Architecture specification

Author: Jonas Norling

Version: 0.2

Date: 2005-09-30

Abstract

This document is a description of the architecture of the Verilog documentation tool that is to be developed for the department of electrical engineering at Linköping University. As such, it is an extension to the requirements specification [Hilding, 2005] detailing the architecture of a system implementing the given requirements.

The architecture of the system is described, as well as the design decisions that have been made in order to arrive at this architecture.

The intended readers of this document are the developers of the system as well as the customer.

Project identity

Project group

Wuzzup DOC
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 (CRM)	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, Computer Engineering, 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
2005-09-23	0.1	Document created	Jonas Norling
2005-09-30	0.2	Document revised after internal inspection. Corrected some spelling errors and added a reference chapter.	Johan Lissing

1 Introduction

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

1.1 Purpose of this document

This architecture specification in intended to serve as an overview of the system that is to be constructed. In this document the architecture of the system is described, as well as the design decisions that have been made in order to arrive at this architecture.

This document describes what is needed to implement the requirements in the requirement specification. A more detailed and careful design will be made in the design specification, using the architecture described in this document.

1.2 Document overview

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

1.2.1 Introduction

Describes the disposition and contents of the architecture specification.

1.2.2 System overview

Gives a background of the project and an overview of the system to be developed.

1.2.3 Design decisions

Presents important design decisions affecting the architecture.

1.2.4 Architectural description

Contains a description of the architecture of the Doxygen and the Verilog documentation tool.

1.2.5 Module interfaces

Describes the interfaces between the modules in the system.

1.2.6 File formats

Lists and describes the file formats that will be used by the Verilog documentation tool.

1.2.7 Code libraries and components

This chapter gives a list and an introduction to components that will or might be used in the product.

1.2.8 Design guidelines

This chapter contains guidelines and recommendations for implementing the system.

1.2.9 Connection to requirements

Maps the requirements from the requirements specification to the parts of the architecture that implement them.

1.3 Related documents

In this document, many references are made to the requirements specification [Hilding 2005]. The following things can be found in that document, apart from the basic requirements of the system: a brief description of the project, a use case, product components to be shipped and tests for each requirement.

1.4 Reading instructions

To get a grasp of what the system is expected to do, read chapter 2 "System overview" . For an understanding of the proposed architecture, read chapter 2 to chapter 5. The part on design decisions can be safely skipped. In order to get a grip on how to implement the system, reading the whole document is recommended. You are not expected to have read the requirements specification, but it is recommended for a full understanding of this document.

For evaluation of traceability and the connection to the requirements specification, chapter 9 is recommended reading along with the referenced sections.

1.5 Document dependencies

Changes in these documents might result in changes in the architecture specification:

- Requirements specification [Hilding, 2005]

Changes in this architecture specification might result in changes in the following documents:

- Project plan [Jormedal, 2005]
- Design specification [Norling, 2005]
- Technical documentation [Lissing, 2005]
- Test plan [Åberg, 2005]

1.6 Distribution

This document should be distributed to:

- Hans Olsén and Johan Fagerström, examiners of the architecture specification.
- Per Karlström, the customer.
- The project folder.

1.7 Glossary

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

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

Lexical analyzer -- (lexer or scanner) A piece of software that breaks down the input into word-like tokens.

Bison -- The GNU parser generator (a variant of YACC). See [Bison].

Flex -- The GNU lexical analyzer generator (a variant of lex). See [Flex].

2 System overview

This chapter gives a background of the project and an overview of the system to be developed. See the requirements specification [Hilding, 2005] for further details.

2.1 Background

Verilog [Verilog] is a common hardware design language (HDL) used by electrical engineers worldwide and specifically by the customer. Structurally, it resembles normal computer programming languages but naturally it has some special features and data structures, such as modules, wires and so on.

A documentation tool is a program that automatically creates documentation from source code files. The documentation gives the reader a good overview by showing relations between classes and brief descriptions of them.

While there are several documentation tools for the most popular computer programming languages, such as C++ and Java, there is no tool capable of documenting Verilog code.

2.2 Goal of project

This project will probably not have sufficient resources to produce a complete Verilog documentation tool. The goal with this project is therefore to make a foundation for a documentation tool and to investigate whether this could be done using existing tools for other languages, e.g. Doxygen [Doxygen].

All findings, research and/or source code, will be extensively documented for future projects on the same topic.

2.3 Users and environment

Our product is not meant to be a complete and usable tool, but merely a foundation for such. The final product, should it ever be developed, is targeted at employees at the department of electrical engineering, Linköping University, with programming experience. The program will be run at their computer system, which is mainly UNIX and GNU/Linux.

Use cases detailing the intended uses of the system can be found in the requirements specification.

2.4 Properties of the system

The focus of the developed system is on extendability and documentation of the implementation. Completeness, usability, user interface and performance are not considered very important.

3 Design decisions

In this chapter, important design decisions that have been made during the work on the architecture are presented.

3.1 Platform

Early on in the negotiations with the customer, Doxygen was introduced as the preferred platform to build the Verilog documentation tool upon. Some time has been invested in investigating Doxygen's suitability as a base for this project, resulting in the decision to make this product an extension to said program. This decision represents a great risk, as we are very dependent on Doxygen's flexibility and extendability, but is in line with the customer's intentions and goal of the project.

Doxygen runs on all major platforms (of which UNIX and Windows is of interest in this project) due to the use of portable code and Qt as a portability layer. The code written in the scope of this project should use the same mechanisms in order to be portable.

See [Doxygen] for further details.

3.2 Languages and tools

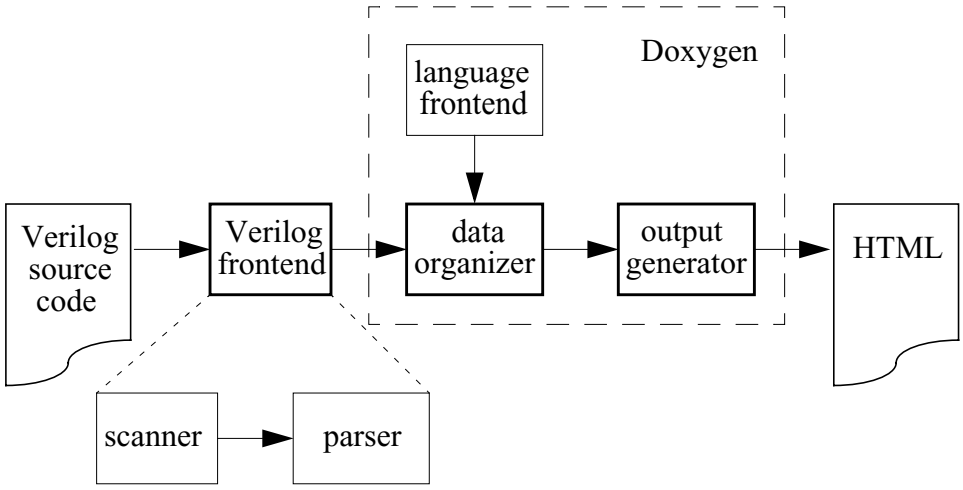
The decision to use Doxygen as a base for the product implies using C++ as a programming language. This is also the wish of the customer and is given as a requirement in the requirements specification. Flex (the GNU lexical analyser generator) and possibly Bison (the GNU parser generator) will be used to generate C/C++ code for the Verilog parser.

See [Flex] and [Bison] for more information.

4 Architectural description

The system will be realised as an extension to and modification of Doxygen. This means that many design and architectural decisions already have been made for us.

This chapter describes Doxygen's architecture and the changes that will be made to it. An overview of this architecture is presented in figure 1.



Figur 1: An overview of the proposed architecture. Doxygen is modified and extended to be able to parse and generate documentation from Verilog files. The main modules in the system is the Verilog front-end (containing a scanner/lexical analyzer and a parser), the data organizer and the output generator.

4.1 Doxygen's architecture

Doxygen is roughly composed of a language front-end (parser), a data organizer and a number of output generators. The parts have the following roles:

- Language front-end: This module is used to parse source code files in a multitude of languages such as C, C++, Java and IDL. It is implemented as a monolithic lexical analyzer and parser in flex. An abstract syntax tree (AST) is generated from each source file.
- Data organizer: This is a loosely coupled set of functions used to extract hierarchies from the AST. Lists of files, functions, classes, namespaces etc are extracted in this module.
- Output generators: This is a number of classes used to generate output in formats such as HTML, LaTeX, man files and so on.
- The program flow is driven by Doxygen's main function performing a few basic steps: reading the configuration, parsing of input files and generation of output files. Output generation is mainly driven by the nodes in the AST.

4.2 Our architecture, an extension to Doxygen

For each module, these changes and extensions are covered by this project:

4.2.1 Verilog front-end

Input: A Verilog file.

Output: A syntax tree (AST).

This is a new module replacing or working alongside Doxygen's language frontend. It implements Doxygen's abstract parser class.

The language front-end will be written in C++, Flex (lexical analyzer) and possibly Bison (parser). It will output an abstract syntax tree of instances of class Entry (which is a part of Doxygen). Apart from parsing the Verilog code, documenting comments have to be extracted and passed to Doxygen's comment parser.

For testing and verification purposes the Verilog front-end will be written as a self-sustaining module, making it easy to write a test framework that will output the generated AST in a human-readable form.

4.2.2 Data organizer

Input: A tree of Entry (AST).

Output: A number of trees.

This is a part of Doxygen. It generates trees of files, classes, namespaces and so on from the extracted AST. This module will have to be extended to handle new types of structures, such as instantiation of modules in Verilog. The following structures (at least) will be extracted from the AST:

- Source files (already in Doxygen)
- Module declarations (will need an extension to Doxygen)
- In/out signals in modules (will need an extension to Doxygen)
- Processes (will need an extension to Doxygen)

4.2.3 Output generator

The output generator for HTML should work with our extensions, possibly after minor modifications.

The interfaces between the modules are described in more detail in the next chapter.

4.3 Alternative architectures

4.3.1 Filter to Doxygen

Doxygen support for some languages have been implemented as a filter executed before Doxygen that translates the source language to (pseudo-) C++ or Java code. This method was rejected because it was thought to prove inflexible and a mapping to C++ and Java concepts could be hard to find.

4.3.2 Own architecture (no Doxygen involved)

The documentation tool does not necessarily have to be an extension to an existing tool. The question whether to use Doxygen as a base for the Verilog documentation tool led to lengthy discussions and investigations during the early weeks of the project. The decision to use Doxygen was made because of the so much greater potential for getting a useful Verilog documentation tool even though the risk of running into problems related to Doxygen cannot be neglected.

4.4 Testability

The Verilog parser is written as a self-containing module so that module and regression tests can be performed. We expect to be able to use Doxygen's XML and/or HTML output for manual inspection of parsing results.

5 Module interfaces

This chapter describes the interfaces between the different modules in the architecture.

5.1 Verilog source code - Verilog parser

The Verilog source code file may contain custom directives in the code comments. These directives should be handled by the Verilog parser to fulfill requirement F-8 in the requirements specification [Hilding, 2005]. The following directives will be implemented to fulfill the requirement:

- \auth {author_name}
  The author of the source code.
- \date {date}
  The date when the source code was last updated.
- \bug {bug_list}
  Known bugs in the source code.
- \clock {signal_name}
  Specifies which signal is used as clock signal.
- \reset {signal_name}
  Specifies which signal is used as reset signal.
- \comb
  Indicates that the following code block is purely combinatorial.
- \state {state_alias}
  An alias for a state in a state machine.
- \class {class_name signal_list}
  Denotes that all signals in the signal list belong to the same class.

5.2 Verilog parser - Data organizer

The Verilog parser will construct an AST of the processed Verilog source code. This AST will constitute the parser's output interface and it has two, possibly different, specifications.

As stated in requirement F-2 in the requirements specification [Hilding, 2005], a basic requirement is to implement an arbitrary data type for the AST.

If possible, the AST will be represented in a way that it can be understood by Doxygen's data organizer. This is stated in requirement F-5 in the requirements specification [Hilding, 2005].

The Verilog parser will be a C++ class inheriting from the abstract parser class in Doxygen (class ParserInterface). The parser will need to implement the parseInput() method, taking a buffer with source code as argument, returning an AST, possibly constructed by Entry nodes.

To make Doxygen's data organizer compatible with Verilog, some modifications to the data organizer will probably be necessary. For instance, the introduction of Verilog data types, such as "module" and "wire", or the custom directives listed in the previous section.

5.3 Data organizer - Output generator

The data organizer is not a module per se, but more like a losely connected set of Doxygen functions. When the input files are parsed, the AST is found in a global variable. Information is extracted from this AST in a number of steps and put in a collection of global lists.

The output generation step takes the data generated by the data organizer (a number of global lists) and iterates through it calling the output generator classes for each datum to be output. The interface to the output generation classes are extensive, but we expect no need to modify them.

6 File formats

This chapter lists the file formats used in the project. The file extension is given in the heading and then a description of that file format follows.

6.1 .V - Verilog source file

Verilog source files are plain text files whose source code syntax follow the IEEE standards 1364 and P1800 [IEEE].

One of the key concepts in Verilog is the module, which reads its input parameters from its signal inputs, performs some kind of function and then writes the result to its signal outputs. The module's function may be implemented using logic primitives, bit operations, other modules, and so on.

Verilog files are commented using either the short comment (//) or the long comment (/* and */) format. Both have the same functionality as their C++ equivalents. The comments may contain anything without interfering with the source code. Therefore, we can use the comment blocks to introduce special directives. See section 5.1 "Verilog source code - Verilog parser" .

For more information, and an extensive BNF notation, see [Verilog].

6.2 .HTML - HyperText Markup Language

HTML is a popular and versatile file format for presenting practically any kind of information. The files consist of text blocks surrounded by tags that describe how to format the text. For more information on HTML, see [W3C].

7 Code libraries and components

7.1 Doxygen

The source code documentation tool Doxygen will be the main component of the product. It is written in C++ and flex and is heavily used for documentation of C++ code as well as a number of other languages. The official homepage can be found at [Doxygen].

7.2 The Icarus parser

Icarus is a free software Verilog simulation and synthesis tool. We hope to be able to use the parser from Icarus as a part of our project or as a reference. The official homepage can be found at [Icarus].

8 Design guidelines

This chapter contains guidelines for the design and particularly the design specification [Norling, 2005].

8.1 AST format

If the AST is constructed using arbitrary data types, according to section 5.2 "Verilog parser - Data organizer" , these data types should be as similar as possible to the data types Doxygen uses. This ensures a smooth transition to a fully Doxygen compatible AST either later in this project or in future projects.

9 Connection to requirements

In this section, each requirement from the requirements specification [Hilding, 2005] is listed along with a reference to the part of the architecture to which it can be traced.

Some requirements can't be connected to a single part of the architecture. For these requirements a short description of how they affect the architecture is given.

9.1 Basic functional requirements

Requement	Point of implementation
F-1 Parser for verilog code	section 4.2.1 "Verilog front-end"
F-2 Create a syntax tree	section 4.2.1 "Verilog front-end"
F-3 Print syntax tree	section 4.2.1 "Verilog front-end"
F-4 Linux compatible	section 3.1 "Platform"

9.2 Normal functional requirements

Requement	Point of implementation
F-5 Doxygen compatible entry tree	section 4.2.1 "Verilog front-end"
F-6 Extract hierarchy	section 4.2.2 "Data organizer"
F-7 Extract comments	section 4.2.1 "Verilog front-end"

9.3 Extra functional requirements

Requement	Point of implementation
F-8 Custom directives	section 5.1 "Verilog source code - Verilog parser"
F-9 Generate HTML	section 4.2.3 "Output generator"
F-10 Windows compatible	section 3.1 "Platform"
F-11 Other outputs	section 4.2.3 "Output generator"

9.4 Basic non-functional requirements

Requement	Point of implementation
N-1 Programming language	section 3.2 "Languages and tools"
N-2 Documented with doxygen	-
N-3 Upgradeable	-
N-4 English usage	-
N-5 Documentation format	-
N-6 Revision control	-

A number of these requirements can't be connected to a specific part of the architecture and some of them are not applicable to the architecture at all:

N-2: This requirement only affects the actual implementation.

N-3: Upgradeability has been kept in mind during the architecture work, but can't be attributed to one single part of the architecture.

N-4: This is not applicable to the architecture. All documents are written in english, and all documentation will be.

N-5: This requirement states that documents are to be written in FrameMaker format and is as such not applicable to the architecture.

N-6: All code should be put in a Subversion repository. Not applicable to the architecture.

10 References

10.1 Internal documents

- [Jormedal, 2005] Jormedal, Martin, "Project plan" (2005)
- [Hilding, 2005] Hilding, Daniel, "Requirements specification" (2005)
- [Norling, 2005] Norling, Jonas, "Design specification" (2005)
- [Lissing, 2005] Lissing, Johan, "Technical documentation" (2005)
- [Åberg, 2005] Åberg, Eric, "Test plan" (2005)

10.2 External documents

- [Bison] Free Software Foundation,
  WWW: http://www.gnu.org/software/bison/
- [Flex] Free Software Foundation,
  WWW: http://www.gnu.org/software/flex/
- [Verilog] McNamara, Michael, WWW: http://www.verilog.com/
- [Doxygen] van Heesch, Dimitri, WWW: http://www.doxygen.org/
- [IEEE] IEEE Standards Association,
  WWW: http://standards.ieee.org/
- [W3C] World Wide Web Consortium,
  WWW: http://www.w3.org/MarkUp/
- [Icarus] Williams, Stephen,
  WWW: http://www.icarus.com/eda/verilog/