Design specification

Author: Johan Lissing

Version: 1.1

Date: 2005-11-27

Summary

This document specifies the design to be used for the Verilog documentation tool "Wazzup DOC?!", developed by PUM group 12. The design is intended to ease and control the implementation of the product.

The product will be realized as an extension to Doxygen, an existing documentation tool for C++, Java, and a few other programming languages. Therefore, a major part of this document is dedicated to describing how Doxygen is designed.

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
2005-11-02	0.1	Document created	Johan Lissing
2005-11-03	0.2	Document updated after comments. Corrected a few spelling errors.	Johan Lissing
2005-11-04	1.0	Document updated after inspection. Extended section 4.1.1 and section 5.6. Added section 5.3.3. Corrected a few spelling errors.	Johan Lissing
2005-11-27	1.1	Document updated after research during implementation. Revised section 3.4. Extended section 5. Added Appendix A.	Johan Lissing

1 Introduction

This chapter contains information about the disposition and contents of the design specification. 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 design specification is intended to ease and control the implementation of "Wazzup DOC?!", a documentation tool for Verilog, developed by PUM group 12. The document is based on the architecture specification [Norling, 2005] but will describe the design on a more detailed level. This will ensure a functioning and well-structured implementation.

As this project is part implementation, part research, some details necessary for a full implementation of the documentation tool may be omitted. However, all pre-implementation research findings will be thoroughly described in this document, to give any future project groups a head start for expanding the product to a fully functional Verilog documentation tool.

The intented readers of the design specification are the implementation team members, both in this project group and any future groups working on the same project.

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 design specification.

1.3.2 Design philosophy

Describes how the design was created and how research was done.

1.3.3 Design decisions

Presents decisions and alternate solutions to design issues.

1.3.4 Critical parts

Identifies the critical parts of the design.

1.3.5 Detailed description

An extensive description of the entire design with diagrams and detailed information.

1.3.6 Reuse

Lists re-used code from other software.

1.3.7 User interface

Describes the user interface of the product.

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. For an implementation team, sections 4, 5 and 6 are of particular interest. A future project group, striving to extend or modify the product, should read sections 2 and 3 carefully.

1.5 Document dependencies

Changes in these documents might require changes in the design specification:

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

Changes in the design specification might require changes in the following documents:

- Project plan [Jormedal, 2005]
- Architecture specification [Norling, 2005]
- Technical documentation [Lissing, 2005:2]
- Test plan [Åberg, 2005]

1.6 Distribution

This document will be distributed to:

- Johan Fagerström and Jonas Wallgren, examiners of the design specification
- 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.

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

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

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.

UML - Unified Modelling Language. A standard to model the structure and behavior of a design. See [OMG].

2 Design philosophy

This chapter describes the philosophy of the design. The design is highly affected by the fact that the project will, to a large extent, consist of research.

2.1 Doxygen research

As the product will be both an extension and a modification to Doxygen, a lot of research on Doxygen will have to be done. A basic investigation has already been conducted, which came to the conclusion that Doxygen was suitable as a product foundation.

A more in-depth research of Doxygen was made for this design specification. The approach is described in section 2.2. One of the main goals has been to see how all necessary modifications to Doxygen can be made with minimum changes in its own design. This would make our product more suitable as a patch, rather than a complete rewrite.

Some issues remain to be resolved for the implementation. This research will be done alongside with the implementation phase.

2.2 Prototype testing

To find out how Doxygen should be modified and extended to support Verilog source code, small prototype tests have been made to examine different parts of Doxygen. The results were used to make important design decisions. The result of each prototype test is described together with the corresponding design decision in section 3.

The tests also lead to knowledge of Doxygen's design which was necessary in order to write the detailed description in section 5.

Some tests made it clear that even more testing and research is necessary in a certain parts of the system. Due to limited resources, this research was not done in time for this design specification, but will be conducted during the implementation phase. Such issues may be critical for the system and they are therefore listed in section 4 "Critical parts" .

3 Design decisions

This chapter describes the decisions made in the design. Most of the decisions are based on the results of the prototype tests, in accordance with our design philosophy in section 2.2. For an illustration of how these decisions are realised in the design, see section 5.

Decisions made on an architectural level are listed in the architecture specification [Norling, 2005]. All decisions regarding the implementation, such as code layout and commenting, are made in the programming handbook [Lissing, 2005:1].

3.1 Separate scanner file

For most supported languages, Doxygen uses one big scanner file. For languages that are similar to each other, such as Java and C++, this may be a practical approach. Since Verilog is so different from these, we would have to do a lot of conditional testing in the file to make sure that only Verilog rules were applied to Verilog source code. This would make the scanner file even larger and it would make the installation of our product difficult, with many small changes to this file. We wanted to know if it is possible to write the Verilog scanner as a separate file to avoid the aforementioned problems.

The prototype test showed that this is indeed possible. The Flex generated scanner class should inherit the `ParserInterface` class and implement the `parseInput()` method. The class is then registered as a parser in the `initDoxygen()` method in `doxygen.cpp`.

Decision: We will use a separate scanner file for Verilog, implemented with Flex.

The design of the scanner in the Verilog front-end is described further in section 5.3.1.

3.2 Scanner+parser layout

In order to construct an AST representing Verilog source code, tree nodes have to be created and linked into the tree for each syntactic construct of the source language. This can be done either directly with C++ code in the Flex rules in the scanner, or by first passing the scanned data (a stream of tokens) to a parser, which then constructs the nodes.

All language front-ends in Doxygen are implemented purely as scanners without a separate parser component. For a sense of conformity, we would like the Verilog front-end to be implemented the same way. Another advantage is the possible reuse of code from the already implemented scanners, as mentioned in section 6.2.

A prototype test indicated that the scanner-only approach should work well for the Verilog front-end.

Decision: The AST will be constructed with C++ code in the scanner rules. A parser generator (such as Bison) will not be used. This design decision is closely linked to the decision in section 3.1 "Separate scanner file" .

The design of the Verilog front-end is described further in section 5.3.

3.3 Node class

We must also estimate if we directly can use Doxygen's `Entry` class for the AST nodes, or if we should start by implementing our own simplified node class. The two options correspond to requirements F-5 and F-2, respectively, in [Hilding, 2005]. Using an arbitrary node class might save time in the scanner implementation. A big drawback, however, is that all operations on the AST, such as generating output, must be adapted to the new node class.

Testing showed that the `Entry` class is rather generic, but will have to be adapted to be able to represent all constructs in the Verilog language. However, we estimate that these changes should not be too extensive. See section 5.

Decision: The `Entry` class provided by Doxygen will be used.

The design of the node class and modifications to the Entry class are further described in section 5.

3.4 Tree printer

To be able to print the AST, in conformance with requirement F-3 in [Hilding, 2005], we must determine how to construct the tree printer module and find the most suitable place to call it.

We found that the best place to print the AST is after all input files have been read by the scanner, and hence all AST nodes are constructed. In Doxygen, this corresponds to the `parseFiles()` function, which has a `while` loop iterating over the input files. When the loop is finished, the AST is complete. However, to avoid too much involvement from Doxygen and make the setup as small as possible, we decided not to integrate the tree printer into Doxygen. Instead, a test script will be written that first scans the source code and then instantiates and calls the tree printer.

To be able to convert the tree data into a readable form, the tree printer module will be implemented as the class `TreePrinter` with the appropriate functions.

Decision: The tree printer module will be implemented as a class, which will be called from a stand-alone test script.

The design of the tree printer module is further described in section 5.3.5.

4 Critical parts

This section identifies critical parts in the design, which we have not been able to test and/or design. They have been considered when writing this document and must also be kept in mind when implementing the design.

4.1 Doxygen

Doxygen itself is one big critical part. It is a rather large program written externally, which means that it is hard to fully understand all parts of its source code.

4.1.1 Output generation

As stated in section 5, the `Entry` class must be modified to be able to handle all concepts in the Verilog language, such as the module construct. However, the output generators must also be able to handle the corresponding sections correctly. For example, there should be a module hierarchy instead of a class hierarchy in documentation for Verilog code.

The necessary modifications to the output generator classes and functions in Doxygen have not been fully investigated due to lack of resources. The output generator module will be subject to further research during the implementation phase and in the meantime this module is listed as a critical part. On a side note, the level of the output requirements (F-9 and F-11 in [Hilding, 2005]) is "Extra", and hence the module may not be implemented at all.

4.2 Verilog

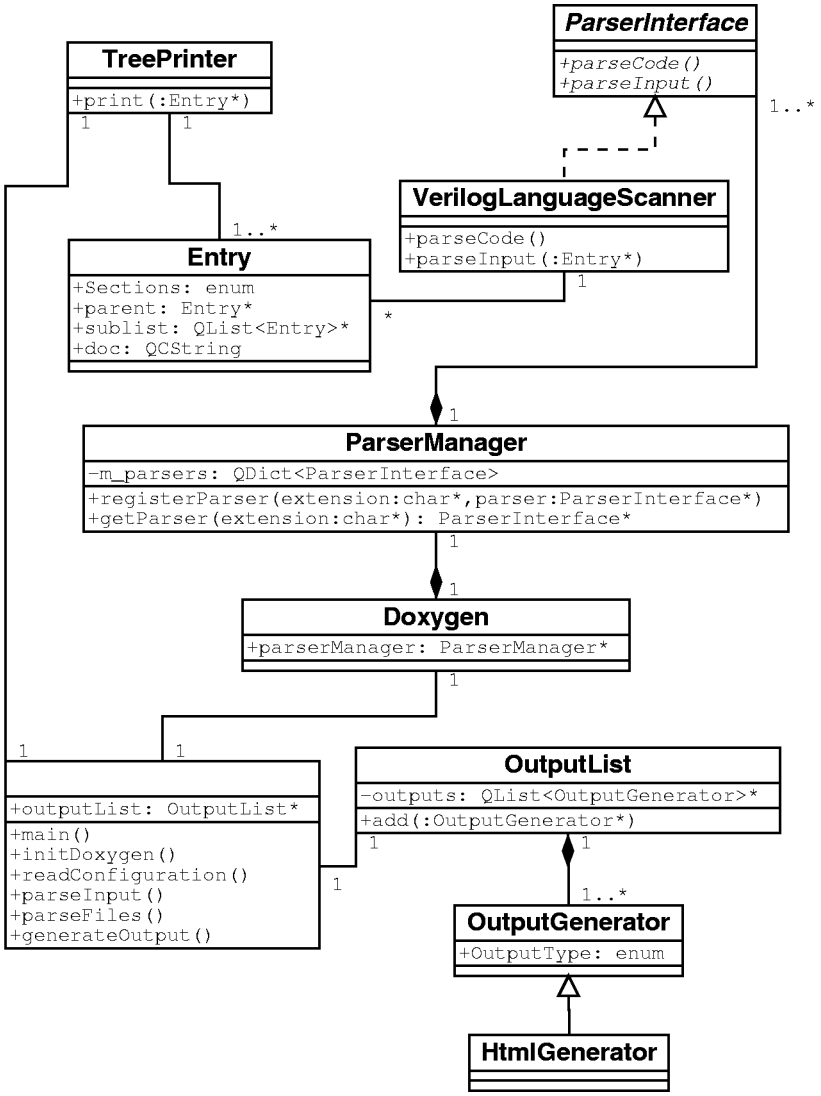
It may be difficult to parse all concepts in the Verilog language. Since none of the project members is a Verilog expert, further Verilog studies may be required.

5 Detailed description

This chapter describes the design in detail. The first subsection gives an overview of the system. Then, the internals of each module are described. The modules are defined in [Norling, 2005].

5.1 System overview

A class diagram for the most important classes, attributes and operations is shown in figure 1. These are the classes that will affect or will be affected by the addition of Verilog documentation capabilities in Doxygen. All classes belong to Doxygen's original source code, except for `VerilogLanguageScanner` and `TreePrinter`. These correspond to the Verilog front-end module, described in section 5.3, and the tree printer module, described in section 5.3.5. They will have to be written from scratch.



Figur 1: System overview class diagram

The notation for the class diagram is standardized UML and can be found at [OMG].

5.2 The preprocessor module

Before the contents of the Verilog source is scanned, any compiler directives in the code must be preprocessed. The compiler directives can, for example, be used to define macros and to only activate certain parts of the Verilog code based on conditional testing. All compiler directives in Verilog start with a grave accent, also known as "backtick", followed by a command. This is in contrast to C++ compiler directives, which all start with the number sign #, also known as hash mark.

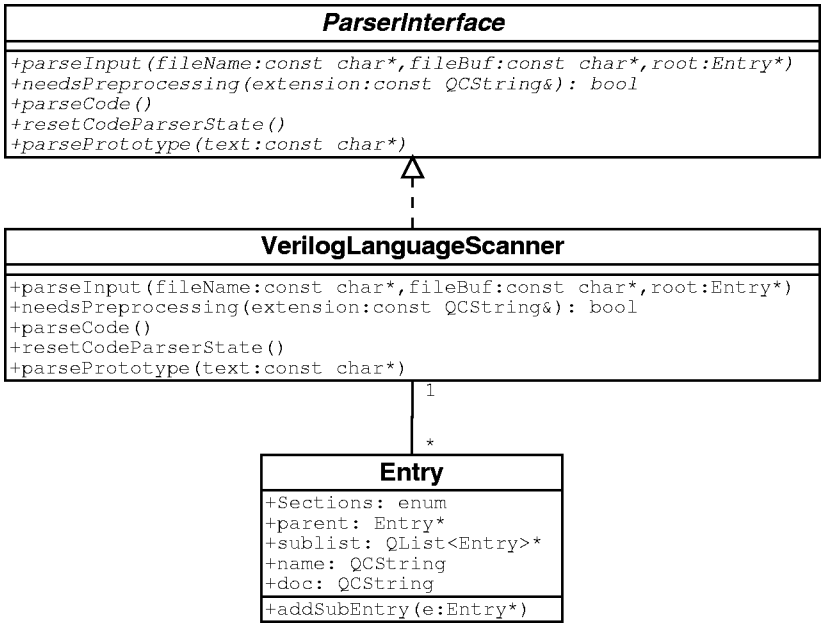
Doxygen is equipped with a preprocessor for C++ code. It is written as a Flex scanner residing in `pre.l`, and is started with a call to its `preprocessFile()` function. Since most of the Verilog compiler directives have C++ equivalents, we will use the Doxygen preprocessor to preprocess Verilog code. A few rules and pattern will have to be added to the scanner.

5.2.1 Affected classes and members

Member	Description
-	New scanner rules and actions for Verilog compiler directives

Table 9: Affected members in the preprocessor module in `pre.{h,l,cpp}`.

5.3 Verilog front-end module



Figur 2: UML diagram of the classes in the front-end module. `ParserInterface` is an abstract class defining the interface that is used by all language parsers in Doxygen. Note that the `Entry` class is simplified, since it has a large number of data members that are of no interest to our project.

The Verilog front-end module consists of the `VerilogLanguageScanner` class, as shown in figure 2. This class contains one important method, `parseInput()`, as well as some auxiliary functionality.

The `parseInput()` method, taking a buffer with source file data and a pointer to the root of a syntax tree, calls a Flex generated scanner to do the actual scanning and parsing.

The Verilog front-end module is defined in `verilogscanner.l` which is translated by Flex into the C++ source file `verilogscanner.cpp`. The affected files, classes and members are shown in section 5.3.5 "Affected classes and members" on page 24.

5.3.1 A Verilog parser in Flex

The lexical analysis step will be implemented in Flex and the actual parsing (building of a syntax tree) is done with embedded C++ code and ample use of start conditions (scanner states) in Flex. See the Flex homepage [FSF] for an introduction to GNU Flex, and the dragon book [Aho, et al] for an introduction to lex and a description of lexical analyzers and parsers in general.

Suitable start conditions have been extracted from the Verilog BNF found at [Satterlee]. The start conditions and the transitions between them are shown in appendix A.2 and some less important start conditions are shown in appendix A.3. The words that trigger the transitions are written on the edges. Some transitions may be triggered by several words and these words are grouped into patterns. An explanation of these is given in appendix A.4.

The lexer/parser combination works as follows (imagine parsing a simple Verilog file with a single module declaration containing a few wires and instantiations of other modules):

1. The lexer is initialized in start condition **Search**. In this start condition, the token `module`, indicating the start of a module declaration, triggers a transition to ModuleDec.
2. In the **ModuleDec** start condition, an identifier defining the name of the module is expected. A syntax tree node for the module definition is prepared. The next state is ModuleArgs.
3. An argument list for a module is optional. In the **ModuleArgs** start condition, either a left parenthesis or a semicolon is expected (after whitespace is removed), indicating either the start of an argument list or the end of the declaration.
4. If a left parenthesis is found, the **ArgList** state is entered. Here, a list of identifiers is added to the module declaration tree node. When a right parenthesis is found, the argument list has ended and the **ModuleArgs** state is entered once again.
5. When the module declaration has been terminated by a semicolon, the **ModuleBody** start condition is entered. In this state, the lexer looks for all Verilog constructs that are valid in a module, such as wire definitions and module instantiations.
6. When wire definitions or module instantiations are found, the **NetDec** and **Instantiation** states are entered, respectively. When the statements are finished, e.g. with a terminating semicolon, the **ModuleBody** start condition is entered once again.
7. Upon occurance of the keyword `endmodule`, the module body is terminated and the **Search** start condition is entered once again.

The start conditions in appendix A.3 will be used for skipping irrelevant code blocks that are terminated in a non-trivial way, e.g. not necessarily with a semicolon.

From any of the start conditions described here, there may also be transitions via global scanner rules to documentation start conditions when documentation blocks are encountered. This is covered in the next section. There is also a pair of global rules for skipping entire sections of the code between two special markers. This can be useful as a temporary solution if the scanner produces erroneous results from certain types of code.

5.3.2 Handling comments and documentation blocks

A description of Verilog source code and documenting comments och more introductory nature can be found in the architecture specification [Norling, 2005].

The comment formats recognized in the source code are:

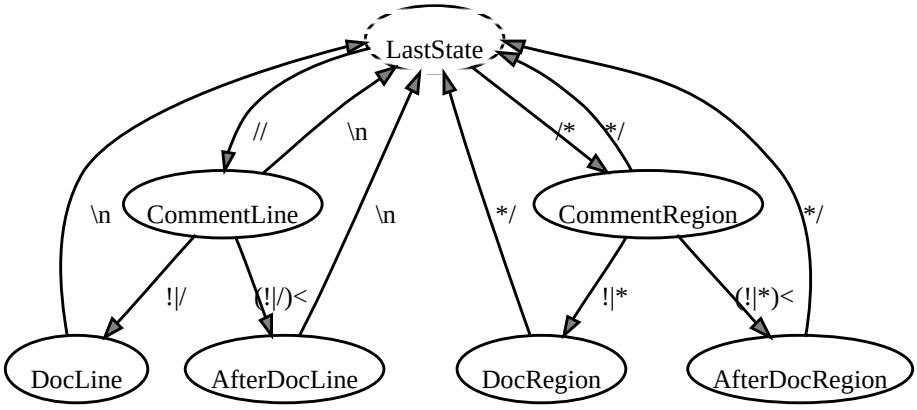
`// ...`	C style comment. Terminated at line end.
`/* ... */`	C++ style comment.
`/// ...`	Documentation line. Terminated at line end.
`//! ...`	Documentation line. Terminated at line end.
`/** ... */`	Documentation block.
`/*! ... */`	Documentation block.

The documentation found in such documentation blocks pertain to the construct that follows them. To indicate that a documentation block belongs to the preceding construct, the documentation is prefixed with the '<' symbol. This is useful when adding documentation for wire declarations as this example shows:

```
wire readStrobe; ///< This is a wire
```

Comments in the Verilog source code are caught by global rules (i.e. rules that are valid in all start conditions). The start conditions **CommentLine** and **CommentRegion** are entered using the `yy_push_state()` function meaning that the previous start condition is put on a *start condition stack* before the new one is entered. The old start condition can then be restored by calling `yy_pop_state()`. The start conditions that are used for handling comments simply skip all input up to the end of the comment.

If the comment is discovered to be a documenting comment, control is handed over to one of a set of special start conditions that extract the documentation and add it to the corresponding tree node. Documentation text is handed over to `handleCommentBlock()`, a global function in the scanner file, which calls `parseCommentBlock()`, as described in section 5.3.3. A diagram with details of the involved states and transitions is shown in figure 3.



Figur 3: Start conditions involved in comment and documentation parsing. ***LastState*** *is the state that was active when the comment was encountered, it is pushed onto the start condition stack. The edges show the symbols that trigger transitions to other states, in lex syntax (i.e. the vertical bar represents an OR-function and the parenthesis are grouping operators). The actual documentation is extracted in the four bottom states. When all is finished, LastState is popped and entered.*

5.3.3 Handling custom directives

The comment lines and regions described in the previous section may contain custom directives, as defined in [Norling, 2005]. The whole documentation block, together with the current `Entry` node, is handed to `parseCommentBlock()` in the dedicated comment scanner. This scanner is defined in `commentscan.l`, and searches for directives in the block.

When a directive is found, the matching section in the current `Entry` node is updated with the directive information.

Some of the directives in [Norling, 2005], such as `\author`, are already supported by Doxygen, since they are applicable to several languages. However, the scanner will have to be extended with rules to support the new, Verilog-specific, directives.

The following directives will be implemented:

- `\author` - already in Doxygen
- `\date` - already in Doxygen
- `\bug` - already in Doxygen
- `\clock` - new scanner rule needed
- `\reset` - new scanner rule needed
- `\comb` - new scanner rule needed
- `\state` - new scanner rule needed
- `\class` - scanner rules for the Doxygen group concept will be used or adapted

5.3.4 Constructing the AST

As the scanner described in section 5.3.1 works its way through the source it has a pointer, `current`, to the `Entry` node it is currently entering data in. As soon as the node data is complete, e.g. after a wire declaration has

terminated, a new `Entry` node is added to the AST and the `current` pointer is redirected to the new node.

5.3.5 Affected classes and members

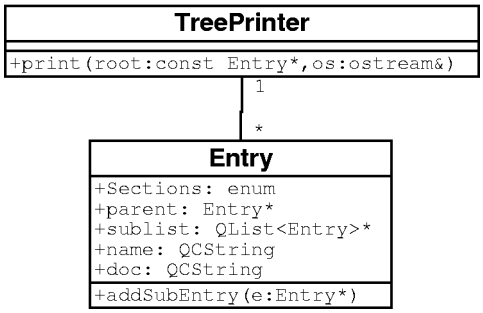
Member	Description
-	New scanner rules
`parseInput()`	Main parse function
`needsPreprocessing()`	Unused inherited function
`parseCode()`	Unused inherited function
`parsePrototype()`	Unused inherited function
`resetCodeParserState()`	Unused inherited function

Table 10: Affected members in the `VerilogLanguageScanner` class in `verilogscanner.{h,l,cpp}`.

Member	Description
`Sections`	New enumeration values for Verilog module section and Verilog module documentation section

Table 11: Affected members in the `Entry` class in `entry.{h,cpp}`.

5.4 Tree printer module



Figur 4: UML diagram of the `TreePrinter` class that is used to print a human-readable representation of a syntax tree.

The tree printer module is implemented as a stand-alone class, `TreePrinter` (see figure 4). Apart from the constructor, it has one public function, `print()`, which is called when all Verilog source files have been scanned by the Verilog front-end. The `print()` function takes a pointer to the root node (class `Entry`) of the AST and an `ostream` (output stream) reference. The AST is traversed, and each node is printed to the `ostream`. A typical call is

```
TreePrinter tp();
tp.print(std::cout, rootPtr);
```

where `rootPtr` has the type `Entry*`.

5.4.1 Affected classes and members

Member	Description
`print()`	Main print function

Table 12: Affected members in the `TreePrinter` class `treeprinter.{h,cpp}`.

5.5 Data organizer module

The data organizer module is responsible for the program flow in Doxygen. In the current version of Doxygen, the following steps are taken to parse a set of input files and generate documentation from them:

1. All global structures and variables are initialized. Parser modules are registered here.
2. The configuration file is parsed.
3. Input files are parsed in `parseInput()` and `parseFiles()`, leaving a tree of Entry nodes for the subsequent steps.
4. The `Entry` tree is traversed several times in search for all declarations of classes, namespaces and so on. This step leaves a number of dictionaries (lists) of `ClassDef` instances and similar objects.
5. The documentation is written. This step is driven by `generateOutput()` which calls a number of functions for writing documentation for a number of constructs. The actual documentation strings are generated by `ClassDef` and its sibling classes.

To represent the Verilog constructs that are necessary to generate useful documentation, some changes and extensions will be made to the data organizer module, as described in the following subsections.

5.5.1 The module concept

Verilog's module concept is represented as a new `Sections` enumeration value, `MODULE_SEC,` in the `Entry` class (`entry.{h,cpp}`) and a new `CompoundType` enumeration value is added to `ClassDef` (`classdef.{h,cpp}`). Support functions are updated to handle conversions between them.

5.5.2 Instantiations of modules

Requirement F-7 (at the extra level) calls for a graph (presented graphically or otherwise) of module instantiations in the system described by the Verilog source code. To support this, a class representing a module instantiation, `InstantiationInfo`, is defined. A list of such objects is held in an `Entry` and a `ClassDef`.

5.5.3 The configuration file parser

The "writing part" of the configuration file parser needs to be extended. See section 5.6.1.

5.5.4 Affected classes and members

Member	Description
`CompoundType`	New enumeration value for Verilog module section
`compoundTypeString()`	New string for Verilog module
`qualifiedNameWithTemplateParameters()`	New scope separator string for Verilog

Table 13: Affected members in the `ClassDef` class in `classdef.{h,cpp}`.

Member	Description
`writeDocumentation()`	Prevent Doxygen from writing "[private]" after Verilog variables

Table 14: Affected members in the `MemberDef` class in `memberdef.{h,cpp}`.

Member	Description
`convertToCompoundType()`	Add conversion between Verilog module section and compundtype
`initDoxygen()`	Register Verilog scanner

Table 15: Affected members in `doxygen.cpp`.

Member	Description
`check()`	Add Verilog source file extension
`create()`	Add Verilog output optimization option

Table 16: Affected members in the `Config` class in `config.{h,l,cpp}`.

5.6 Output generator module

Output generation in Doxygen is initiated by the `generateOutput()` function. A list with `OutputGenerator`s, one for each output format, is traversed and called for each construct to be output. How each construct (e.g. a class) is output, is defined in the corresponding construct definition class (e.g. `ClassDef`, which defines a compound, such as a class or namespace). Hence, we must add knowledge of the module concept to `ClassDef`. The Verilog data types wire and register may utilize the `MemberDef` class, since it allows generic data types.

In order to add a section in the output data for module instantiations, methods will have to be added to the output generator classes as well as the translation interface. The details of the necessary modifications will be investigated at a later date. Therefore, this module is included in the Critical parts chapter, section 4.1.1 "Output generation" .

5.6.1 Configuration for Verilog documentation

The configuration file, `Doxyfile`, is created when Doxygen is run with the `-g` flag. It allows the user to set numerous options regarding which files and which language constructs to document and much more. Two of the configuration options are the boolean items `OPTIMIZE_OUTPUT_FOR_C` and `OPTIMIZE_OUTPUT_FOR_JAVA`. When on of them is set to `YES`, the output documentation will be tailored to the selected language. We may use a similar approach with an `OPTIMIZE_OUTPUT_FOR_VERILOG` option, to be able to make the documentation more suitable for Verilog. This option, and an explanatory information text, will be added to the configuration file parser. The parser, `config.l`, is used to both read and write configuration file entries.

5.6.2 Fine-tuning of documentation

Most headings in the documentation are not affected by the addition of the new module compound type. The headings must be changed to reflect the Verilog language. For example, the heading "Member list" should be changed to "Signal list". The strings used in the headings are fetched from functions in the active translator class, based on the users selection of language. At least the english translator, translator_en.h, should be extended with alternative text strings suitable for Verilog documentation. The configuration setting `OPTIMIZE_OUTPUT_FOR_VERILOG` will be used to determine which string a function should return.

5.6.3 Affected classes and members

Member	Description
`trMemberDataDocumentation()`	Adapt return string to Verilog
`trListOfAllMembers()`	Adapt return string to Verilog
`trMemberList()`	Adapt return string to Verilog
`trThisIsTheListOfAllMembers()`	Adapt return string to Verilog
`trIncludingInheritedMembers()`	Adapt return string to Verilog
`trCompoundList()`	Adapt return string to Verilog
`trCompoundMembers()`	Adapt return string to Verilog
`trCompoundListDescription()`	Adapt return string to Verilog
`trCompoundMembersDescription()`	Adapt return string to Verilog
`trCompoundIndex()`	Adapt return string to Verilog
`trClassDocumentation()`	Adapt return string to Verilog
`trCompounds()`	Adapt return string to Verilog
`trCompoundReference()`	Adapt return string to Verilog
`trGeneratedFromFiles()`	Adapt return string to Verilog
`trPublicAttribs()`	Adapt return string to Verilog
`trClasses()`	Adapt return string to Verilog
`trClass()`	Adapt return string to Verilog

Table 17: Affected members in the `TranslatorEnglish` and `TranslatorSwedish` classes in `translator_en.h` and `translator_se.h`, respectively.

5.7 Compiling and linking

The changes and additions to Doxygen will be introduced into the source tree of the latest version of Doxygen available when the development is started. Doxygen's build system, using a combination of make and tmake (Trolltech's predecessor to qmake), will be used as it is. The build system realizes, in essence, the following make rules:

- Run Flex on `*.l`, generating C++ files.
- Compile `*.cpp`.
- Put all object files in `libdoxygen.a`, except for `main.o`.
- Link `libdoxygen.a`, `main.o` and a few auxiliary libraries into an executable.

This building scheme is quite flexible and allows for linking the modules we are constructing to module testbenches.

6 Reuse

This section lists software elements that can be reused when implementing the product.

6.1 Doxygen

Naturally, a large portion of Doxygen can and will be reused for the product. The data organizer and output generator modules will almost entirely consist of unmodified Doxygen code. The necessary modifications are described in the corresponding subsections of section 5 "Detailed description" .

6.2 Verilog front-end

The syntax rules of the scanner in the Verilog front-end module will have to be written from scratch. However, the associated C++ code to create AST nodes can be adopted from other scanners used in Doxygen. The Python scanner in Doxygen is, just as our Verilog scanner will be, implemented as a stand-alone scanner file, which makes it a good role model.

7 User interface

This chapter describes the user interface of the product.

7.1 User interface of the product

The product is an extension to Doxygen, an existing software. The extensions and modifications needed for Verilog support do neither require nor justify modifications to Doxygen's user interface. For more information on Doxygen's user interface, see [van Heesch, 2005]. Use cases are described in the requirements specification [Hilding, 2005].

8 References

8.1 Internal documents

- [Hilding, 2005] Hilding, Daniel, "Requirements specification" (2005)
- [Jormedal, 2005] Jormedal, Martin, "Project plan" (2005)
- [Lissing, 2005:1] Lissing, Johan, "Programming handbook" (2005)
- [Lissing, 2005:2] Lissing, Johan, "Technical documentation" (2005)
- [Norling, 2005] Norling, Jonas, "Architecture specification" (2005)
- [Åberg, 2005] Åberg, Eric, "Test plan" (2005)

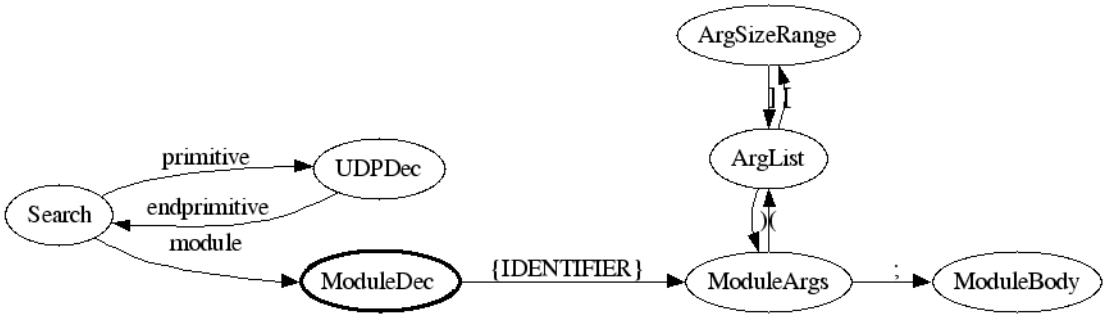
8.2 External documents

- [Aho, et al] Aho, Sethi, Ullman, "Compilers: principles, techniques and tools" (1986, 2003)
- [FSF] Free Software Foundation, "Flex GNU Project" (2005)
  WWW: `http://www.gnu.org/software/flex`
- [OMG] Object Management Group, "UML" (2005),
  WWW: `http://www.uml.org`
- [Satterlee] Satterlee, Chris, "Verilog Formal Syntax Specification" (1995)
  WWW: `http://www.verilog.com/VerilogBNF.html`
- [van Heesch, 2005] van Heesch, Dimitri, "Doxygen" (2005), software version 1.4.4, WWW: `http://www.doxygen.org`

Appendix A Verilog front-end scanner

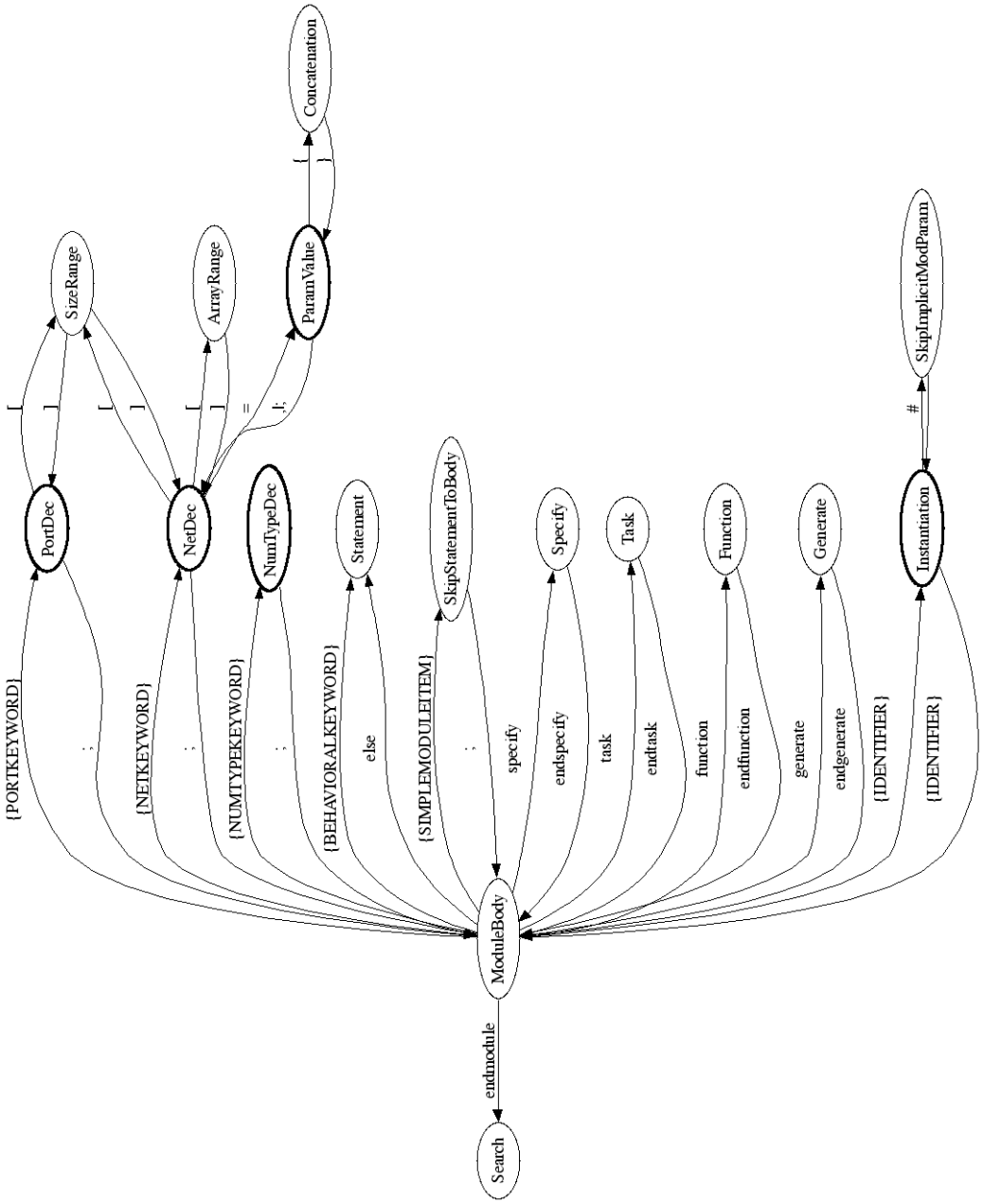
This appendix contains graphs that illustrate the tranistions between start conditions in the scanner of the Verilog front-end. The start conditions are represented by nodes and the patterns that trigger the transitions are written on the directed edges. An explanation of the patterns is given in appendix A.4. A bold outline of a node indicates that the start condition it represents can create new `Entry` nodes in the AST.

A.1 Entering a module



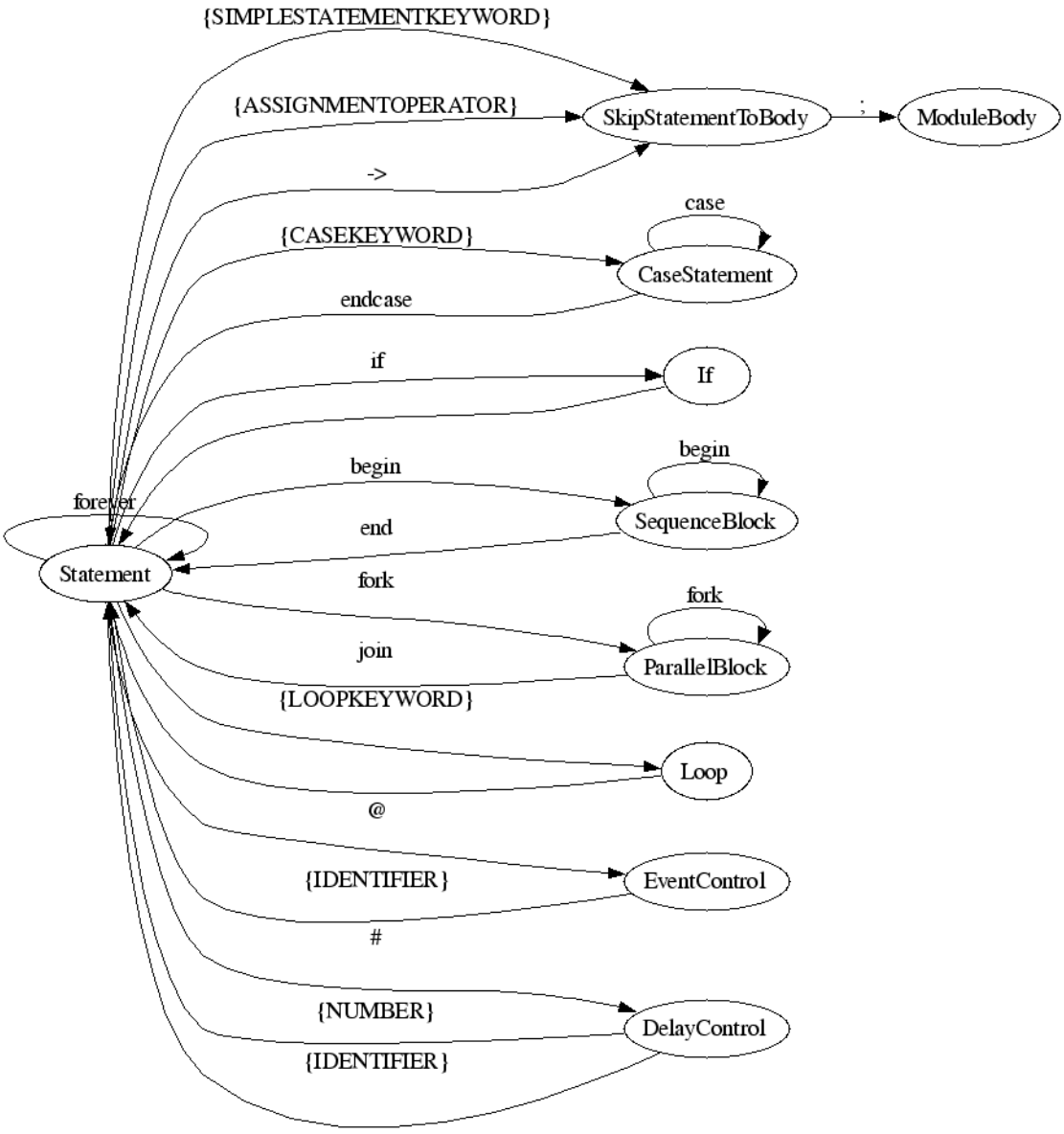
Figur 1: A transition graph for the scanner entering a Verilog module. The scanner is initiated in the Search start condition. The graph continues in figure 2 with transitions from the ModuleBody start condition.

A.2 Inside a module



Figur 2: A transition graph for the scanner inside a Verilog module. Continued from figure 1. Transitions from the Statement start condition are shown in figure 3.

A.3 The Statement start condition



Figur 3: A transition graph for the scanner in the Statement start condition. Continued from figure 2.

A.4 Scanner patterns

The following table defines the patterns that are used to trigger transitions between start conditions in the scanner. The definitions are only semi-formal and do not follow Flex syntax rules.

Pattern	Definition
IDENTIFIER	At least one letter or digit. Defined in [Satterlee]
PORTKEYWORD	input, output, inout
NETKEYWORD	wire, tri, tri1, supply0, wand, triand, tri0, supply1, wor, trior, trireg, reg, parameter
NUMTYPEKEYWORD	time, integer, real
BEHAVIORALKEYWORD	initial, always
SIMPLEMODULEITEM	assign, defparam, event, {GATEKEYWORD}
SIMPLESTATEMENTKEYWORD	disable, assign, deassign, force, release, ->
CASEKEYWORD	case, casex, casez
LOOPKEYWORD	repeat, while, for, wait
ASSIGNMENTOPERATOR	=, <=
NUMBER	An optional numeric base followed by at least one digit. Defined in [Satterlee].
GATEKEYWORD	and, nand, or, nor, xor, xnor, buf, bufif0, bufif1, not, notif0, notif1, pulldown, pullup, nmos, rnmos, pmos, rpmos, cmos, rcmos, tran, rtran, tranif0, rtranif0, tranif1, rtranif1

Table 18: Patterns for triggering scanner transitions. Options are separated by commas.