Technical documentation

Author: Johan Lissing

Version: 0.1

Date: 2005-12-01

Summary

This is the technical documentation for "Wazzup DOC?!", a Verilog documentation tool developed by PUM group 12 at Linköping University. The document describes how the product should be installed, maintained and extended.

Project identity

Project group

PUM 12 2005
Linköpings tekniska högskola
Institutionen för datavetenskap (IDA)

Project members

Name	Area of responsibility	Telephone	E-mail
Martin Jormedal	Project leader (PL)	073-3121319	ook4mi@gmail.com
Daniel Hilding	Customer relations (CR)	070-7440440	danhi139@student.liu.se
Joakim Svartengren	Documentation manager (DOC)	070-4040005	joasv190@student.liu.se
Jonas Norling	Design manager (DES)	070-3904809	norling@lysator.liu.se
Johan Lissing	Implementation manager (IM)	073-9036256	johli650@student.liu.se
Thobias Bergqvist	Quality manager (QM)	073-6223040	thobe651@student.liu.se
Eric Åberg	Test manager (TM)	070-4058130	eriab522@student.liu.se

Mailinglist for group

pum12@und.ida.liu.se

Web page

http://www-und.ida.liu.se/~pum12/

Customer

Per Karlström, ISY LiU

Customer contact person

Per Karlström, 013-28 29 03, perk@isy.liu.se

Project supervisor

David Broman, 013-28 57 24, davbr@ida.liu.se

Examiner

Robert Kaminski, 013-28 24 57, robka@ida.liu.se

Document History

Date	Version	Changes	Name
2005-12-01	0.1	Document created	Johan Lissing

1 Introduction

This chapter contains information about the disposition and contents of the technical documentation. It is intended to work as a guide for the reader, as well as anyone introducing modifications to this document.

1.1 Purpose of this document

The purpose of the technincal documentation is to ease future development of the product "Wazzup DOC?!", a documentation tool for Verilog, developed by PUM group 12. It should be an entry point in the search for information on how to modify or extend the product. The document contains a system overview, the shortcomings of the product, as well as suggestions for future extensions and improvements. The testing is also described here, so that it can be replicated and the test scripts can be reused.

The document is also the result of all research on Doxygen conducted during the implementation phase. It can therefore be useful to anyone who intends to add support for other languages to Doxygen.

Finally, the technical documentation contains directions on where to find and how to install the product.

The intented readers of this document are future developers and maintenance personnel.

1.2 Background

For a detailed project description and background, see the requirements specification [Hilding, 2005].

1.3 Document overview

This section describes the contents of each chapter in this document.

1.3.1 Introduction

Describes the disposition and contents of the technical documentation.

1.3.2 System overview

Gives an overview of the complete system.

1.3.3 ToDo list

Lists features that remain to be implemented.

1.3.4 Makeshift solutions

Describes any makeshift solutions or work-arounds in the software.

1.3.5 Extensions and improvements

Suggestions for further development.

1.3.6 Installation and development

How to install the product and develop it.

1.3.7 Testing

How the testing was done.

1.3.8 References

Lists all referenced resources. Text within square brackets refer to this section.

1.4 Reading instructions

As this is a highly technical document, all of it should be read for a full understanding. A future project group, striving to extend or modify the product, should also, at least briefly, read the architecture specification [Norling, 2005] and the design specification [Lissing, 2005].

1.5 Document dependencies

Changes in these documents might require changes in the technical documentation:

- Requirements specification [Hilding, 2005]
- Architecture specification [Norling, 2005]
- Design specification [Lissing, 2005]

1.6 Distribution

This document will be distributed to:

- Thomas Gustavsson and Angela Johansson, examiners of the technical documentation
- David Broman, project supervisor
- Per Karlström, customer
- The project locker

1.7 Glossary

AST - Abstract Syntax Tree. A tree describing the structure of a source file.

Flex - The GNU lexical analyzer (a variant of lex).

Lexical analyzer - see *scanner*.

Parser - A piece of software that determines the syntactic structure of a language.

Scanner - A piece of software that breaks down the input into word-like tokens.

Verilog - A hardware description language.

2 System overview

This chapter gives an overview of the system by first describing the functionality of the system and then briefly describe how each module works. There is one subsection for each module. The modules are defined in the architecture specification [Norling, 2005] and described in detail in the design specification [Lissing, 2005]. See section 6.4 "Code structure" to find the correspondence between a module and its source files.

2.1 System functionality

Doxygen is a tool for generating documentation from source files in various programming languages, such as C++ and Java. Constructs in the source files, like classes, functions and variables, are presented as hierarchical tables in the documentation. If the constructs are commented using Doxygen's special comment format, these comments also appear in the documentation.

The product "Wazzup DOC?!" is an extension to Doxygen. It allows Doxygen to generate documentation for Verilog source code and the constructs therein, such as modules, registers, wires and so on.

To generate documentation from source files, a user will simply have to execute the program in the directory with the source files. Subfolders will be created with the documentation in different formats, such as HTML and LaTeX. The user can control the documentation process with a configuration file, where various options can be selected, e.g. output format, documentation detail, etc. How to optimize the configuration file when documenting Verilog source files is described in section 6.3.2 "Generating documentation for a Verilog project" .

2.2 Preprocessor

The preprocessor for Verilog is implemented as small changes to the C++ preprocessor in Doxygen. It reads Verilog source files and evaluates compiler directives, such as macros and `ifndef-endif` statements.

2.3 Verilog front-end

The Verilog front-end module consists of the `VerilogLanguageScanner` class. Its code is generated by Flex from the file `verilogscanner.l`. This module scans one or several (preprocessed) Verilog source files and generates an AST, which is returned to the data organizer module. The AST nodes are instances of the Doxygen `Entry` class. Whenever an interesting construct is found a the Verilog file, a new node is created to hold information about it and its documentation.

2.3.1 Scanner rules

The scanner rules are described in [Lissing, 2005] on a Verilog keyword abstraction level. The associated C++ code fragments are used to create the AST nodes.

2.3.2 The Entry class

Instances of the Doxygen `Entry` class are used as tree nodes in the AST. An `Entry` instance can for example represent a Verilog module or wire. A documentation block regarding such a construct will be placed in the same `Entry` as the construct itself.

2.4 Tree printer

The tree printer was created only to test the Verilog front-end and is thus not necessary for documentation generation. It can, however, be useful when improving the scanner or when writing completely new scanners to add support for other languages to Doxygen.

The `print()` function in class `TreePrinter` takes a pointer to the root node of the AST as an argument, and prints its contents to a specified outputstream. The output of each tree node is indented according to the tree level of the node.

2.5 Data organizer

The data organizer module is the central part of Doxygen. It calls the Verilog front-end and receives the complete AST for the Verilog code. The AST data is the organized into lists and tables and passed to the output generator module. Various changes have been made to this module to introduce the Verilog concepts. The `ClassDef` and `MemberDef` classes have been modified to support Verilog constructs, such as `module` and `wire`.

2.6 Output generator

The output generator module generates the documentation from the lists gathered by the data organizer. Besides HTML and LaTeX, a few other output formats are also supported. This module is basically the same as the Doxygen equivalent. Only a few changes have been made to the English and Swedish translator classes, to make the output more tailored for Verilog documentation.

3 ToDo list

This chapter describes what remains to be done for the product to meet all the requirements in the requirements specification [Hilding, 2005].

3.1 Requirements not met

This section shows which requirements that remain to be met and gives hints on how they could be implemented.

3.1.1 F-8 Custom directives (extra level)

The custom directives specified in the architecture specification [Norling, 2005] are not fully functional. The ones already supported by Doxygen are working, but the Verilog-specific ones - `\clock`, `\reset`, `\comb`, `\state` and `\class` - are not. The functionality of the `\class` directive can be acheived using other standard Doxygen directives, such as `\ingroup`. The other unimplemented directives require connections to clocked or combinatorial nets (such as statements starting with the `always` keyword), which are currently just skipped by the Verilog front-end.

To implement these Verilog-specific custom directives, the comment scanner in `commentscan.l` will have to be extended. The scanner is called from the Verilog scanner (`verilogscanner.l`) whenever a documentation block is encountered. A documentation block may contain any number of these directives, or tags, which are caught by rules in the comment scanner.

The implementation also requires major changes in Doxygen's data organizer and output generator, to make them able to handle and store information about the clocked and combinatorial nets and documentation blocks for them. They will have to be added to the Entry tree node class and documentation sections for them will have to be added to the output generator classes.

Apparently, an implementation of requirement F-8 would be very time consuming and due to lack of implementation time it was not done in this project.

3.1.2 F-10 Windows compatible (extra level)

Doxygen claims to be Windows compatible, according to [van Heesch], although the installation is rather bulky. It requires many tools that are normally not included in Windows.

The changes and additions to Doxygen in the project should not affect Doxygen's Windows compatibility. However, a test of this would be very time consuming and require resources from the more important tests of basic and normal requirements. Therefore, requirement F-10 has not been tested and can not be considered met.

3.1.3 F-13 Verilog-2001 compatible parser (extra level)

The concepts in the latest Verilog specification, IEEE standard 1364-2001 or Verilog-2001, are not covered by the implementation of the Verilog front-end.

The reason for this is that the Verilog-2001 specification was discovered by the project group too late in the implementation phase. The customer did not explicitly express a wish for Verilog-2001 compliance during the definition phase and it was therefore not included in the requirements specification [Hilding, 2005] until later. It has also been hard to find a complete specification of all new concepts in Verilog-2001.

To make the Verilog front-end compatible with Verilog 2001, new scanner rules will have to be added to `verilogscanner.l`. New compiler directives will also have to be implemented in the preprocessor, `pre.l`.

Until requirement F-13 is met, two special directives can be placed around Verilog-2001 code to make the Verilog front-end ignore it. This way the code can still be documented, although the documentation may be incomplete. See section 4.2 "Skipping troublesome Verilog code" for a usage example.

3.2 Known errors and limitations

3.2.1 Synthesizable Verilog code required

The Verilog front-end module assumes that the Verilog code it scans is correct in the sense that it follows the IEEE1364 standard and may be synthesized. No specific tests are done to make sure that the code is correct and erroneous code may thus produce unexpected results.

3.2.2 Scope separator

The scope separator for generated Verilog documentation is currently the same as for C++, namely the double colon "::", which means that all internal signals will appear in the documentation as:

```
module_name::signal_name
```

The scope separator should be changed to something more Verilog-like, for instance a dot ".". The change can be made in the `MemberDef` class by checking the value of the OPTIMIZE_OUTPUT_FOR_VERILOG user option, similar to how the "[private]" annotation was removed, as described in section 4.1.

A solution to prevent the module name and the scope separator to be printed before the signal name without modifying the source code is to change the HIDE_SCOPE_NAMES setting in `Doxyfile`. See section 6.3.2.

4 Makeshift solutions

4.1 Use of Doxygen data types

The use of predefined data types in Doxygen has required some makeshift solutions. In the nodes of the AST, all Verilog ports (input, output, inout) are stored as public compound members in `Entry` instances, while signals and variables inside a Verilog module are stored as private compound members.

The documentation originally generated by Doxygen from this setup had some issues that needed to be resolved. For instance, after each private member the word "private" was printed in brackets, which made no sense in a Verilog documentation. This particular flaw was corrected in the `MemberDef` class by checking the value of the `OPTIMIZE_OUPUT_FOR_VERILOG` user option, as described in the design specification. All detected flaws in the documentation generation have been corrected this way to contain more Verilog-like text. It is a bulky solution, but Doxygen uses it similarly in a number of places, to tailor the documentation to suit different languages.

To avoid the need of pin-pointing each documentation flaw when extending the product in the future, the data types in Doxygen could be made more general and even more language environment testing could be used to tailor the output. Another option would be to have specific data types for each supported programming language.

4.2 Skipping troublesome Verilog code

The Verilog front-end has been tested to work correctly with Verilog code. However, as section 3.1 says, the scanner is not yet compatible with Verilog-2001 and some of the custom directives. To prevent the scanner from producing erroneous results when encountering incompatible Verilog code, the two directives `\startskip` and `\stopskip` can be used to skip certain portions of the code. In this way, a Verilog file containing a few incompatible statements can be documented, although the documentation may be incomplete. The example in figure 1 shows how to use `\startskip` and `\stopskip`.

```
...
wire a; //This will be documented as usual.

//! \startskip

//Multidimensional arrays are new in Verilog-2001.
//This will be ignored by the scanner.
reg [15:0] array [0:255][0:255];

//! \stopskip

reg [7:0] c; //This will be documented as usual.
...
```

Figur 1: An example of skipping code using special directives.

To eliminate the need for these directives, the scanner should be made compatible with the latest Verilog standard. See section 3.1.3 "F-13 Verilog-2001 compatible parser (extra level)" .

5 Extensions and improvements

This section contains suggestions on how the product can be extended and improved in the future. These suggestions can be used as requirements in a future project.

5.1 Extensions to the implementation

This section contains suggestions for extensions to the product. Other possible extensions are the requirements that were not implemented in this project, listed in section 3.1 "Requirements not met" .

5.1.1 Translation to other languages

The Verilog concepts introduced to Doxygen have been translated only to English and Swedish. To make them usable in other languages, all the other translator classes in Doxygen should be updated.

5.1.2 Graphical representation of instantiations

When a modules are instantiated from within other modules, a graphical representation of their connection could be presented in the documentation. The diagram could then show which ports are connected and possibly suggest a layout to minimize wiring distances.

5.1.3 Highlighted code in the documentation

Doxygen has a dedicated code scanner, `code.l`, for C++ to produce highlighted code that is included in the documentation. Keywords, classes and functions are all printed in different colors, making them easy to distinguish. This could be also be done for Verilog code, showing e.g. modules, ports and variables in different colors.

5.1.4 Verilog-2005 compliance

Apart from making the Verilog front-end compatible with the Verilog-2001 specification, as mentioned in section 3.1.3, it should also be made compatible with Verilog-2005, or IEEE 1364-2005, when this upcoming standard is finalized. The approach should be the same as with the Verilog-2001 compliance update.

5.2 Improvements to current implementation

This section suggests how the current implementation and its requirements can be refined. Other possible improvements are to fix the errors and limitations in the current implementation. These are listed in section 3.2 "Known errors and limitations" .

5.2.1 More thorough documentation

The documentation of Verilog source code could be made more thorough, to include more Verilog constructs, e.g. named sequence blocks, functions and user-defined primitives. These are currently just skipped by the Verilog front-end. At least functions should be relatively easy to document, mimicing how Doxygen documents C++ functions. Other Verilog concepts may

be harder to document and may require the introduction of new data types in Doxygen, as mentioned in section 4.1.

5.2.2 More custom directives

More custom directives could be made available to provide even more information in the documentation.

5.3 Extending and improving Doxygen

The suggested extensions and improvements given in the previous sections mainly apply to the "Wazzup DOC?!" Verilog documentation tool. Since the tool is based on and integrated to Doxygen, it relies on the capabilities of that program. Doxygen has its own ToDo/wishlist on its homepage, [van Heesch]. It includes requests for the support of more programming languages, more documentation formats, and much more.

6 Installation and development

This chapter describes the installation procedure for the product. It also gives an overview of the code structure. Finally, the prerequisites for further development are specified.

6.1 Installation prerequisites

This section lists the software that is required or optional to install and run the product. Note that the product has only been tested with the software versions mentioned here. The product is not guaranteed to be compatible with other versions of the software.

6.1.1 Required software

These software packages are required to install and run the product. The product may work with other versions of the software packages, but has only been tested with the versions given below. See also the Doxygen installation requirements at [van Heesch].

- A **UNIX**-like operating system. Sun Solaris and GNU/Linux are known to work.
- The **"Wazzup DOC?!"** product software: a fork of Doxygen v1.4.5.
- The GNU tools **flex** v2.5.4, **bison** v1.875d and **make** v3.80.
  Available from [FSF]. Newer versions of flex are known not to work.
- GNU **gcc** (the GNU compiler collection) v3.3.6.
  Available from [FSF].
- **Perl** v5.8.6.
  Available from [O'Reilly].

6.1.2 Optional software

This software is not necessary to install and run the product, but certain features may not be available without them.

- **Graphviz dot** v1.16
  Available from [Graphviz]. The dot tool of the Graphviz package is necessary to generate graphs in the documentation.
- **LaTeX**
  Available from [LaTeX3]. Necessary for LaTeX output.
- A **web browser**
  Necessary to view HTML output.
- A **Subversion client**, a version control system, if you have access to and want to check out the latest revision from the ISY Subversion server. Version 1.2.3 is known to work.
  Available from [Subversion].

6.2 Installation procedure

1. Install all the required software packages or check that they are available at your site. Installation instructions are available at the respective home pages if you want to install them manually. All the required software components are known to be available as packages for most GNU/Linux systems.

2. Extract the "Wazzup DOC?!" archive file using GNU tar...

```
$ tar xvzf hdldoctool-1.0.tar.gz
```

3. ...or check out the latest revision from the ISY Subversion server if it is available to you:

```
$ svn checkout https://svn.isy.liu.se/hdldoctool/
doxygen --username <your username>
```

4. Change to the newly created directory and run the configure script:

```
$ cd hdldoctool-1.0/
$ ./configure
```

5. Now run make to build the system:

```
$ make
```

6. The compiled binary file can be found in the `bin/` directory. You might want to copy it to where you like to keep executable files:

```
$ sudo cp bin/doxygen /usr/bin/doxygen-verilog
```

7. Now you are ready to run the tool, see the next section.

6.3 User's guide

As the tool is not ready for prime-time, it is merely a study of Doxygen's suitability as a base for a Verilog documentation tool, no proper user's guide will be written. This section lists and details the hops a developer or tester has to go through to generate a piece of documentation for a Verilog project, and may serve as a brief user's guide.

6.3.1 Prerequisites

1. Some experience of using Doxygen. Read the introductory chapters in Doxygen's manual, [van Heesch, 2005], and try it out on a C++ or Java project. Virtually all the configuration options and behaviour are the same in the extended version of Doxygen as in an original Doxygen installation. The product described in this document, "Wazzup Doc?!" has all the functionality of a regular Doxygen and can be used for documentation of for example C++ code.
2. A Verilog project to document: A directory with one or more Verilog (.v) files optionally organized in subdirectories. The Verilog code should adhere to the 1995 edition of the Verilog standard. Documentation should be placed in comments following the usual Doxygen practices, as described in section 5.2.2 of the architecture specification [Norling, 2005]. All directives described in Doxygen's manual may be used, e.g. for specifying the author, todo lists, etc.
3. A compiled version of the tool, as described in section 6.2. The executable file will hereafter be referred to as `doxygen-verilog`.

6.3.2 Generating documentation for a Verilog project

1. Let Doxygen generate a default configuration file (Doxyfile) for you:

```
$ doxygen-verilog -g
```

2. A few settings will have to be changed in the default configuration file:

Configuration directive and value	Explanation
OPTIMIZE_OUTPUT_FOR_VERILOG = YES	Causes names of concepts in the generated documentation to match Verilog lingo better.
EXTRACT_ALL = YES	Generate documentation for modules and nets that have no comments.
EXTRACT_PRIVATE = YES	Document wires, regs, parameters and so on.
HIDE_SCOPE_NAMES = YES	Don't show what module a wire is in.
RECURSIVE = YES	Set to YES if your project is divided into several directories.
HAVE_DOT = YES	Set to YES if you have Graphviz installed and want module hierarchy graphs to be generated.

3. Run Doxygen to generate the documentation:
```
$ doxygen-verilog
```
4. View the generated documentation by pointing a web browser to the `html` directory that was created for you, or print the latex documentation:
```
$ cd latex/
$ make ps
$ lp refman.ps
```

6.4 Code structure

The code structure has adopted the rather flat structure of Doxygen, since there are only a few completely new files for the actual Verilog language scanner. The rest of the product code is implemented as changes or additions in existing Doxygen files.

Doxygen's source tree is organized as follows:

Directory	Purpose
`doxygen/`	Contains top-level makefile and configuration script
`doxygen/src`	All source files are found here, flat structure
`doxygen/bin`	Generated executables are put here
`doxygen/lib`	Generated library files are put here
`doxygen/objects`	Compiled object files are put here
`doxygen/packages`	Automatically generated RPM package
`doxygen/qtools`	An old version of a subset of Qt, portability layer
`doxygen/tmake`	An old version of tmake, Qt's (old) build system
`doxygen/wintools`	

Directory	Purpose
`doxygen/libmd5`	MD5 message digest algorithm
`doxygen/libpng`	PNG image compression algorithm

For a list of the correspondances between modules and source files in `doxygen/src`, see the tables on page 27 and on.

6.5 Development prerequisites

If you have installed and run "Wazzup Doc?!" as described in the previous few sections, and have a solid understanding of the C++ language and the GNU Flex scanner generator, all you need to continue the development of "Wazzup Doc?!" is in place.

6.5.1 Checking out a revision from the Subversion server

To check out the latest revision from the ISY Subversion server:

```
$ svn checkout https://svn.isy.liu.se/hdldoctool/
doxygen --username <your username>
```

If you want a specific revision, use the `-r <rev#>` flag to svn.

6.5.2 Build system

The build system is based on make and Qt's tmake. If you need to add new source or header files, they should be added to `libdoxygen.pro`. The makefiles will be updated to include the new files when `make` is run.

Initially, the `configure` script and `make` needs to be run in `doxygen/`. To recompile the project, just run make in `doxygen/` or `doxygen/src/`. This will create the archive (library) file `libdoxygen.a` and link it together with a `main()` routine to create the doxygen executable.

6.5.3 Checking in changes

Please refer to Subversion's reference manual, at [Subversion], for a description of how to add and check in files to the repository.

7 Testing

7.1 General testing information

The testing has been divided into 4 major sections: unit testing, module testing, integration testing and system testing. The planning and the results of these tests are available in the test plan [Åberg, 2005:1] and test report [Åberg, 2005:2], respectively.

The test personnel have mainly been TM, DOC, CRM and QM. TM has been in charge of the planning and preperation of the testing, while DOC and CRM have been writing the test skripts. QM has mainly done the static code inspection due to his programming experience.

7.2 Test documents

The existing test documents are:

- Test plan [Åberg, 2005:1]
- Test report [Åberg, 2005:2]

These documents will be delivered on the CD together with the program. They will be placed in `/test/documents/`.

7.3 Test scripts

The test scripts used during the test phase will be placed on the CD under `/test/testscript/`.

7.3.1 Treeprinter

Compile the test script together with Doxygen and the Treeprinter. The Treeprinter is called when the test script is run.

7.3.2 Verilog scanner

Compile `parse-and-print.cpp` together with Doxygen and the Verilog scanner. Parse-and-print takes a Verilog file as an argument and uses the Tree printer to visualize it.

7.3.3 Preprocessor

Compile `preprocess.cpp` together with Doxygen and the preprocessor. Preprocess takes a Verilog file as an argument and produces another Verilog file without any preprocessing syntaxes.

8 References

8.1 Internal documents

- [Hilding, 2005] Hilding, Daniel, "Requirements specification" (2005)
- [Lissing, 2005] Lissing, Johan, "Design specification" (2005)
- [Norling, 2005] Norling, Jonas, "Architecture specification" (2005)
- [Åberg, 2005:1] Åberg, Eric, "Test plan" (2005)
- [Åberg, 2005:1] Åberg, Eric, "Test report" (2005)

8.2 External documents

- [FSF] Free Software Foundation, "Free Software Directory" (2005) WWW: `http://directory.fsf.org/`
- [Graphviz] AT&T, "Graphviz - Graph Visualization Software" (2005) WWW: http://www.graphviz.org
- [LaTeX3] LaTeX Project "LaTeX - a document preparation system" (2005) WWW: `http://www.latex-project.org/`
- [O'Reilly] O'Reilly Media Inc., "The Source for Perl" (2005) WWW: `http://www.perl.com`
- [Subversion] CollabNet, "Subversion" (2005) WWW: http://subversion.tigris.org
- [van Heesch, 2005] van Heesch, Dimitri, "Doxygen" (2005), software version 1.4.4, WWW: `http://www.doxygen.org`

Appendix A Source files

The following tables show which of the most important source files that belong to which module. The status of each file is classified as new (written from scratch), changed (modified Doxygen files) or unchanged (original Doxygen files). Files belonging to Doxygen modules irrelevant to the "Wazzup DOC?!" product, such as front-ends for other programming languages or simple helper classes, are omitted.

A.1 Preprocessor

File name	Status
`define.cpp`	Unchanged
`define.h`	Unchanged
`pre.cpp`	**Changed** (generated from `pre.l`)
`pre.h`	Unchanged
`pre.l`	**Changed**

Table 9: The source files belonging to the preprocessor module.

A.2 Verilog front-end

File name	Status
`commentscan.cpp`	Unchanged
`commentscan.h`	Unchanged
`commentscan.l`	Unchanged
`debug.cpp`	**Changed**
`debug.h`	**Changed**
`entry.cpp`	Unchanged
`entry.h`	**Changed**
`parserintf.h`	Unchanged
`verilogscanner.cpp`	**New** (generated from `verilogscanner.l`)
`verilogscanner.h`	**New**
`verilogscanner.l`	**New**

Table 10: The source files belonging to the Verilog front-end module.

A.3 Tree printer

File name	Status
`treeprinter.h`	**New**
`treeprinter.cpp`	**New**

Table 11: The source files belonging to the Tree printer module.

A.4 Data organizer

File name	Status
`classdef.cpp`	**Changed**
`classdef.h`	**Changed**
`config.cpp`	**Changed** (generated from `config.l`)
`config.h`	Unchanged
`config.l`	**Changed**
`defgen.cpp`	Unchanged
`defgen.h`	Unchanged
`definition.cpp`	Unchanged
`definition.h`	Unchanged
`dirdef.cpp`	Unchanged
`dirdef.h`	Unchanged
`doxygen.cpp`	**Changed**
`doxygen.h`	Unchanged
`filedef.cpp`	Unchanged
`filedef.h`	Unchanged
`groupdef.cpp`	Unchanged
`groupdef.h`	Unchanged
`main.cpp`	Unchanged
`memberdef.cpp`	**Changed**
`memberdef.h`	Unchanged
`membergroup.cpp`	Unchanged
`membergroup.h`	Unchanged
`memberlist.cpp`	Unchanged
`memberlist.h`	Unchanged
`pagedef.cpp`	Unchanged
`pagedef.h`	Unchanged

Table 12: The source files belonging to the data organizer module.

A.5 Output generator

File name	Status
docparser.cpp	Unchanged
docparser.h	Unchanged
doctokenizer.cpp	Unchanged
doctokenizer.h	Unchanged
doctokenizer.l	Unchanged
doxygen_css.h	Unchanged
doxygen.css	Unchanged
htmlgen.cpp	Unchanged
htmlgen.h	Unchanged
index.cpp	Unchanged
index.h	Unchanged
latexgen.cpp	Unchanged
latexgen.h	Unchanged
outputgen.cpp	Unchanged
outputgen.h	Unchanged
outputlist.cpp	Unchanged
outputlist.h	Unchanged
translator_adapter.h	Unchanged
translator_br.h	Unchanged
translator_ca.h	Unchanged
translator_cn.h	Unchanged
translator_cz.h	Unchanged
translator_de.h	Unchanged
translator_dk.h	Unchanged
translator_en.h	**Changed**
translator_es.h	Unchanged
translator_fi.h	Unchanged
translator_fr.h	Unchanged
translator_gr.h	Unchanged
translator_hr.h	Unchanged
translator_hu.h	Unchanged
translator_id.h	Unchanged
translator_it.h	Unchanged

Table 13: The source files belonging to the output generator module.

File name	Status
`translator_je.h`	Unchanged
`translator_jp.h`	Unchanged
`translator_ke.h`	Unchanged
`translator_kr.h`	Unchanged
`translator_lt.h`	Unchanged
`translator_nl.h`	Unchanged
`translator_no.h`	Unchanged
`translator_pl.h`	Unchanged
`translator_pt.h`	Unchanged
`translator_ro.h`	Unchanged
`translator_ru.h`	Unchanged
`translator_se.h`	**Changed**
`translator_si.h`	Unchanged
`translator_sk.h`	Unchanged
`translator_sr.h`	Unchanged
`translator_tw.h`	Unchanged
`translator_ua.h`	Unchanged
`translator_za.h`	Unchanged
`translator.cpp`	Unchanged
`translator.h`	Unchanged

Table 13: The source files belonging to the output generator module.