha1::hmac] [arg key] [arg text]]

The command takes a key string and a text and returns the hmac of the

[list_end]

[section EXAMPLES]

[para]
[example {
% sha1::sha1 "hello world"
2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
}]

[para]
[example {
% sha1::hmac "our little secret" "hello world"
a7ed9d62819b9788e22171d9108a00c370104526
}]

[keywords sha1 hashing security]
[manpage_end]
```

Figure 2: Excerpted table of contents for Tcllib

```
[toc_begin {Tcllib -- Table of Contents} Modules]
[item tcllib/modules/base64/base64.man base64 {Procedures to encode and decode base64}]
[item tcllib/modules/base64/uuencode.man uuencode {encode/decoding a binary file}]
[item tcllib/modules/base64/yencode.man yencode {encode/decoding a binary file}]
...
[item tcllib/modules/crc/cksum.man cksum {calculate a cksum(1) compatible checksum}]
[item tcllib/modules/crc/crc16.man crc16 {Perform a 16bit Cyclic Redundancy Check}]
[item tcllib/modules/crc/crc32.man crc32 {Perform a 32bit Cyclic Redundancy Check}]
[item tcllib/modules/crc/sum.man sum {calculate a sum(1) compatible checksum}]
[item tcllib/modules/csv/csv.man csv {Procedures to handle CSV data.}]
[item tcllib/modules/des/des.man des {Perform DES encryption of Tcl data}]
[item tcllib/modules/dns/tcllib_dns.man dns {Tcl Domain Name Service Client}]
[item tcllib/modules/html/html.man html {Procedures to generate HTML structures}]
[item tcllib/modules/irc/irc.man irc {Create IRC connection and interface.}]
...
[item tcllib/modules/smtpd/smtpd.man smtpd {Tcl SMTP server implementation}]
[item tcllib/modules/soundex/soundex.man soundex Soundex]
[item tcllib/modules/stooop/stooop.man stooop {Object oriented extension.}]
[item tcllib/modules/struct/struct_list.man list {Procedures for manipulating lists}]
[item tcllib/modules/struct/struct_tree.man tree {Create and manipulate tree objects}]
[item tcllib/modules/uri/uri.man uri {URI utilities}]
[toc_end]
```

Figure 3: Excerpted keyword index of Tcllib

```
[index_begin Tcllib {Keyword index}]
  [key C++]
    [manpage tcllib/modules/stooop/stooop.man stooop]
  [key CGI]
    [manpage tcllib/modules/ncgi/ncgi.man ncgi]
  [key DES]
    [manpage tcllib/modules/des/des.man des]
  [key DNS]
    [manpage tcllib/modules/dns/tcllib_dns.man dns]
  [key HTML]
    [manpage tcllib/modules/doctools/docidx.man docidx]
    [manpage tcllib/modules/doctools/docidx_api.man docidx_api]
    [manpage tcllib/modules/doctools/docidx_fmt.man docidx_fmt]
    [manpage tcllib/modules/doctools/doctoc.man doctoc]
    [manpage tcllib/modules/doctools/doctoc_api.man doctoc_api]
    [manpage tcllib/modules/doctools/doctoc_fmt.man doctoc_fmt]
    [manpage tcllib/modules/doctools/doctools.man doctools]
    [manpage tcllib/modules/doctools/doctools_api.man doctools_api]
    [manpage tcllib/modules/doctools/doctools_fmt.man doctools_fmt]
    [manpage tcllib/modules/doctools/mpexpand.man mpexpand]
  [key LaTeX]
    [manpage tcllib/modules/doctools/docidx_api.man docidx_api]
    [manpage tcllib/modules/doctools/docidx_fmt.man docidx_fmt]
    [manpage tcllib/modules/doctools/doctoc_api.man doctoc_api]
    [manpage tcllib/modules/doctools/doctoc_fmt.man doctoc_fmt]
    [manpage tcllib/modules/doctools/doctools_api.man doctools_api]
    [manpage tcllib/modules/doctools/doctools_fmt.man doctools_fmt]
...
[index_end]
```

Figure 4: sha1 manpage, converted to plain text

```
sha1 - sha1 hash
Generated from file 'tcllib/modules/sha1/sha1.man' by tcllib/doctools with format 'text'
sha1(n) 1.0.3  "sha1 hash"

NAME
====

sha1 - Perform sha1 hashing

SYNOPSIS
========

package require Tcl 8.2
package require sha1 ?1.0.3?

::sha1::sha1 msg
::sha1::hmac key text

DESCRIPTION
===========

This package provides commands to compute a SHA1 digests of arbitrary messages.

COMMANDS
========

    ::sha1::sha1 msg

        The command takes a message and returns the SHA1 digest of this message
        as a hexadecimal string.

    ::sha1::hmac key text

        The command takes a key string and a text and returns the hmac of the

EXAMPLES
========

| % sha1::sha1 "hello world"
| 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed

| % sha1::hmac "our little secret" "hello world"
| a7ed9d62819b9788e22171d9108a00c370104526

KEYWORDS
========

hashing, security, sha1
```

Another concept borrowed from that ancestor and TMML is that of *semantic markup*. This means that all markup defines *what* a component of the document is, but not its appearance. This is left to the conversion into an output format. The opposing term is *visual markup* which declares the appearance of the content without giving it structure. Examples for that are HTML, or naked TEX[2]. The use of semantic markup is important as it is this feature which allows the creation of tools which extract meta information from a document and use that to generate other documents, like an index of keywords. From the point of view of the system itself the extraction of meta data is actually nothing more than just another output format.

The full specifications of the three formats are part of the `doctools` module in `tcllib` and are not iterated here.

3. FORMATTING ENGINES

The implementation of the format processors is based upon the macro processor expand [20], combined with a plugin architecture for flexibility.

A more detailed picture of the internal architecture can be found in figure 5. This general setup is used for the processors of all three formats.

The most important feature is that various parts of the system are placed into their own interpreters, encapsulating them, making tampering difficult, allowing communication only through guarded APIs. The communication between the interpreters is done through command aliases.

At the beginning of the pipeline is an expander object. This reads the input, parses it into text and macros, and then executes the macros it found in the syntax checker. This interpreter knows the specification of the formatting language and checks the input for conformance.

As a secondary task it also directly executes some formatting commands which are independent of the output format, i.e `[include]` and `[vset]` (File inclusion and document variables). If the macro processor finds commands in the input which do not belong to the doc* formatting language they will currently also be handed to the syntax checker. This may allow input documents to wreak havoc with the syntax checking through the use of normal Tcl commands. In the future we may insert another interpreter between expander and checker to segregate the execution of such commands from the checking itself.

The syntax checker will hand all formatting commands conforming to the syntax and not handled by itself over to the last interpreter in the pipeline. This is the formatting engine. It contains and executes the code for the generation of the chosen output format, loaded into it during the initialization of a processor object for doc*. To prevent this (untrusted) code from inadvertent (or malicious) tampering with the environment is the main reason for the chosen multi-interpreter architecture. To further this goal a safe interpreter is used here.

When looking for a plugin implementing a format the system will first check if the specified name of the format is actually also the name of a file in the filesystem. If so, it will assume that this file contains the code for the formatter. Otherwise it will construct the name of a file from the name of the format and then search this file in a number of directories. The standard directories are setup so that the predefined formats are found, but any user of the system can extend this list according to her needs.

The main API commands which have to be implemented by any plugin for doctools for successful communication with the generic framework are listed in table 1. The remainder is documented in the manpages coming with `doctools` and will not be iterated here. The APIs for the other two languages are similar.

Beyond that the plugin is free in its activities, restrained only by the restrictions placed on the safe interpreter it is running in.

One of the most simplest plugins / output formats is `list`. Its code is shown in figure 6. This output format extracts just the meta data from the document and returns it for further processing by other tools.

4. INDICES AND TABLE OF CONTENTS

In chapter 2 it was said that while documents in the `doctoc` and `docidx` formats can be written manually their generation from documents written in `doctools` is more likely. In figure 7 we find a shell script showing the basics for doing so. This particular script is part of the `dtp` application and can be extracted from it by executing `dtp script`. In the future this may be converted into Tcl code and become a regular method of `dtp`.

The flow of information inside of the script is shown in figure 8. There are three important concepts buried in these figures.

For one we have the extraction of the metadata from the set of manpages, just another output format from the point of view of the engine. This format is actually a Tcl script by itself. See figure 9 for an example. Here we use it to generate the table of contents of the keyword index, first by generating them in `doc*`, then converting that into the final output format, here HTML.

The second concept has not been talked about before. All the references to other documents listed in either a table of contents or a keyword index are based on symbolic names and not physical ones, in conformance with the idea that these documents are independent of any output format. And embedding file names which are specific to some output format would destroy this property. This on the other hand means that a generator for a particular output format has to be told how to map from the symbolic to the actual file names. This is the second branch in the data flow, first creating the mapping and then using it anywhere a `doc*` document is transformed into the final output.

At last the script demonstrates the use of *engine parameters*. Any engine can export named parameters through which the user can influence its behaviour. In the case of the HTML engine we demonstrate there are mainly three, named *meta*, *header*, and *footer*. When set the engine will assume that their contents are raw HTML and inject them in specific places of the output. Here we use only the *header* to inject a navigation bar just before the actual contents of any document, be it table of contents, index, or manpage. Another parameter, *xref*, is not directly visible in figure 7, but internally used by `dtp` to pass in the information about command and keyword cross references, thus allowing the engine to properly create links which are between manpages, and links between manpages and the index.

At ActiveState a more fancy version of the shown script additionally inserts references to the company's stylesheet

[2]TEX without additional macros on top of the base language

Figure 5: Internal architecture of doctools

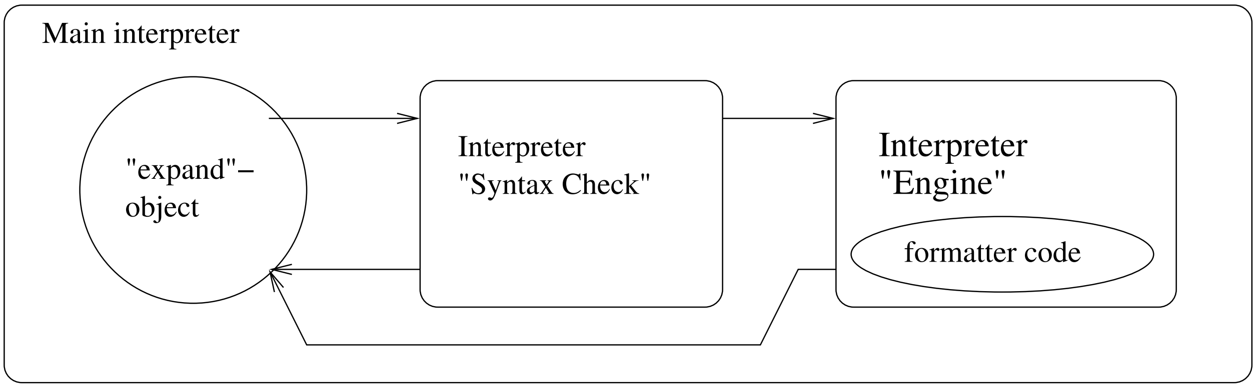

Table 1: Main plugin API

fmt_numpasses	This command is called immediately after the formatter is loaded and has to return the number of passes required by this formatter to process a manpage. This information has to be an integer number greater or equal to one.
fmt_initialize	This command is called at the beginning of every conversion run and is responsible for initializing the general state of the formatting engine.
fmt_setup *n*	This command is called at the beginning of each pass over the input and is given the id of the current pass as its first argument. It is responsible for setting up the internal state of the formatting for this particular pass.
fmt_postprocess *text*	This command is called immediately after the last pass, with the expansion result of that pass as argument, and can do any last-ditch modifications of the generated result. Its result will be the final result of the conversion. Most formats will use *identity* here.
fmt_shutdown	This command is called at the end of every conversion run and is responsible for cleaning up of all the state in the formatting engine.
fmt_plain_text *text*	This command is called for any plain text encountered by the processor in the input and can do any special processing required for plain text. Its result is the string written into the expansion. Most formats will use *identity* here.
fmt_*	Implementations of all the formatting commands as specified in the language specification, but prefixed with the string "fmt_". The sole exceptions to this are the formatting commands `vset` and `include`. These two commands are processed by the generic layer and will never be seen by the formatting engine.

Figure 6: List output

```
# -*- tcl -*-
#
# -- Extraction of basic meta information (title section version) from a manpage.
#
# Copyright (c) 2001-2002 Andreas Kupries <andreas_kupries@sourceforge.net>
# Copyright (c) 2003     Andreas Kupries <andreas_kupries@sourceforge.net>
#
################################################################

# Take the null format as a base and extend it a bit.
dt_source fmt.null

global    data
array set data {}

proc fmt_numpasses   {}     {return 1}
proc fmt_postprocess {text} {
    global data
    foreach key {seealso keywords} {
array set _ {}
foreach ref $data($key) {set _($ref) .}
set data($key) [array names _]
unset _
    }
    return [list manpage [array get data]]\n
}
proc fmt_plain_text  {text} {return ""}
proc fmt_setup       {n}    {return}

proc fmt_manpage_begin {title section version} {
    global data
    set    data(title)     $title
    set    data(section)   $section
    set    data(version)   $version
    set    data(file)      [dt_file]
    set    data(fid)       [dt_fileid]
    set    data(module)    [dt_module]
    set    data(desc)      ""
    set    data(shortdesc) ""
    set    data(keywords)  [list]
    set    data(seealso)   [list]
    return
}

proc fmt_moddesc   {desc} {global data ; set data(shortdesc) $desc}
proc fmt_titledesc {desc} {global data ; set data(desc)      $desc}
proc fmt_keywords  {args} {global data ; foreach ref $args {lappend data(keywords) $ref} ; return}
proc fmt_see_also  {args} {global data ; foreach ref $args {lappend data(seealso)  $ref} ; return}

################################################################
```

Figure 7: TOC and Index generation

```
#!/bin/sh
# Arguments: Package directory, Destination directory, optional label

src=$1 ; shift
dst=$1 ; shift
lbl=$1

if [ x$src = x -o x$dst = x ] ; then
    echo 1>&2 usage: $0 source destination
    exit 1
fi

if [ x$lbl = x ] ; then lbl=`basename $src` ; fi

rm -rf   $dst
mkdir -p $dst

echo Find and map sources ...
    dtp map -ext html -out $dst -trail 2 `find $src -type f -name '*.man' | sort` > $$.map
    echo _index_ $dst/index.html >> $$.map
    echo _toc_   $dst/toc.html   >> $$.map

echo Fixed nagivation bars ...
    dtp navbar $$.map _toc_   _toc_ 'Table Of Contents' /off   _index_ 'Index' /on   > $$.nb_toc
    dtp navbar $$.map _index_ _toc_ 'Table Of Contents' /on    _index_ 'Index' /off  > $$.nb_idx
    dtp navbar $$.map _index_ _toc_ 'Table Of Contents' /pass  _index_ 'Index' /pass > $$.nb_page
    # In the last command _index_ is a dummy, but has to be a valid symbolic filename.

echo Meta information ...
    dtp meta $$.map > $$.meta

echo Table Of Contents ...
    dtp toc \
        -title "$lbl -- Table of Contents" \
        -desc Modules \
        $$.meta \
        > $$.toc

    dtp gen-toc \
        -varfile header $$.nb_toc \
        html $$.map $$.toc > $dst/toc.html

echo Index ...
    dtp idx \
        -title $lbl \
        -desc "Keyword index" \
        $$.meta \
        > $$.idx
    dtp gen-idx \
        -varfile header $$.nb_idx \
        html $$.map $$.idx > $dst/index.html

echo Pages ...
    dtp gen-doc \
        -varfile header $$.nb_page \
        -subst   header _toc_   $dst/toc.html \
        -subst   header _index_ $dst/index.html \
        html $$.map $$.meta

rm $$.*
exit
```

Figure 8: Dataflow in gendoc.sh

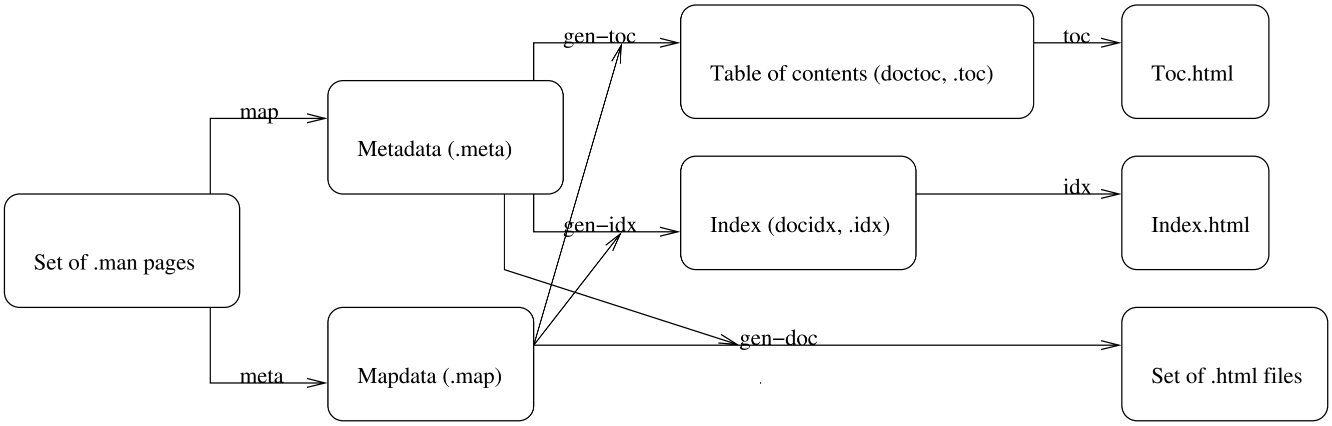

Figure 9: Excerpt of extracted meta data

```
manpage {
    desc       {Handle text in Emacs ChangeLog format}
    fid        changelog
    file       input/tcllib/modules/doctools/changelog.man
    keywords   {changelog emacs doctools}
    module     {}
    path       input/tcllib/modules/doctools/changelog.man
    section    n
    seealso    {}
    shortdesc  {Documentation tools}
    title      doctools::changelog
    version    0.1
}
manpage {
    desc       {Handle text in 'cvs log' format}
    fid        cvs
    file       input/tcllib/modules/doctools/cvs.man
    keywords   {cvs changelog {cvs log} log}
    module     {}
    path       input/tcllib/modules/doctools/cvs.man
    section    n
    seealso    {{}}
    shortdesc  {Documentation tools}
    title      doctools::cvs
    version    0.1
}
```

as a means of altering the visual appearance of the output without having to modify the engine, and copyright information.

The navigation bar itself also contains symbolic document references which are substituted with the correct reference while formatting each document. This is necessary because the relative position of the document containing the reference and the document referred to can change from document to document. The details of this are described in the command line help for `dtp`.

5. LIMITATIONS

Doctools and its formats are not big document processing systems, although they may come near with their ability to generate indices and tables of contents.

For example support for template text blocks and parameterization of such is limited to the very basics, i.e. the abilities to include a document into other documents and to set and get the value of document variables. There are neither conditionals nor looping constructs. Nor is it possible to define format specific text, i.e. it misses the ability to define different texts for web and print presentations.

The above, while possibly annoying in some situations, is not a very significant problem in my eyes. More serious to me is image support, or rather, the lack of it. While it is possible to use `[uri]` commands to insert a link to image into the text this will mean nothing to most output format, and in the case of HTML the result will be a regular `a` link to the image, and not an `img` tag. In other words the image will not be shown inline, but as clickable link to it.

There are some ideas floating around on how to remedy this, they will be discussed in chapter 7 about future work.

6. CONCLUSIONS

In this paper we have shown (yet another) language for writing documentation in, designed to fit into the Tcl world, with a tclish visual appearance, easy to write manually, also easy to process in pure Tcl, yet powerful and flexible enough to cover not only the most basics of needs of a documentation writer, but most of them.

We have shown the current implementation of the language, its internal organization, and especially its plugin mechanism which allows for a multitude of output formats and thus allows us to keep the system current with any changes in the environment which may and will come in the future. In line with the paradigm of Tcl as glue to anything else.

At last we have shown and discussed a tool based upon the doctools system and its actual use in a production environment, the build system for ActiveState's ActiveTcl and Tcl Dev Kit distributions [23].

With that we can now go on and try to take a look into the future and what we can and may do to enhance and extend the system even further.

7. FUTURE WORK

The limitations discussed in chapter 5 are obvious places where more work can be done in the future to enhance the doctools system.

With regard to image support for example I currently see two possible alternatives on implementing it. One method is simpler to implement in doctools, leaving the main part of the burden in the users hand. This method is implemented in the rendering system for TIPs [17]: Links to images are given in symbolic form, and the actual mapping from symbol to image data is then given to the format engine through means outside of the format itself. In the case of TIPs this is actually semi-automatic, each renderer uses the symbol as base path and then looks for a file with the proper file extension.

The alternative would be to extend doctools beyond text formatting by incorporating commands for the creation of vector and raster graphics [25]. This is more difficult to do, simply because all relevant formatting engines have to be extended to understand the input. And for those that don't or can't do graphics we either have to extend them so that the graphics commands do nothing, are the generic layer has to be able to detect if an engine is able to do graphics and do the skipping of graphics on its own. A question to consider before the implementation is: How far can we actually go with this?

A preliminary answer is: Quite far. In the XML world we have Scalable Vector Graphics, this can be combined with TMML. And in the *roff world we have the 'pic' language [8] for the same purpose[3]. Which means that it is possible to incorporate at least vector graphics into manpages. And that is actually the most we will most likely need. As for the TEX/LaTEX world, graphics can be incorporated either through PostScript, or a special sub-language for the description of vector graphics. In other words the same as proposed here.

A last note on this topic: The two alternatives discussed above do not exclude each other. They are actually quite orthogonal, one geared towards stronger vector graphics [4], the other more usable for raster images.

A less visible change would be to rewrite the interface between generic framework and formatting engines in an object-based manner. Actually this would be, for the sake of backward compatibility, rather the addition of an object-based interface to the system instead of replacing the existing interface completely. The point here is that the object-based approach makes a number of customization task a lot easier, by creating new engines by either derived them from an existing one, or by wrapping the custom code around it, i.e. delegation [19]. Even if this change is not done for the core system itself it is something writers of formatting engines should consider for their code.

Again more obvious work, extend the number of predefined formatting engines. Useful formats are, for example, DocBook [3] for connectivity to Linux documentation, Python's ReST (structured plain text, [5]), and Tcl's TIP format [17]. And beyond that an engine converting doctools into a format usable by Tk's text widget would allow for the portable display of our documents without the requiring additional extensions, like TkHTML [21].

And now looking at things from a completely different angle, consider the writing parsers which read formats like TMML, DocBook, ReST and return doctools. This way we can import existing documentation into the unified format. The most beneficial parser right now would be one for TMML. This is because Joe English already has a suite of

[3]Please note that this requires the use `troff`. `nroff` is not able to handle this.

[4]Encoding of raster images is also possible, see Richard Suchenwirth's strimj's [18]

applications which read *roff manpages and convert them into TMML, using a number of heuristics to guess the semantic markup. A TMML to doctools then simply completes the chain allowing the import of preexisting Unix manpages.

A simpler application would be to read doctools, and then write it back, in other words a pretty-printer.

The general theme the above is falling under is the (automatic, or semi-automatic) generation of documentation, using doctools as the generic intermediate language. In the discussion above the source is simply other existing documentation. A seed for this is already present in the doctools module, two packages parsing the output of `cvs log` and of emacs `ChangeLog` files, thus extending the generation of user-visible documentation from texts written by developers.

The next obvious step here is the extraction of documentation directly from the Tcl code, in other words, support for various types of *literate programming*. Examples of tools doing this are `AutoDoc` [9] and `zdoc` [10]. Inside of AutoDoc for example we have derivations of class `genericFormatter`, enabling it to write its result not only in HTML but any other format a class exists for. These parts could be rewritten to simply generate doctools and thus leverage of the formatting engines and tools already present for it. For a different approach in the same area see the `User Documentation Project` on the Wiki [1], which is thinking about using either doctools formatting directly for the embedded documentation or at least something very close it. Stepping further we come to tools which perform complex analyses on the source code of packages and application and have to report the results in some manner. An example of these class of applications is Source Navigator [22].

The last line of thought I wish to discuss here is the possibility to make the formatting language itself extensible. Look for example to the beginning of this chapter, where we talked about the support for images. One of the approaches discussed was to extend the language to support vector graphics. Is it possible to generalize such work ? I.e. to create a framework where language extensions itself can be plugged into the system, loaded on demand when used in a document ? Is this actually useful, or is graphics support the only important thing missing, language-wise ? This is also related to the topic of better support for templating and parameterized macros as described in chapter 5. For what else are such macros and templates than extensions of the formatting language itself ?

APPENDIX

A. REFERENCES

[1] User Documentation Project
`http://wiki.tcl.tk/8948`

[2] The Tcl Manpage Markup Language, Dean Richard Hipp, Joe English
`http://wiki.tcl.tk/TMML`
`http://tmml.sourceforge.net`

[3] The DocBook Format, Norman Walsh
`http://www.docbook.org/`
`http://wiki.tcl.tk/Docbook`

[4] Plain Old Documentation
`http://aspn.activestate.com/ASPN/Reference/`
`↪Products/ActivePerl/lib/Pod/`
`↪perlpod.html`
`http://aspn.activestate.com/ASPN/Reference/`
`↪Products/ActivePerl/lib/Pod/`
`↪perlpodspec.html`

[5] ReStructured Text
`http://docutils.sourceforge.net/`

[6] TEX and LATEX, Donald E. Knuth, Leslie Lamport
`http://www.tug.org/`
`http://www.latex-project.org/`

[7] HTML Specification
`http://www.w3.org/MarkUp/`

[8] nroff and companions (tbl, eqn, pic, ...)
`http://groff.ffii.org/`
`http://docs.rinet.ru:8083/UNIXi/ch08.htm`

[9] Autodoc; Extractor for documentation embedded in Tcl, Andreas Kupries
`http://www.purl.org/NET/akupries/soft/autodoc/`

[10] zdoc; Extractor for documentation embedded in Tcl
`http://www.oklin.com/zdoc/`

[11] Tcllib, Doctools module
`http://wiki.tcl.tk/doctools`
`http://tcllib.sourceforge.net/doc/#DIVd0e138`

[12] Tcllib, Doctools Processor
`http://wiki.tcl.tk/dtp`

[13] comp.lang.tcl
`news:comp.lang.tcl`

[14] XML Specification
`http://www.w3.org/XML/`

[15] TclXML, Steve Ball
`http://tclxml.sourceforge.net`

[16] TclDOM, Steve Ball
`http://tclxml.sourceforge.net`

[17] Tcl Improvement Proposals
`http://www.purl.org/tcl/tip/`
`http://www.purl.org/tcl/tip/3.html`
`http://sourceforge.net/projects/tiprender/`

[18] Strimj, Richard Suchenwirth
`http://wiki.tcl.tk/strimj`

[19] SNIT, Will Duquette
`http://wiki.tcl.tk/snit`

[20] Expand, Will Duquette
`http://wiki.tcl.tk/expand`

[21] TkHTML, Dean Richard Hipp
`http://wiki.tcl.tk/tkhtml`

[22] Source Navigator, RedHat
`http://wiki.tcl.tk/890`

[23] ActiveState's ActiveTcl, Tcl Dev Kit
`http://www.ActiveState.com/Tcl`

[24] Tcler's Wiki Format
`http://wiki.tcl.tk/Formatting%20Rules`

[25] TclMagick binding to ImageMagick
`http://wiki.tcl.tk/tcl-magick`