Sieve: An XML-Based Structural Verilog Rules Check Tool

by

Tina Cheng

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author .....
Department of Electrical Engineering and Computer Science
August 22, 2003

Certified by .....
Krste Asanović
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Sieve: An XML-Based Structural Verilog Rules Check Tool

by

Tina Cheng

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The complexity of microprocessor chip designs continues to grow with every generation. At the same time, the amount of manpower needed for these projects also continues to grow, creating the need for a better integration flow. Due to this trend, many design conventions are set before the implementation of the chip commences to aid in the integration. This thesis describes the development of a suite of tools which check various design rules in accordance with predefined conventions, in particular the SCALE-0 VLSI design conventions. The tool suite consists of units that check naming conventions, units that check that the design is structural Verilog, and units that check leaf signal rules. A flexible input format for describing the rules is also developed so the tool can be easily adapted for new conventions and new chip designs. The input to the tools is a Verilog design file. Icarus Verilog is modified to parse this Verilog into an XML format. The tool then uses this format, along with the rules that have been defined, as inputs and performs the checks that are specified.

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Acknowledgments

First and foremost, I would like to thank Krste Asanović for all his help and guidance through this long thesis process. The road to completion has been a long and windy road for me, and I thank him for his encouragement and patience through the whole journey. I would also like to thank the members of Assam for their help in any matters in which I had questions. I would like to especially thank Chris Batten who answered many of my questions and assisted me in various tasks.

None of this would have been possible without the words of encouragement from my family and friends to continue pursuing this degree because god knows I reconsidered it many times. A huge thanks for my wonderful husband, Philippe, who really pushed me along, and provided aid from proofreading to helping me debug code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

With each new generation of microprocessors, chips continue to grow in scale and complexity. Custom tools have played a larger and larger role in the design of these chips. As the designs increase in size, designers have looked to hardware description languages (HDLs), such as VHDL and Verilog, to abstract away some of the details and make the design more manageable.

Another effect of this growth is the increase in man-power that is needed to make the design work. Often, the various sections of the chip are developed separately and must be integrated together for the final product. From this arises the need for conventions and rules to be defined when implementing different parts of the chip to ensure uniformity and readability in the design as a whole. For example, naming conventions for cells are developed to help in the documentation of the design, to help reduce name clashes, and to ensure that the static checks these tools provide can be done easily.

These larger designs are usually accompanied by a complex tool flow that includes merging tools from many various third parties and often calls for custom tools to be written. These custom tools often have to interact with the Verilog/VHDL code. Tools such as Vexworks [2] give designers an API that they can use to build their own tools without having to delve into the HDL details.

This thesis describes the development and implementation of another set of custom tools that can check the validity of the rules and conventions defined for a design. The tool, known as Sieve, also includes a robust format for defining rules and conventions so that it can be easily adaptable to future designs with different conventions.

Sieve is a tool that is designed primarily for use during the development of microprocessor chips in the SCALE (Software-Controlled Architectures for Low Energy) project, in particular the SCALE-0 chip [1]. The SCALE-0 chip will use TSMC's CLO18 technology and will be implemented in a hierarchical or structural design. The implementation will consist of leaf cells and larger blocks which will be built up from these leaf cells.

1.2 Organization

This thesis is organized as follows. Chapter 2 gives an overview of the Sieve on a top level and gives background on the rules of the SCALE-0 VLSI design. Chapter 3 discusses in detail the framework the XML Verilog parser. Chapter 4 describes the format of the input file which specifies the conventions and rules that the tool suite will check. Chapter 5 describes the framework for the actual checker software. Chapter 6 shows sample files and example rules that the software was tested on and outlines the testing strategy used. Chapter 7 concludes with a summary and future work.

Chapter 2

An Overview of Sieve

2.1 Top-Level View

Sieve is composed of two main units. The first unit takes in structural Verilog code, parses it, and translates into an XML format. The second unit takes in this XML format of the Verilog and an additional rules file and performs the actual checking of the rules. The basic flow is shown in Figure 2-1.

The first unit is the *Parser* unit. This unit is written in C as a target module to Icarus Verilog (IVerilog) [6]. IVerilog is a software tool that compiles Verilog HDL into an internal representation as C objects. The *Parser* uses this internal representation to do a preliminary check to ensure that the input is indeed structural Verilog. It produces a XML output file (*ModuleData.xml*) which is an intermediate representation describing the hierarchy of the input design along with module names and signal names and connections.

The second unit is the *Rule Checker* unit. This is also written in C and uses two outside libraries, libxml [5] and the Carp regular expression library [4]. The inputs to this unit are two XML files. The first is the output from the first unit, *ModuleData.xml*. The second is the rules file, *Rules.xml*. The unit first reads these two files in and parses them into an internal C data structure. It then runs the name check and rules check on these data structures, outputting any error messages. A verbose description of what was checked in the design can be found in the output file,

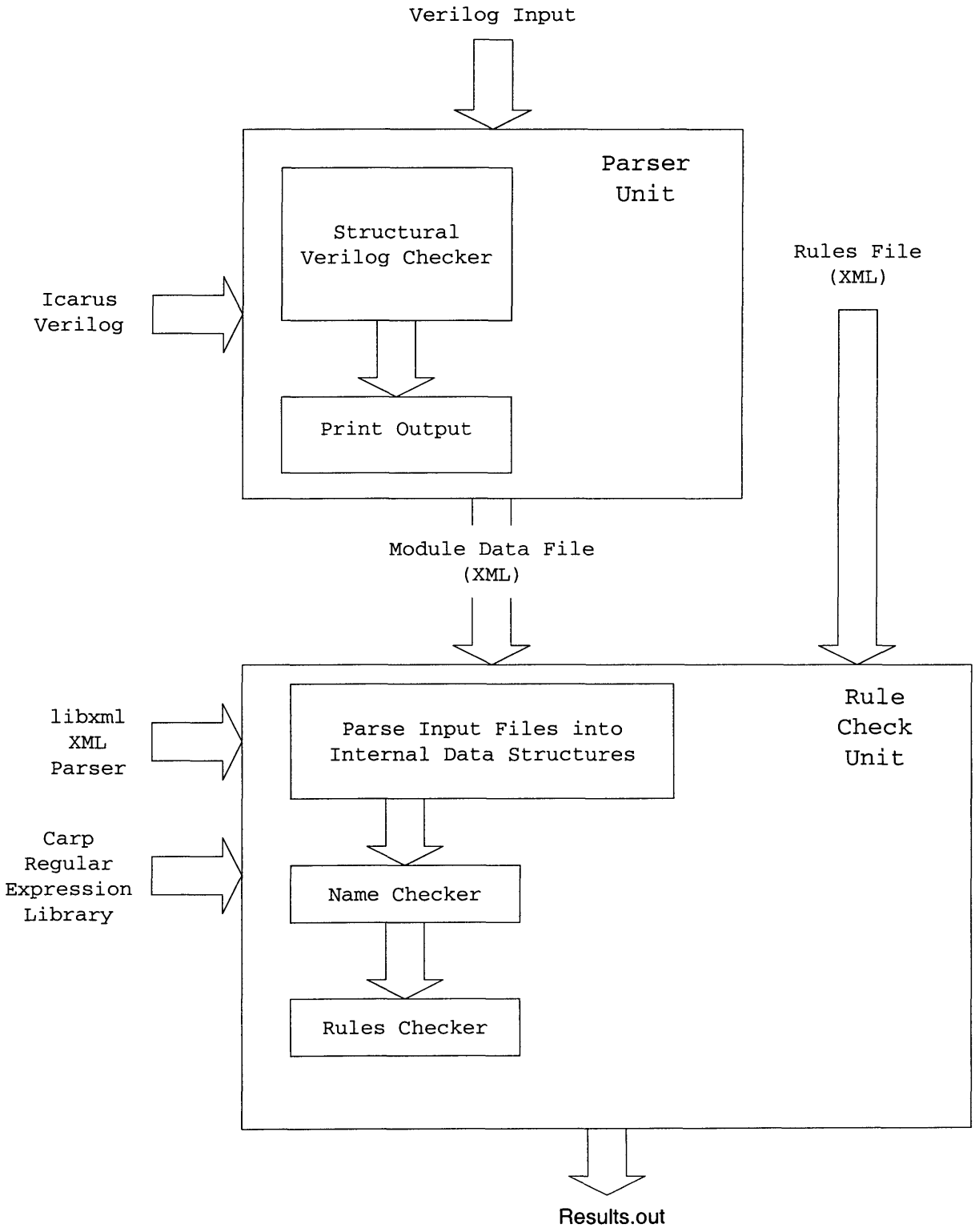

Figure 2-1: Software Flow Diagram

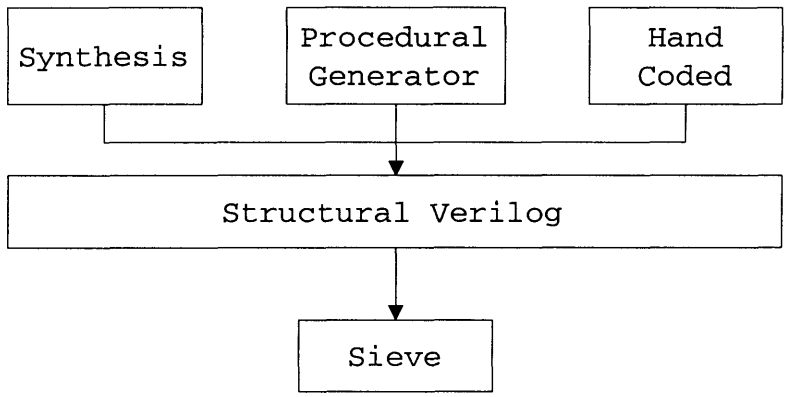

Figure 2-2: SCALE 0 Tool Flow and Sieve

Results.out.

The rules file, which is an input into the second unit, is defined by the user and specifies the rules that the names of the modules and wires must follow, as well as requirements on how inputs and outputs are named within modules. This gives the user flexibility in controlling exactly what will be checked and allows this tool suite to be adaptable to future designs with different rules and conventions.

2.2 SCALE-0 Rules

Though the tool is designed to be flexible for different rule sets and conventions, the examples that will used throughout the rest of this thesis will be from the SCALE-0 VLSI design document [1].

To understand where this tool fits into the design flow for SCALE-0, consider the diagram shown in Figure 2-2.

The SCALE-0 design is translated into structural Verilog in three possible ways. First, the Verilog may come from synthesis. This is most likely written in Verilog RTL and synthesized with Cadence tools, in particular, Cadence Physically Knowledgable Synthesis (PKS). The structural Verilog may also be an output from a procedural tool, such as Spongepaint [5]. Lastly, some parts of the design may be hand coded directly into structural Verilog. Sieve can be run on the derived structural Verilog to check for any rule violations.

2.2.1 Cell Naming Conventions

These conventions are relevant for all standard cells, datapath cell, and pads. In a structural Verilog design, a cell can either be a leaf cell or a collection of leaf and non-leaf cells wired together. Non-leaf cells do not contain any logic or transistors. The cell names are globally unique and adhere to the following naming convention. The naming convention is designed to help with the design documentation, to reduce naming clashes, and to support simple static correctness checks.

The first letter describes the general category of the cell. This letter indicates if a cell should be a leaf cell or non-leaf cell. The acceptable first letters are summarized in Table 2.1.

Descriptor	Category of cell
`s`	Row-based standard cells
`d`	Datapath cells
`e`	Datapath edge cells
`a`	Array cells
`m`	Non-leaf cells

Table 2.1: Cell categories (first letter of prefix)

Non-leaf cell names have the format `m_cellname`. Leaf cell names have a further two-letter prefix following the category which describes how the cell interacts with the clock signal. After this prefix, the name of the actual cell is appended. The two-letter prefix possibilities are shown in Table 2.2.

	Type of leaf cell
`cc`	Combinational circuit, no clock inputs
`pf`	Outputs registered with pos-edge D-flip-flops
`nf`	Outputs registered with neg-edge D-flip-flops
`hl`	Outputs latched with transparent-high latch
`ll`	Outputs latched with transparent-low latch
`hv`	Outputs eval on clk high, invalid on clk low
`lv`	Outputs eval on clk low, invalid on clk high
`xx`	Unspecified — not allowed on standard cells

Table 2.2: Leaf cell types

The actual name of both leaf and non-leaf cells must contain only lowercase letter, numbers, or the underscore (_) character. Leaf cells can also have an optional suffix. This suffix is of the form $pinj$ and it indicates that the gate has a drive equivalent to an inverter with an $i \times 0.1\,\mu$m-wide p-FET and a $j \times 0.1\,\mu$m-wide n-FET. Standard cells and datapath cells are required to have a drive strength suffix. Table 2.3 show some possible suffixes and what their meanings are.

Suffix	Equivalent Drive
_p2n2	Minimum inverter
_p3n2	1.5 p/n ratio inverter
_p6p2	Equal rise-fall times
_p8p2	Skewed towards pullup

Table 2.3: Drive Suffix Samples

Putting all this, some valid sample cell names are given in Table 2.4.

Cell Name	Description
spf_ff_p6n2	Standard cell pos-edge flip-flop
scc_inv_p12n4	Standard cell inv, x2 drive
dcc_imux4_p6n3	Dpath 4-input inverting-mux
dcc_nand2_p3n2	Dpath 2-input skewed nand
dcc_inv_p2n2	Dpath min sized inverter
ecc_cdrv_p18n6	Dpath edge clock driver
axx_srambit	SRAM bit cell
alv_sramsa	Sense amp, eval on clk low
m_cpudp	CPU datapath module

Table 2.4: Sample cell names

2.2.2 Signal Naming Conventions

Signal naming conventions for SCALE-0 are based on timing and are used to perform simple static checks on the clocking strategy. Wire names for leaf modules are made of lower case letters, numbers, and the underscore character. Wire names for non-leaf modules are further appended by a suffix, as shown in Table 2.5.

Suffix	Meaning
_pn	The signal becomes valid before the positive edge and remains valid until the negative edge
_np	The signal becomes valid before the negative edge and remains valid until the positive edge
_p	The signal becomes valid before and until the positive edge
_n	The signal becomes valid before and until the negative edge
_pulse	The signal is not necessarily valid across any clock edges

Table 2.5: Signal suffix types

Furthermore, there are some rules that determine what types of output signals may be produced given input signal types and a leaf cell type. Table 2.6 presents these legal combinations.

Allowable Input Signals	Leaf Cell Type	Output Signal
any	cc	GLB(inputs)
np, pn, p	pf	np
pn, np, n	nf	pn
pn, np, n	hl	np
np, pn, p	ll	pn
pn	hv	n
np	lv	p
pulse	xx	anything

Table 2.6: Relationship of signals to leaf cell types

In Table 2.6, GLB is the Greatest Lower Bound function. The ordering for comparison is defined as follows:

$$\{pulse\} < \{p, n\} < \{pn, np\} \tag{2.1}$$

Chapter 3

Verilog Parser

The parser is a written in C as a target module to Icarus Verilog (IVerilog). The parser outputs an XML format of the structural Verilog input.

3.1 Icarus Verilog

Before delving into the design and implementation details of the actual parser unit, this section gives some background on Icarus Verilog (IVerilog) and how it ties into the design.

IVerilog is the software that Sieve was originally to be built on top of [6]. IVerilog is a tool that compiles Verilog HDL into an internal representation as C objects. Back end modules can then be written to use this representation to do a variety of custom functions. These modules are called targets.

At the start of the project, Sieve was designed to be a target itself and to use directly this internal representation of the Verilog code provided by IVerilog. As the development went underway, it was discovered that other outside libraries were needed. A restriction with Icarus Verilog is that it does not allow outside libraries to be utilized within a target module, so the design was changed accordingly. Icarus Verilog now plays the role of parsing the design, checking that the design is structural Verilog, and outputting the XML structural Verilog format.

The part of IVerilog that is still utilized is the parser, which reads in the Verilog

files and generates an internal netlist. The steps that the parser takes are outlined below.

Step 1: Processing This step utilizes a program called ivlpp. It takes out the 'include and 'define directives and instead inlines them. For 'include directives, the contents of the included file are substituted in place of the line with the directive. For the 'define directives, the actual definition is placed in all references to the define in the code.

Step 2: Parse This step parses the Verilog into a rough PFORM. This is not a complete translation yet as it may still have dangling references and does not yet know who the root module is. The PFORM can be read with a -P option in IVerilog.

Step 3: Elaboration The PFORM is taken and all references are resolved and all instantiations are expanded. The result is a netlist which contains behavioral descriptions, gates, and wires. It does this is two steps: scope and netlist elaboration. In scope elaboration, a tree of NetScope objects is built from scopes and parameters. In netlist elaboration, the PFORM is traversed to generate the actual netlist.

Step 4: Optimization Here, optimizations on the netlist, such as eliminating null circuitry and combinatorial reduction, are performed.

After these steps are taken, the netlist is used to drive the code generator. This code generator is the target tool to be written. To write a new target, the target source C file is placed into a `tgt_name` directory. In the source file, the function that is invoked when IVerilog looks for the code generator is

```
int target_design (ivl_design_t des) {}
```

The parameter that this file takes is the top level structure that contains the whole design. From this structure, the root module can be accessed and the whole design can be traversed, in a recursive manner.

Once the target has been written and compiled, in order for IVerilog to be aware that there is a new target module, it must be installed into the correct directory (`/lib/ivl`) and the file `iverilog.conf` must be modified to include the new target. For example, a target called null will be added by using the following line:

```
[-tnull] <ivl>%B/ivl %[v-v] -C%C %g %W %s %[M-M%M] %[N-N%N] %[T-T%T] -tdll
                          -fDLL=%B/null.tgt -- -
```

3.2 Parser Overview

The parser itself has two main goals. The first is to check the Verilog input to ensure that is in structural Verilog format. This needs to be done before the code is actually translated into the structural XML representation. The second is to parse the Verilog and output the necessary parts of the design to an XML file. Figure 3-1 shows the basic overview of this unit.

The output of the program includes the display of any error messages and two output files, *ModuleData.xml* and *Results.out*. *ModuleData.xml* is the XML file that will hold the parsed design. The format of this file will further be discussed in Section 3.2.3. *Results.out* is an output file that prints out verbose information on what the program sees as it runs.

3.2.1 Structural Verilog Checker

This section checks that the input Verilog modules follow a structural Verilog style. This check must be completed at the initial stage because Sieve assumes that the input is structural Verilog.

The structural Verilog style dictates that all logic should be contained in leaf cells. The non-leaf cells should contain no logic and should only instantiate modules and define how they are connected.

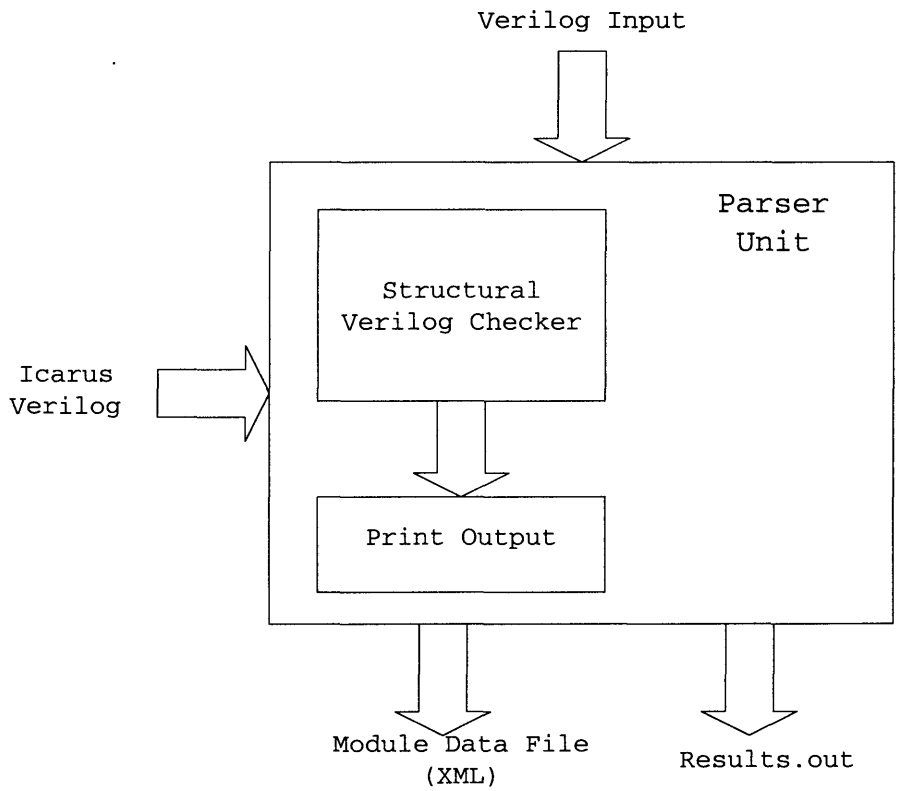

Figure 3-1: Basic Overview of Verilog Parser Unit

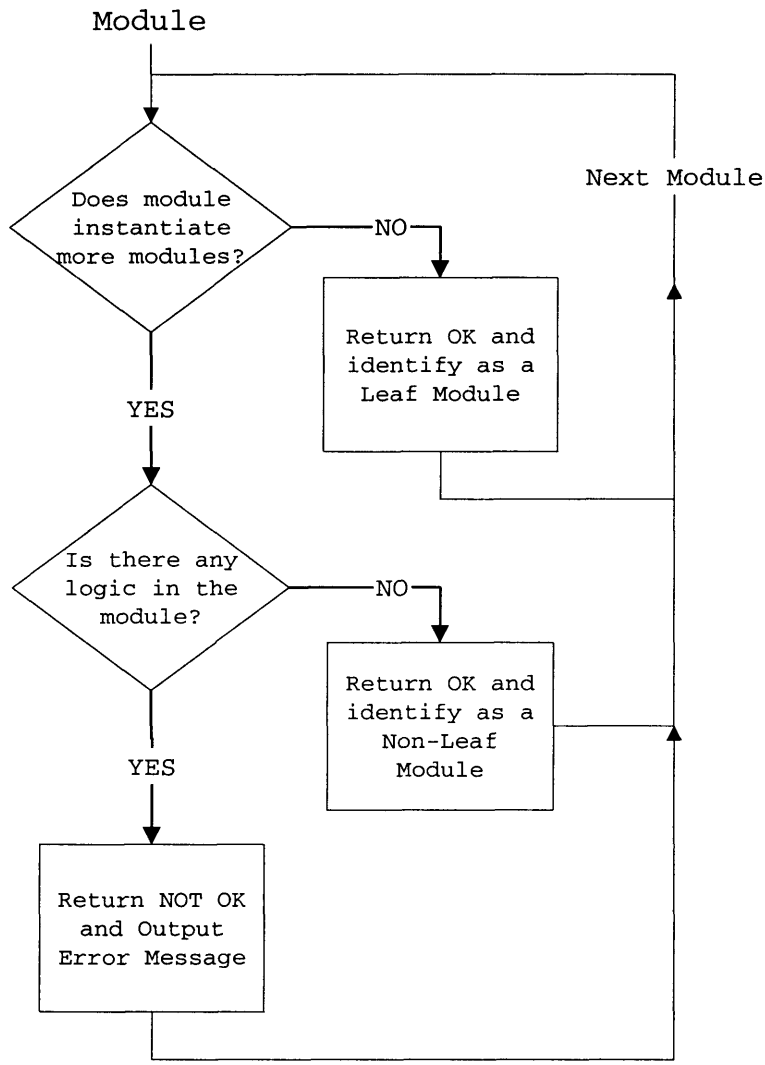

Figure 3-2: Flow Diagram of Structural Check Tool

The basic logic is shown in this flow diagram (Figure 3-2 below). All the modules in the design are iterated through.

From this unit, error messages report whether there is a violation or not. If there is, information about the module where the violation occurred is printed along with a list of the actual violation(s).

During the recursion through all the modules during this check, information about leaf/non-leaf status of each module is recorded to be used when printing the output. The information is kept in a linked list as shown in Figure 3-3.

The linked list is made up of `module` structs. Each struct has two fields. The

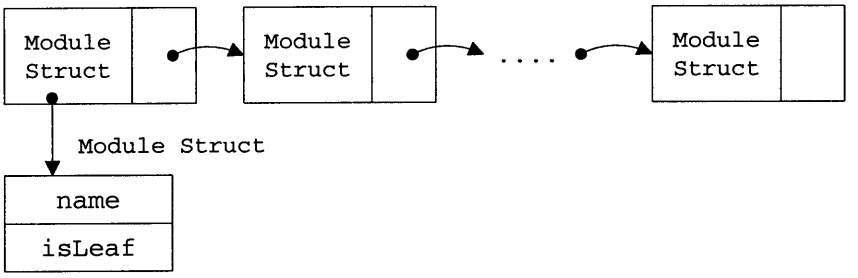

Figure 3-3: Data Structure for Modules in Verilog Parser

`name` field identifies the name of the module. The `isLeaf` field is 1 if the module is a leaf and 0 if it is not.

3.2.2 Print Output

The next section creates the *ModuleData* XML file that will become the input to the rest of the checker program. XML was selected as the output format for the structural Verilog because it is a versatile format that is widely available, supported, and understood. It can be easily read and parsed by libraries directly by another program, in this case, the checker unit of Sieve.

The print section basically iterates through the modules and prints out pertinent information to the XML file. Figure 3-4 shows the basic flow.

For each module, Sieve prints out some information, including various names of the module, whether it is a leaf or non-leaf, and information about the signals it has. Since many modules have more than one signal, the signals are also looped through. Basic information about a signal, such as if it is local and if it is an input or output, are recorded. Each signal also has a number of pins associated with it. These pins are iterated through next. Each pin is connected to a nexus. The nexus is a connection point which contains a list of all the signals that are connected to that point, and therefore, each other. The nexus is traversed and the signal name of each connected signal and the module to which the signal belongs to is also recorded to the *ModuleData* file.

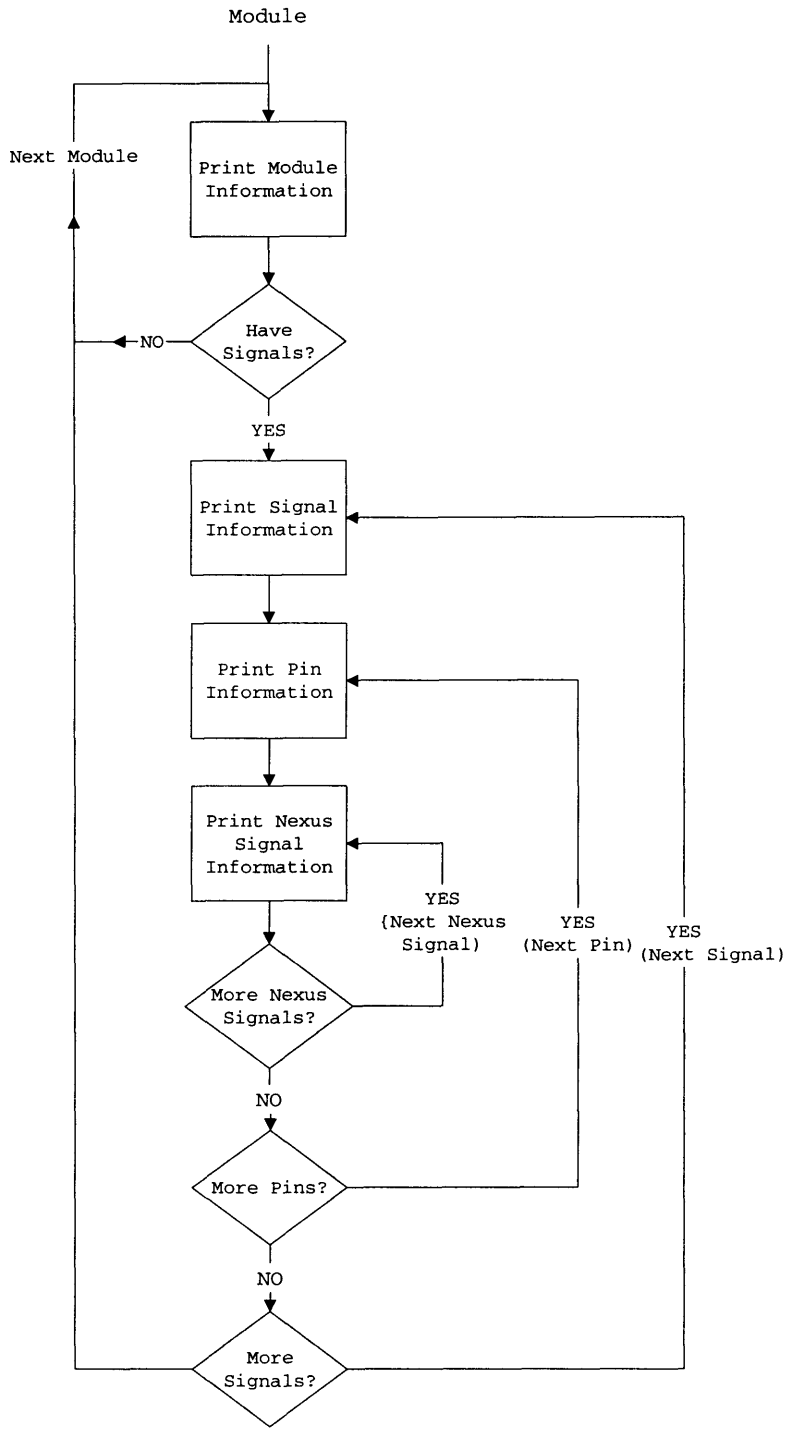

Figure 3-4: Flow Diagram of Creation of XML code

3.2.3 Output File Format

In this section, the basic format of the output *ModuleData.xml* file is described. The schema of the XML looks as follows:

```
<moduleData>
        <module fullName = "str"
                baseName = "str"
                tName = "str"
                isLeaf = "str" >
                <signal name = "str"
                        baseName = "str"
                        numBits = "str"
                        input = "str" >
                        <pin number = "str" >
                             <nexusSignal>
                                     <name>str</name>
                                     <module>str</module>
                                     <pinNum>str</pinNum>
                             </nexusSignal>
                        </pin>
                </signal>
        </module>
</moduleData>
```

The tag <moduleData> indicates the start of the file. Each module is delimited by the <module> tag. Each module has a few attributes which are described in Table 3.1

Attribute	Meaning
fullName	Full Hierarchical Name
baseName	Local Name
tName	Instance Name
isLeaf	1 if Leaf; 0 if not

Table 3.1: Module Attribute Definitions

For example, if a module is instantiated like so inside another module called m_main:

```
module m_main (...) {
  m_beta beta(...);
}
```

Then, attribute *fullName* would be `m_main.beta`, `baseName` would be `beta`, and `tName` would be `m_beta`.

Beyond these attributes, each module contains elements tagged by <Signal>, representing the signals that are present in the module. Each signal has a few attributes also (Table 3.2).

Attribute	Meaning
name	Full Hierarchical Name
baseName	Local Name
numBits	Number of Bits in Signal
input	Direction of Signal: input, output, inout

Table 3.2: Signal Attribute Definitions

For each bit that a signal has, there is an element inside the signal tagged with <Pin> representing that bit of the signal. Each pin has a attribute called *number* which keeps track of which bit in the signal the pin represents. Each pin also contains <nexusSignal> elements. All the <nexusSignals> in a pin make up a nexus where all the <nexusSignals> are connected. Each <nexusSignal> contains 3 elements. The first is <name> which is the local base name of the signal. The second is <module> which indicates which module the signal belongs to. The <module> is the full hierarchical name. The third element is the <pinNum> which is the pin number of the signal on the other module which is connected to the nexus.

3.3 How to Run

In order to run the Verilog Parser module on a system where both IVerilog and the target have been installed, go to the directory where the Verilog source input is and invoke the program with the following command:

```
%iverilog -tparser sourcefile.v
```

This will run Icarus Verilog on the source file with the parser module as the target. The two output files, *Results.out* and *ModuleData.xml* will be created in the directory that the source file is in.

If neither the parser nor IVerilog are installed, first, install IVerilog. In the directory of Icarus Verilog where all the source files are located, create a new target and install the parser there.

The last thing that needs to be done before the target can be used is for it to be added to the configure file. In the install directory's `lib` folder, there should be a folder called `ivl`. This folder should contain a copy of `parser.tgt`. There should also be a file called `iverilog.conf`. In this file, this line should be added:

```
[-tparser] <ivl>%B/ivl %[v-v] -C%C %g %W %s %[M-M%M] %[N-N%N] %[T-T%T]
                    -tdll -fDLL=%B/parser.tgt -- -
```

Chapter 4

Input Format Specification

A rule convention file must be defined and provided as the input to the rule check tool. This file provides flexibility for the program to be used in other designs besides SCALE-0. The file is to be written by the tester and also needs to be easily understood and easy to write.

As for the structural Verilog output described in Section 3.2.2, XML was also selected to be the format for this file for reasons of versatilility. The input format has four main elements: `Sets`, `Regex`, `Rules`, and `Functions`. Each of these elements is associated with a tag. The tag might contain attributes describing the element. They also contain more elements.

The basic outline for the document is:

```
<CheckRules>
        <Sets>
        ...
        </Sets>

        <Regex>
        ...
        </Regex>

        <Rules>
        ...
        </Rules>

        <Functions>
        ...
        </Functions>
</CheckRules>
```

There should only be one occurrence of each of these four elements. The order that they are written in does not matter. The labels are case sensitive.

4.1 Sets

The section for `Sets` is devoted to the definition of categories. Categories can be used in the regular expressions that are to be defined later in the file. Categories are also used in the definition of rules.
Categories are represented by the <category> tag and the syntax is as follows:

```
<category name=''CategoryName''>
        <item>a</item>
        ...
        <item>x<item>
</category>
```

Each <category> has an attribute called *name*. The categories contain elements called *item* which represent the members of the category. For example, for SCALE-0, there is a category called `SignalSuffix` (see Table 2.5) and could take on the values `np`, `pn`, `n`, `p`, and `pulse`. In this file, it would look as follows:

```
<category name=''SignalSuffix''>
        <item>np</item>
        <item>pn</item>
        <item>n</item>
        <item>p</item>
        <item>pulse</item>
</category>
```

Some rules that the `Sets` section must follow are:

- all Category *name* attributes must be unique and cannot be empty
- all Categories must contain at least one *item*
- *items* cannot be empty
- all *items* must be unique

4.2 Regular Expressions

Regular expressions are where the naming structures of the modules and wires are defined. Table 4.1 shows the four regular expressions that *must* be defined.

Regex expression	Meaning
`LeafModuleName`	Name of leaf modules
`NonLeafModuleName`	Name of non-leaf modules
`LeafWire`	Name of wires into leaf modules
`NonLeafWire`	Name of wires into non-leaf modules

Table 4.1: Mandatory Regular Expressions

Regular expressions are defined by <regex> tags and the syntax is as follows:

```
<Regex>
        <Expression name=''regexName''>...</Expression>
        ...
        <Expression name=''regexName2''>...</Expression>
</Regex>
```

The ...between the `expression` tags are where the actual regular expression definition goes. The regular expressions are parsed by the Carp Expression Library (see 5.1.1) and should follow the industry standard regular expression format. In addition to the syntax provided by Carp, this tool defines a couple more variations.

The first variation is that if there are no restrictions on how the naming of a component goes, the regular expression can be defined as an asterisk (*) to show that it can match anything. This can also be expressed using regular expression syntax as `.+`.

The second is that category names are allowed to be part of the regular expression. To use a category name, it should be enclosed between two forward slashes (/.../). The category name should also be one that has been defined in the `Sets` section of the file.

As an example, SCALE-0 defines a leaf module to have a couple of pre-defined prefixes and possibly a suffix. For a leaf module, the name definition regular expression would appear like:

```
<Expression name=''LeafModuleName''>
        /LeafCellCategory//LeafCellType/_([0-9]|[a-z]|_)+(_p[0-9]+n[0-9]+)?
</Expression>
```

The first reference is to a category, `LeafCellCategory`, which can be either `s`, `d`, `e`, or `a`. The second reference is to another category, `LeafCellType`, which can be `cc`, `pf`, `nf`, `hl`, `ll`, `hv`, `lv`, or `xx`. This is followed by string of length one or more of a mix of numbers, lower-case letters, or the underscore character. Finally, this can be appended by an optional suffix of `p`, followed by numbers, `n`, followed by numbers.

There are also a few rules that regular expression definitions should follow:

- all Expression *name* attributes must be unique and cannot be empty
- all Expressions from Table 4.1 must be defined
- all Expressions must be valid regular expressions as defined above and cannot be empty strings.

4.3 Rules

The section for `Rules` is where the rules are defined. Rules are defined in table format. They define how the categories, defined in `Sets`, will interact with each other.
The basic format for a Rule is as follows:

```
<Rules>
        <Rule category=''CategoryName''>
                <ColumnDefinitions>
                        <Column>...</Column>
                        ...
                        <Column>...</Column>
                </ColumnDefinition>

                <Data>
                        <Row>
                                <Column>...</Column>
                                ...
                                <Column>...</Column>
                        </Row>
                        ...
                        <Row>
                                <Column>...</Column>
```

```
                        ...
                        <Column>...</Column>
                    </Row>
                </Data>
        </Rule>
        ...
</Rules>
```

The <Rules> tag defines the beginning of this section. Each rule is represented with a <Rule> tag. A rule must have a *category* attribute. This *category* attribute is either "Leaf" or "NonLeaf". The way the tool works is that it traverses the hierarchy tree of the Verilog design. For each module, it determines if a rule should apply to this module based on if the rules applies to a leaf module or not and if the module is a leaf module or not. The rule itself then has two main elements, <Column Definitions> and <Data>.

As fore-mentioned, rules are represented in a table format. Each rule has x columns and y rows. The <Column Definition> element is where the headings for the columns are defined. These are used to interpret the data that is presented in the table. The <Column Definition> contains items called <Columns>:

```
<ColumnDefinition>
        <Column type=''NonLeafWire'' inout=''input''>SignalSuffix</Column>
        ...
</ColumnDefintion>
```

Each column header has up to two attributes: *type* and *inout*. *Type* identifies which regular expression (see 4.2) this column applies to. This value should be `LeafModuleName`, `NonLeafModuleName`, `LeafWire`, or `NonLeafWire`. If the value is `LeafWire` or `NonLeafWire`, the *inout* attribute must also be defined. The *inout* value can be equal to `input`, `output`, or `inout`. This value defines if this column is relevant for input wire, output wires, or both, respectively.

There are two ways to define rules in this section. The two ways differ in what the data items defined for the rule mean.

4.3.1 Category Based Rule Definition

In the first way, between the <Column> start and end tags, the content should be *category* and should be the name of a `Set` *category* that was defined in Section 4.1 and the <category> should be a part of the regular expression definition for *type*. Putting this all together, each column is defined by a regular expression, an optional in/out value, and a category. In the example above, the column definition indicates that data from that column corresponds to `NonLeafWire` names which are inputs and the part of the regular expression that is of interest is the category `SignalSuffix`.

The second element of the rule is the <Data>. The <Data> is made up of <columns> and <rows>. The meaning of each column of data is determined by the <Column Definition> that was just described. The items that go under a certain column must be members of the *category* that was defined as part of the <Column Definition>. The only exceptions to this are if the item is an asterisk (*) or a function name. An asterisk indicates that that item can be anything from the *category*. A function name is defined by a percent sign (%), followed by the function name. The function should be one that is defined in the next section, 4.4. If the item can take on multiple values from the category, this can be represented by comma separated values.

As an example, consider the rule from SCALE-0 shown in Table 2.6. This table is turned into the following Table 4.2.

LeafModuleName LeafCellType	NonLeafWire SignalSuffix input	NonLeafWire SignalSuffix output
cc	*	%GLB
pf	np,pn,p	np
nf	np,pn,n	pn
hl	pn,np,n	np
ll	np,pn,p	pn
hv	pn	n
lv	np	p
xx	pulse	*

Table 4.2: XML conversion for Relationship of Signals to Leaf Cell Types

Function GLB will be defined in the next section.

Given this translated Table 4.2, the actual XML code will look like:

```
<Rule category=''Leaf''>
        <ColumnDefinitions>
                <Column type=''LeafModuleName''>LeafCellType</Column>
                <Column type=''NonLeafWire'' inout=''input''>
                        SignalSuffixes
                </Column>
                <Column type=''NonLeafWire''inout=''output''>
                        SignalSuffixes
                </Column>
        </ColumnDefinitions>
        <Data>
                <Row>
                        <Column>cc</Column>
                        <Column>*</Column>
                        <Column>%GLB</Column>
                </Row>
                <Row>
                        <Column>pf</Column>
                        <Column>np,pn,p</Column>
                        <Column>np</Column>
                </Row>
                <Row>
                        <Column>nf</Column>
                        <Column>np,pn,n</Column>
                        <Column>pn</Column>
                </Row>
                <Row>
                        <Column>hl</Column>
                        <Column>pn,np,n</Column>
                        <Column>np</Column>
                </Row>
                <Row>
                        <Column>ll</Column>
                        <Column>np,pn,p</Column>
                        <Column>pn</Column>
                </Row>
                <Row>
                        <Column>hv</Column>
                        <Column>pn</Column>
                        <Column>n</Column>
                </Row>
                <Row>
                        <Column>lv</Column>
                        <Column>np</Column>
                        <Column>p</Column>
                </Row>
                <Row>
                        <Column>xx</Column>
                        <Column>pulse</Column>
                        <Column>*</Column>
```

```
                </Row>
        </Data>
</Rule>
```

4.3.2 Non-Category Based Rule Defintion

In the second way to define a rule, there is no *category* specified between the <Column> start and end tags. The <Column Definition> looks as follows:

```
<ColumnDefinition>
        <Column type=''LeafWire'' inout=''input''></Column>
        ...
</ColumnDefintion>
```

The *type* and *inout* attributes still have the same meanings as described in Section 4.3.1. However, now, there is no category defined.

The data still has the same format as described, but each item in the data table has a different meaning now. The items in the data tables should now be regular expressions defined by the rules in Section 4.2.

As an example of when this rule can be used, consider the SCALE-0 convention of when the suffix _p*inj* is mandatory. According to the SCALE-0 design document, standard cells and datapath cells are required to have this drive strength suffix. It is optional in all other types of cells. This can be translated into the following Table 4.3

LeafModuleName LeafCellCategory	LeafModuleName null
s	s/LeafCellType/_([0-9]|[a-z]|_)+_p[0-9]+n[0-9]+
d	d/LeafCellType/_([0-9]|[a-z]|_)+_p[0-9]+n[0-9]+
e	e/LeafCellType/_([0-9]|[a-z]|_)+_p[0-9]+n[0-9]+

Table 4.3: XML conversion for Suffix Rule

This then translates into actual XML the same way that the category based rules do in the previous section.

There are also a few rules that rule definitions should follow:

- <Rule> must have a <ColumnDefinition> and <Data> items

- <Rule> *category* must be defined to be Leaf or NonLeaf
- <Column> must have a type (LeafModuleName, NonLeafModuleName, LeafWire, or NonLeafWire)
- <Column> must have an *inout* defined IF they are LeafWire or NonLeafWire type (input, output, inout)
- If a <Column> has non-empty content, this Set specification should occur in the regular expression defined by the *type*
- The number of columns under <ColumnDefinitions> should be equal to how many columns each row has under <Data>

4.4 Functions

The last section that can be included is the Functions section. A function can be used in the Rules definition in the previous section. Because a function is tied to a rule, the function inputs must derived from the Rule table where it was called from. Given these inputs, the function determines what the appropriate outputs should be. To define a function, the syntax is as follows:

```
<Functions>
        <Function name=''GLB'' Category=''SignalSuffix''>
                <Input>...</Input>
                <ColumnDefinitions>
                        <Column>...</Column>
                        ...
                        <Column>...</Column>
                </ColumnDefinition>

                <Data>
                        <Row>
                                <Column>...</Column>
                                ...
                                <Column>...</Column>
                        </Row>
                        ...
                        <Row>
                                <Column>...</Column>
                                ...
                                <Column>...</Column>
```

```
                    </Row>
                </Data>
        </Function>
        ...
</Functions>
```

The *Functions* tag defines the beginning of this section. Each function is represented with a *Function* tag. A function has two attributes. The first is a *name* for the function which defines how it should be referred to in the Rules Section. The second is a *category* which defines the set from which the outputs should be from. In the example above, the function name is `GLB` and the category is `SignalSuffix`. The outputs should therefore be a subset of {`n, p, np, pn, pulse`} (See Table 2.5).

The first element is the <`Input`> section. This section defines where the input to the function is. Since functions are called from rules, this is defined with similar attributes that define a <`Rule column`>, namely, a *type*, a *inout*, and a *category*.

```
<Input type=''NonLeafWire'' inout=''input'' category=''SignalSuffix />
```

Function are also represented in a tabular format. This option was weighed against other possibilities (simple If statement definitions, allowing the user to write a piece of custom code, etc) and was decided on as the option that was versatile and less complicated than the others, while still being able to do everything it needed to do.

Each <`Function`> has x columns and y rows. The <`Column Definition`> element is where the headings for the columns are defined. The headings are pulled from the members of the `Set` defined by the *category* attribute of the <`Input`> element. In this case, the columns should be labeled `n`, `p`, `pn`, `np`, and `pulse`. Not all the members have to have their own column and the order of the columns does not matter as long as the data columns correspond.

The next element of the function is the <`Data`> section. This section is composed of <`Row`> elements, and within those, <`Column`> elements. Each item can take on the values of 1, 0, or asterisk (*). If an item is a 1 in a certain column, it means that the category (defined in <`Input`> definition) in regular expression for the module currently being evaluated must match the column header for the column. If an item

is 0, it must not match. If it is *, it does not matter if it matches or not. The exception is for the last <Column> for each <Row>. This column is the output column. This is the output the function returns if that row is the right row for the inputs. The values in this column should be members from the Set determined by the *category* attribute of the <Function>.

To see how a function is put into a table format, consider the GLB (Greatest Lower Bound) function. From Table 2.6, the ordering was defined as:

$$\{pulse\} < \{p, n\} < \{pn, np\} \tag{4.1}$$

Intuitively, this says that if any of the inputs are of type pulse, then the lowest bound must be pulse. Otherwise, if they have a p or n but no pulse, the lowest bound is p or n. Lastly, if they have no pulse, n, or p, but only pn or np, the lowest bound is pn or np. This is represented in a table as:

pulse	p	n	np	pn	Output
1	*	*	*	*	pulse
0	1	*	*	*	n,p
0	0	1	*	*	n,p
0	0	0	1	*	np,pn
0	0	0	0	1	np,pn

Table 4.4: GLB in Table Format

This is translated in the XML format as:

```
<ColumnDefinitions>
        <Column>pulse</Column>
        <Column>p</Column>
        <Column>n</Column>
        <Column>np</Column>
        <Column>pn</Column>
</ColumnDefinitions>
<Data>
        <Row>
                <Column>1</Column>
                <Column>*</Column>
                <Column>*</Column>
```

```
                <Column>*</Column>
                <Column>*</Column>
                <Column>pulse</Column>
        </Row>
        <Row>
                <Column>0</Column>
                <Column>1</Column>
                <Column>*</Column>
                <Column>*</Column>
                <Column>*</Column>
                <Column>n,p</Column>
        </Row>
        <Row>
                <Column>0</Column>
                <Column>0</Column>
                <Column>1</Column>
                <Column>*</Column>
                <Column>*</Column>
                <Column>n,p</Column>
        </Row>
        <Row>
                <Column>0</Column>
                <Column>0</Column>
                <Column>0</Column>
                <Column>1</Column>
                <Column>*</Column>
                <Column>np,pn</Column>
        </Row>
        <Row>
                <Column>0</Column>
                <Column>0</Column>
                <Column>0</Column>
                <Column>0</Column>
                <Column>1</Column>
                <Column>np,pn</Column>
        </Row>
</Data>
```

There are also a few rules that regular expression definitions should follow:

- the *name* attribute for the function must not be empty
- <Input> must have declared *type* (`LeafModuleName`, `NonLeafModuleName`, `LeafWire`, or `NonLeafWire`)
- If <Input> type is `LeafWire` or `NonLeafWire`, the inout must be defined (input, output, inout)

- If <Input> *category* is defined, it must be a this set specification should occur in the regular expression defined by the *type*
- In <ColumnDefinition>, the values for the columns must come form the set defined by <Input> *category.*
- If <Input> *category* not defined, the values for the columns must be a valid regular expression
- The number of columns per row in <Data> must be one greater than the number of columns in <ColumnDefinition> (for output)
- The output values must come from the set defined by <Function> category.
- If <Function> *category* not defined, the output values must be a valid regular expression

Chapter 5

Checker Program

The Checker unit of Sieve is where the crux of the rule checks are conducted. By the time this unit is reached, the Verilog has already been checked to be structural Verilog and an XML format of the code has been created. It takes this Verilog XML file and another file with an XML representation of the rules to be checked as input.

The first step is for Sieve to parse these two input files. In order to do this, it relies on the help of the libxml (see Section 5.1.2). Once the input files are parsed into internal data formats, the actual rule checking is done. This part relies heavily on the use of regular expressions for matching module and wire names to the pre-defined expressions. Both these libraries are discussed in detail in the next section.

5.1 Library Overview

5.1.1 Carp Expression Library

The Carp Expression Library provides the support for defining regular expressions and returning information on how a string matches the regular expression [4]. Regular expressions are used in defining the naming conventions of both signal and cell names (see Section 2.2.1 and 2.2.2).

The following describes some of the most commonly used syntax rules for regular expressions. For a complete collection of all the legal syntax that Carp allows, see

the Carp Regular Expression Library documentation [4].

Special Characters

The special characters for Carp are *, +, ?, {, }, [,], (,), \, <, >, ^, $, |, ., carriage return, linefeed, and tab. Special characters have special meaning when present in a regular expression. All other characters will match to their literal selves. In order to write a regular expression that matches one of the special characters, the character must be preceded by a \ in the expression. This is an escape sequence and can also include the following three: *num* for octal characters, \x*num* for hexadecimal characaters, and \c*char* for control characters.

Modifiers

Table 5.1 illustrates the different modifiers that can be used and their meanings. Modifiers apply to the item immediately preceding them in the regular expression.

Modifier	Meaning
*	Item must be matched 0 or more times
+	Item must be matched 1 or more times
?	Item must be matched 0 or 1 times
num	Item must be matched *num* times
min,	Item must be matched at least *min* or more times
min, max	Item must be matched between *min* and *max* times

Table 5.1: Carp Modifiers

Extended Items

Extended items are pre-defined items or groups that have a special meaning. Table 5.2 lists what the items are. Extended items are enclosed between < and >.

Anchors

Anchors determine a bound on where a match must happen. This bound can be either be on a line or a word boundary. ^ matches the beginning of a line and $

Extended Item	Meaning
<digit>	Matches any digit
<lower>	Matches any lower case letter
<upper>	Matches any upper case letter
<alpha>	Matches any letter
<tab>	Matches a tab
<newline>, <nl>	Matches a newline character
<spacechar>	Matches one character with space syntax
<wordchar>	Matches one character with word syntax

Table 5.2: Carp Extended Items

matches the end of a line. <beginningofword> matches at the beginning of a word and <endofword> matches at the end of the word.

Groups

The following special characters include character classes and groupings. To group items together, add a preceding (and append a) to the end of the items. Character classes are defined between [and]. For example, [0-9] defines all numbers between 0 and 9. [^ begins a negated character class. Custom classes may be declared using the alternation symbol, |. If placed between two items, it matches if it matches either of the two items.

5.1.2 LibXML2

LibXML2 is a C language library which implements functions for reading, creating, and manipulating XML data [5]. It is used in this project to read in and parse a XML rule definition file and a XML module and signal hierarchy file. XML is based on the concept of a document composed of a series of entities. Each entity can contain one or more logical elements. Each of these elements can have certain attributes or properties. XML provides a formal syntax for describing the relationships between the entities, elements, and attributes that make up the document. LibXML2 provides many methods and functions that allow for easy parsing of XML documents from inside of a C program. Though it has more capabilities, it is used here for parsing the

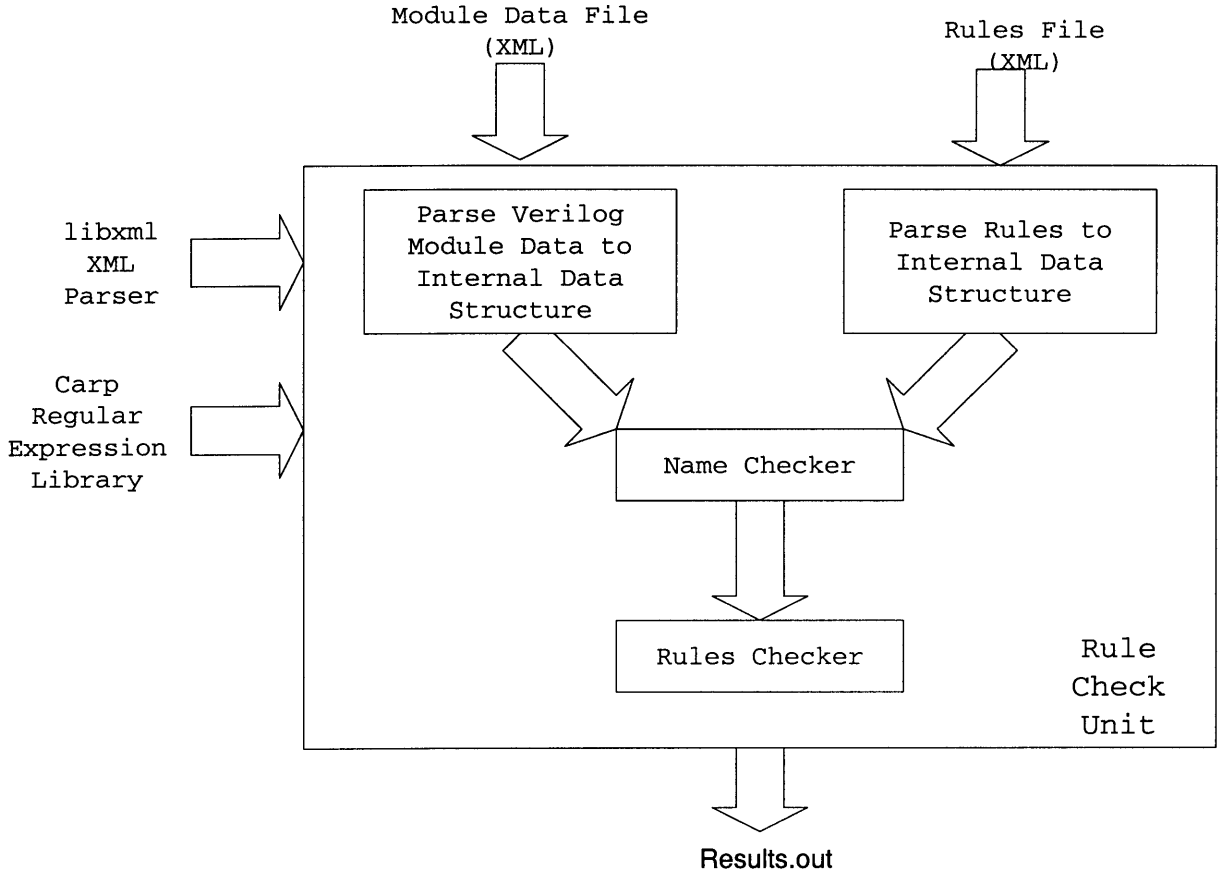

Figure 5-1: Overview of Checker Program

XML, retrieving values for attributes, and retrieving values for text inside an element in this project.

5.2 Overview

The checker program currently does two checks, a name check and a rules check. The basic code flow is indicated in Figure 5-1.

By default, *ModuleData.xml* contains the parsed Verilog from Section 3.2.3. *Rules.xml* contains the rules to check and follows the format defined in Section 4. Both these defaults can be overwritten at the command line prompt when Sieve is invoked.

The first step the program takes is to parse both the input files. It relies on the libxml library functions to perform this step. The files are read in recursively and

comparison of tag values determine what data is being read in. The data is then recorded into internal data structures which are discussed in Section 5.3.

Once the internal data structures have been filled by the input file data, a name check is performed. The name check traverses the entire design and makes sure that each module and each signal follows the naming conventions defined in *Rules.xml*. Afterwards, all the rules are checked for each module. Any subsequent errors are reported.

5.3 Parsing

It was decided that both the input files would be parsed once at the beginning of the program. The information from the files is stored into internal data structures which can be easily referred to. Most of the data structures are C structs containing strings, arraylists, and linked lists.

5.3.1 Parsing RuleCheck

When parsing *Rules.xml*, the document is traversed four times. The first time gathering all `Set` data, the second time gathering all `Regex` data, then `Rules` data, and finally all `Functions` Data. The traversal always happens in this order so the order that these tags occur in the input file does not matter.

Parse Sets

Parsing the sets is a relatively straight-forward process of simply filling out the following data structure (Figure 5-2).

Sets are stored in an array of `set` structs. Each struct has three fields. The first field, `name`, is the name of the category. `Value` is a pointer to an array of strings that holds the actual values that are in the category. `ValueCount` keeps track of the number of values.

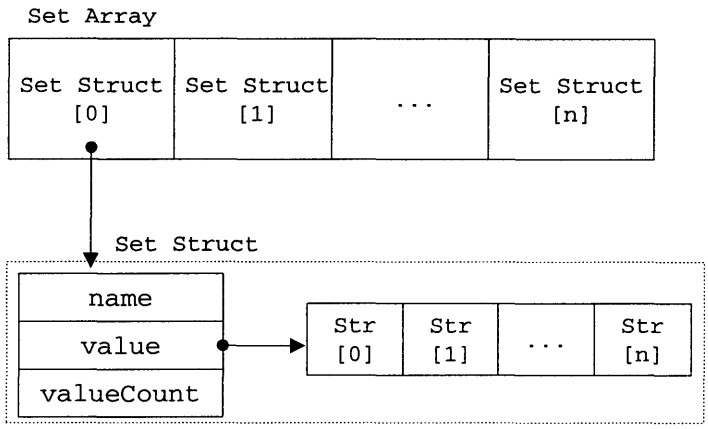

Figure 5-2: Set Array Data Structure

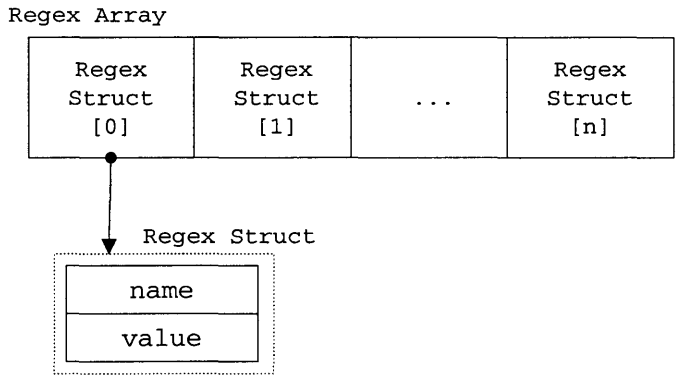

Figure 5-3: Regular Expression Array Data Structure

Parse Regex

Parsing the regular expression is completed in a similar way to parsing the sets. The data structure holding the information is shown in Figure 5-3.

Regular expressions are stored in an array of `regex` structs. Each struct has two fields. The `name` field stores the name of the expression and the `value` field holds the actual expression.

Parse Rules

The rules are parsed and stored into a table format much in the same manner they are represented in the input file. This data structure in Figure 5-4 holds the information

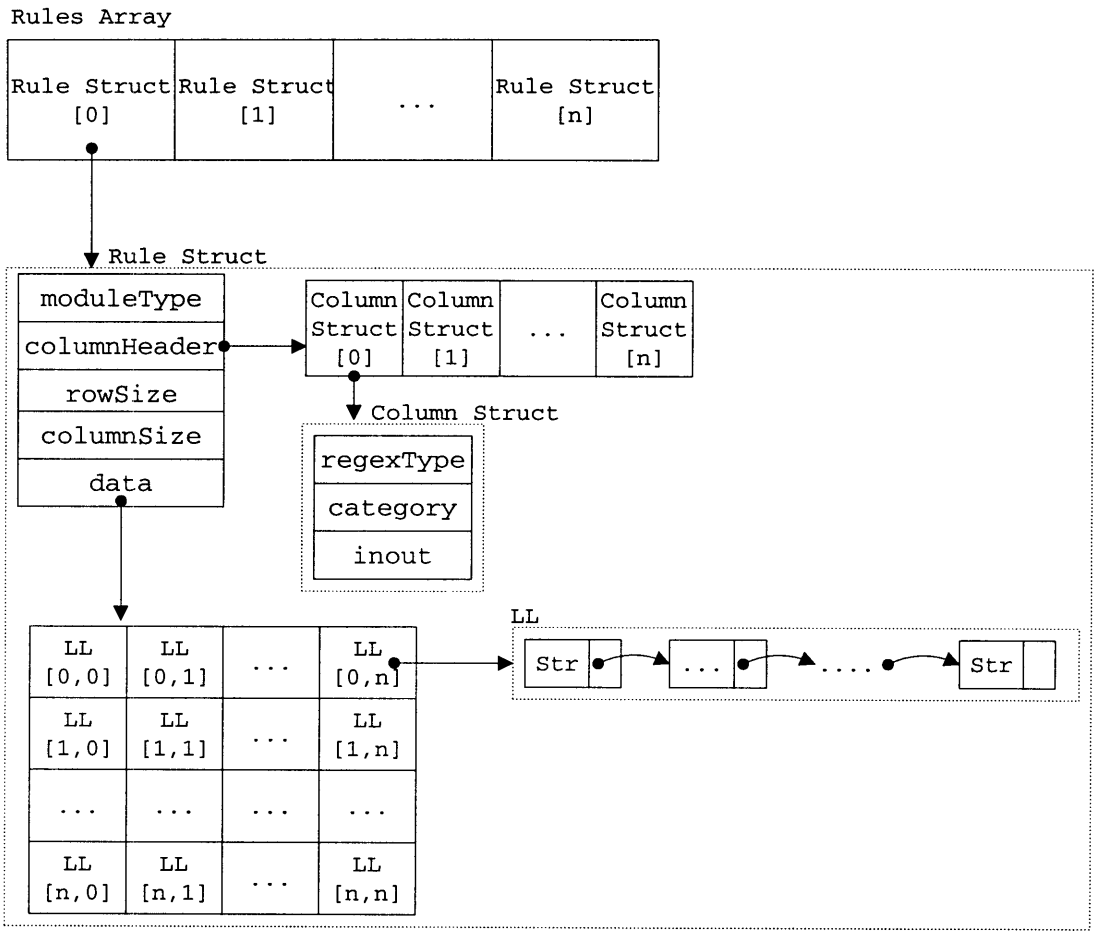

Figure 5-4: Rule Array Data Structure

about the rules.

The rules are again stored in an array of `rule` structs. Each struct has five fields. The `moduleType` field contains the type of module this rule affects (leaf or non-leaf). The `columnSize` and `rowSize` fields respectively define how many columns and rows there are in the rule array. The `columnHeaders` field holds the information about the column headers for the rule array. It is another array of structs, `column` structs, which contain 3 additional fields. The `regexType` field contains the regular expression name that the column refers to. The `category` field contains the set category the column refers too. The `inout` field determines if the column should be applied to inputs, outputs, or both. The final field in the `rules` struct is a two-dimensional array called `data`. This array is composed of linked lists of strings.

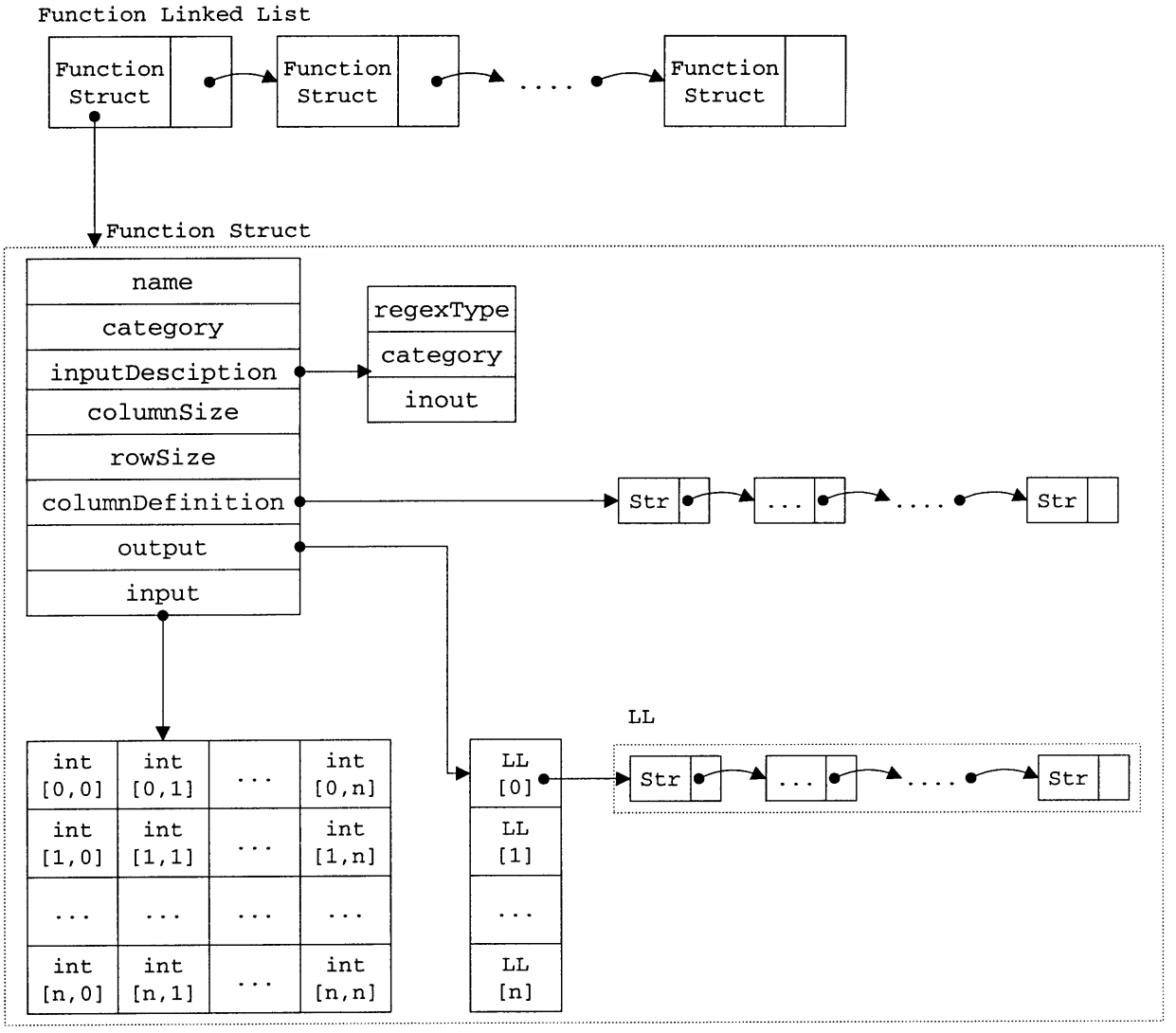

Figure 5-5: Function Linked List Data Structure

Parse Functions

The functions are parsed and stored into a table format like they are formulated in the input file. The data structure for the functions is as follows in Figure 5-5.

The functions are stored in a linked list of `function` structs. A linked link was chosen over an array as in the previous structures since it is easier to maintain and there was no need in this case to access by index into the data structure as there was in the other cases. The structs have eight fields total. The `name` field holds the name of the function. The `category` field holds the output set category. The `rowSize` and

`columnSize` fields respectively hold the number of rows and columns in the function table. The `inputDesciption` field is a struct of 3 fields, `regexType`, `category`, and `inout`. This field defines where the inputs to the function are derived from. These fields have the same meanings as defined in Section 5.3.1. The `columnDefinitions` field holds the definitions of the columns of the function table. It is a ist of character strings. The `input` field is a two dimensional array of integers. The `output` field is an array of lists. A linked list at index i of the array corresponds to the output for row i in the function table. The format is a list of strings since the function can return multiple acceptable values given a specific input.

5.3.2 Parsing ModuleData

When parsing *ModuleData.xml*, the program loops through all the modules and individually parses each module. Each module is parsed into the following data structure shown in Figure 5-6.

Module

The modules are stored in a linked list of `module` structs. Each of the `module` structs are made up of five fields. The `fullName` field stores the full hierarchical name of the module, while the `baseName` field stores the local name and the `tName` field stores the instance name. The `isLeaf` field is an in integer that is 1 is the module is a leaf and a 0 if it is not. The final field is a `signals` field which is a pointer to a linked list of signal structs.

5.3.3 Signals

Each signal is represented by a `signal` struct, which has five fields. The `name` field stores the full hierarchical name of the signal and the `baseName` field stores the local name. The `input` field is a value equal to 0 if the signal is an output, 1 if the signal is an input, and -1 if the signal is an inout. The `numBits` field is an integer that defines how many bits are in the signal. The final field, `pins`, is a pointer to a linked list of

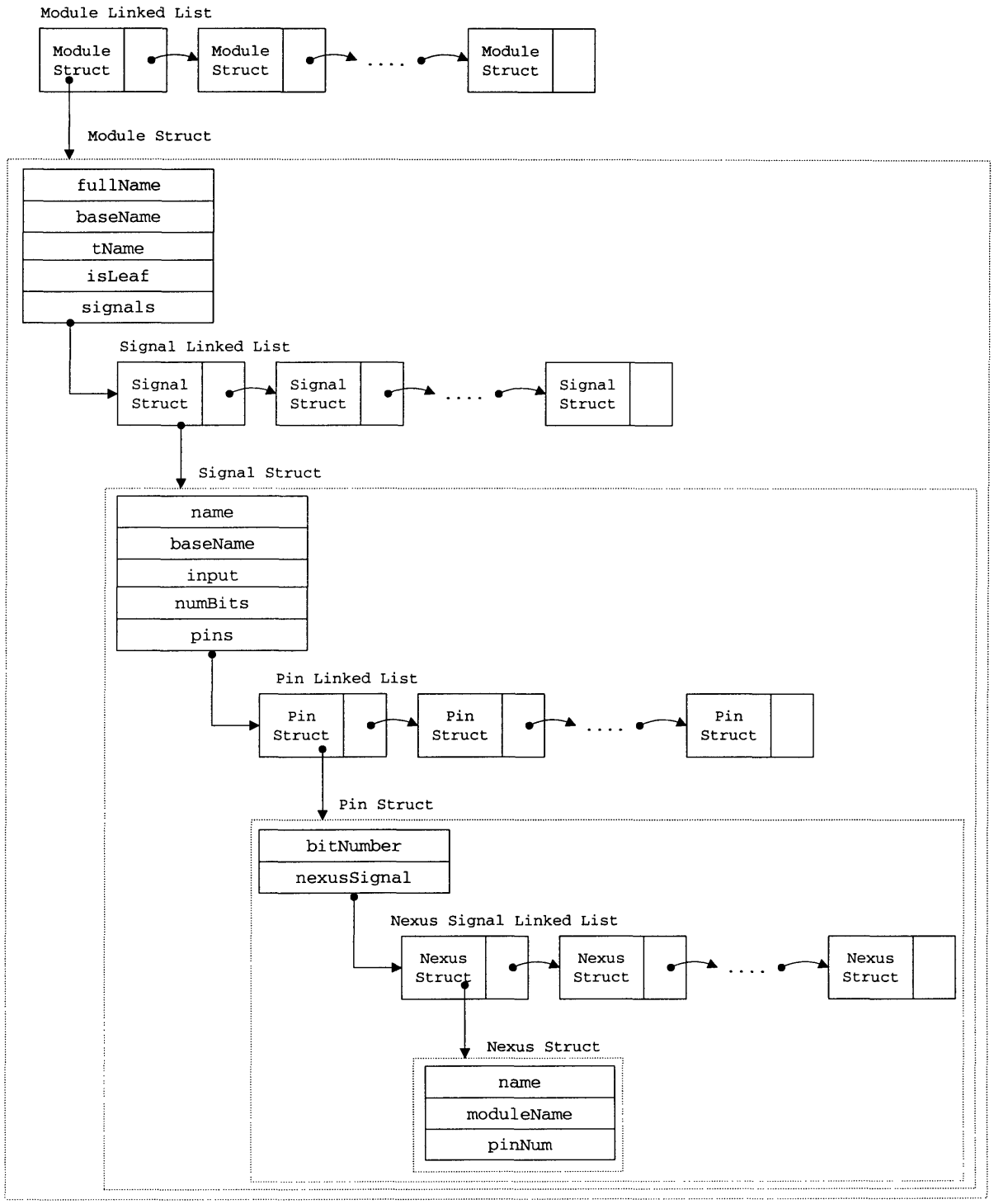

Figure 5-6: Module Linked List Data Structure

pin structs.

5.3.4 Pins

Each pin in a signal is represented by a `pin` struct. A pin struct has two fields. The first field, `bitNumber`, indicates which bit this pin represents in the signal and the second field, `nexusSignals`, is a linked list to `nexus` structs. This represents a list of all other signals that are connected to the same nexus point as this pin of this signal.

5.3.5 Nexus

Each signal that is connected to a nexus is represented by a `nexus` struct with three fields. The `name` field indicates the local name of the nexus signal. The `moduleName` field indicates the hierarchical name of the module this nexus signal belongs to. The final field, `pinNum`, indicates the pin number to nexus connects to for the signal.

5.4 Naming Conventions Check

The first check that is performed is a check whether the names in the design (both module names and signal names) follow the naming conventions specified in the rules. The basic flow of this check is shown in Figure 5-7.

For each module, different functions are called if the module is a leaf or non-leaf. The two paths are identical besides which regular expression it matches for.

5.4.1 Checking Module Names

Both the leaf and non-leaf module name check functions work the same way. The basic skeleton is outlined in Figure 5-8.

First, the correct regular expression is found (either `LeafModuleName` or `NonLeafModuleName`).

Once the expression is found, the expression is sent to a procedure, ReplaceCategories. Because the program's definition allows some special semantics for regular

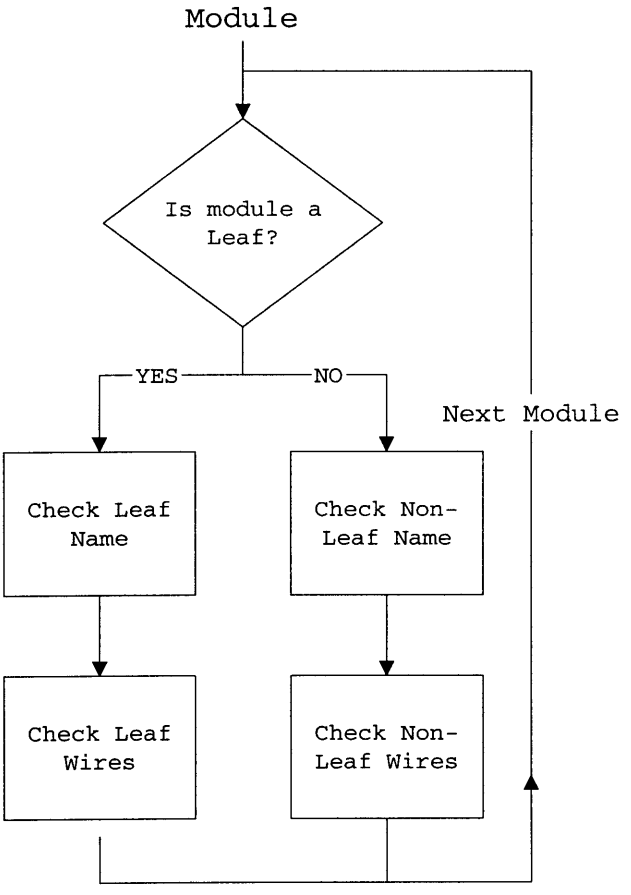

Figure 5-7: Name Check Flow Diagram

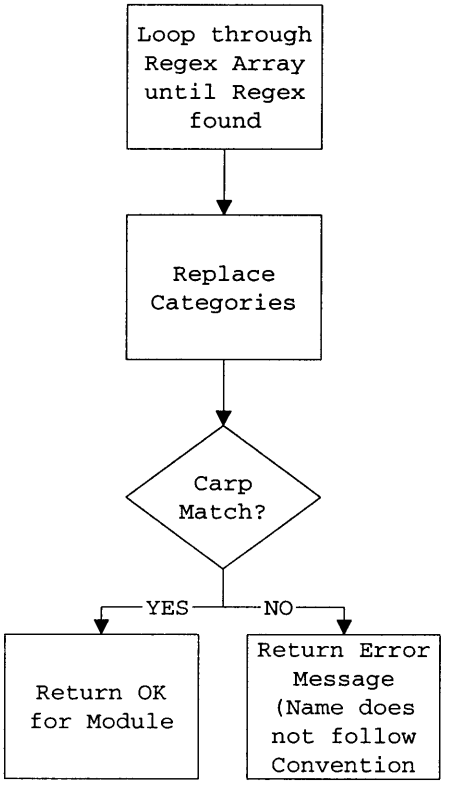

Figure 5-8: Module Name Check Flow Diagram

expression (namely the use of * and allowing categories to be included between forward slashes), the regular expressions need to be cleaned up before they are sent to the Carp expression matcher. Figure 5-9 shows how the function works.

ReplaceCategories first checks if the string input is a *. If it is, it is replaced with the Carp equivalent of .+ and the function returns. If not, regular expression is traversed and each categories are found (done by searching for /.../). When a category is found, the name is retrieved from the expression and is looked up in the `Sets` Array (5.3.1). If it cannot be found, an error is returned. Otherwise, all the members of the sets are substituted into the expression and the category reference is removed. The function returns when there is no more categories to be found. For example, if the module name regular expression is:

```
/LeafCellCategory//LeafCellType/_([0-9]|[a-z]|_)+(_p[0-9]+n[0-9]+)?
```

The returned string from ReplaceCategories is:

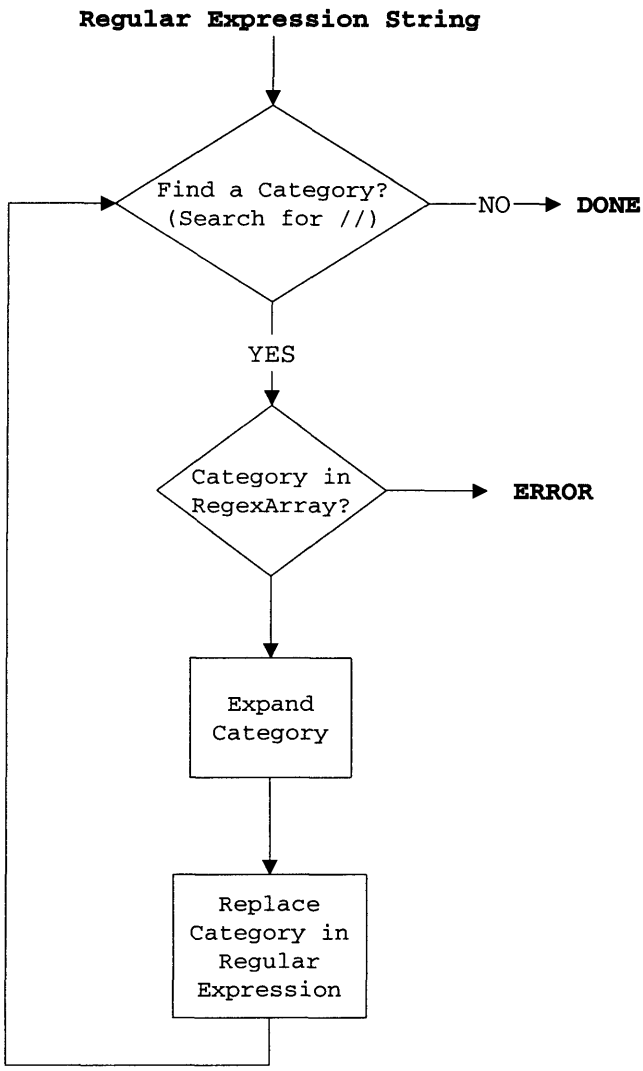

Figure 5-9: Replace Categories Function Flow Diagram

```
(s|d|e|a)(cc|pf|nf|hl|ll|hv|lv|xx)_([0-9]|[a-z]|_)+(_p[0-9]+n[0-9]+)?
```

After the regular expression is cleaned up by ReplaceCategories, the Carp library is used to try to match the name. A new Carp match is defined and a forward match is completed and returns whether the match was successful or not.

5.4.2 Checking Wire Names

Checking wire names on leaf and non-leaf modules is very similar to checking module names. For each module, a regular expression is found and sent to ReplaceCategories to be cleaned up. An additional loop is completed to iterate through all the signals in the modules, and each signal is examined and matched to the regular expression.

5.5 Rules Check

The next check that is performed is checking that the rules defined in the rules file are being followed. The overview of how the rule checking works is presented in Figure 5-10.

All the modules are first iterated through. For each module, every rule is examined and determined if the rule applies to that particular module based on if the module is a leaf module or not and whether the rule applies to leaf modules or not. If the rule does not apply, the next rule is examined.

5.5.1 The Basic Idea

If the rule does apply, the rule table for the rule is traversed. The idea is to try to find one row of the rule array where all the columns "fit". Each node into the array is defined to be a list of strings. What does it mean for a node to "fit"? The meaning of each string in the list at a node is given meaning based on the column header for the node. If the node is in column `i`, refer to the column header in the column header array at index `i`. At that column header node, a regexType, category, and inout value should be defined (though not all of these have to have actual values). Based

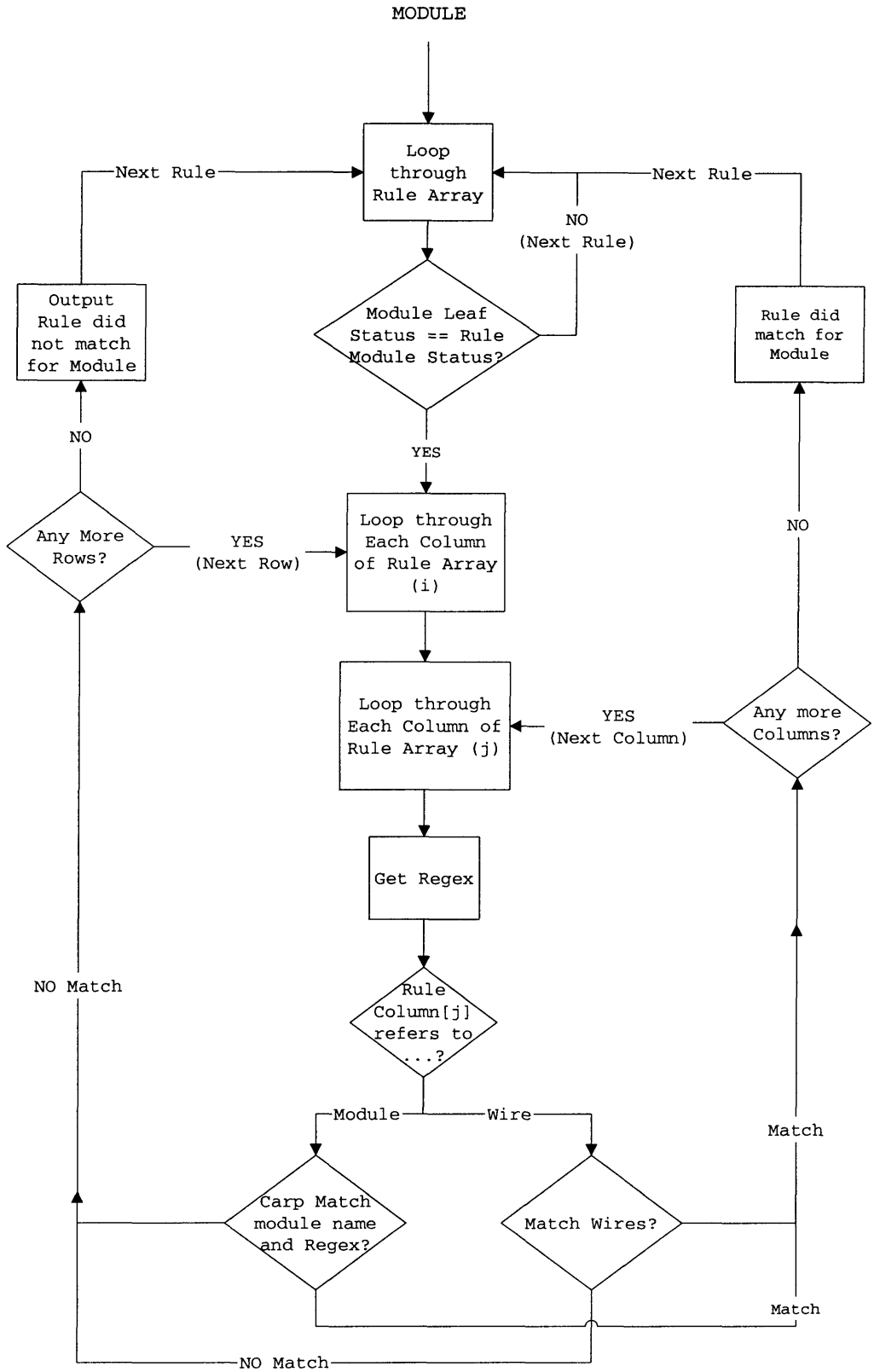

Figure 5-10: Rule Checking Flow Diagram

on this information, a regular expression for that column can be formed. Also, based on the regexType, it can be determined if the name(s) to match to should be the module name or the wire names. Once the names are determined, they are compared via Carp regular expression matching. If there is a match for all the names that need to be examined to the derived regular expression, the node is determined to "fit". If all nodes (columns) in a row fit, the rule has been satisfied for the module. If there is no row where all columns fit, the rule is *not* satisfied.

In order to execute this idea in code, first loop through all rows of the rule array. For each row, loop through all the columns.

5.5.2 How Regular Expressions are Determined

For each column, a regular expression is determined. In order to do this, the process `Get Regex` in Figure 5-10 is expanded to the following diagram (Figure 5-11).

For the rule node that is currently being examined, if the category is NULL, the values in the node should represent an actual Carp regular expression. If the value in the node is a function (defined to be preceded by a %), the regular expression returned is the concatentation of all values that the function returns. If there is no function, the regular expression is simply set to the concatenation of all values in the list at the node.

If the category is not NULL, the values in the node represent values that need to be set for that defined category. If the node is a function, the function needs to be evaluated and the values that are returned by the function become the new values for that node. These returned values are sent to a function that replaces the defined category in the regular expression (regexType for the node) by these values. If the node is not a function, the values for the node replace the defined category in the regular expression directly. The replacement is completed via the function ReplaceCategoriesRestrained. The end result of this unit is the definition of a final regular expression.

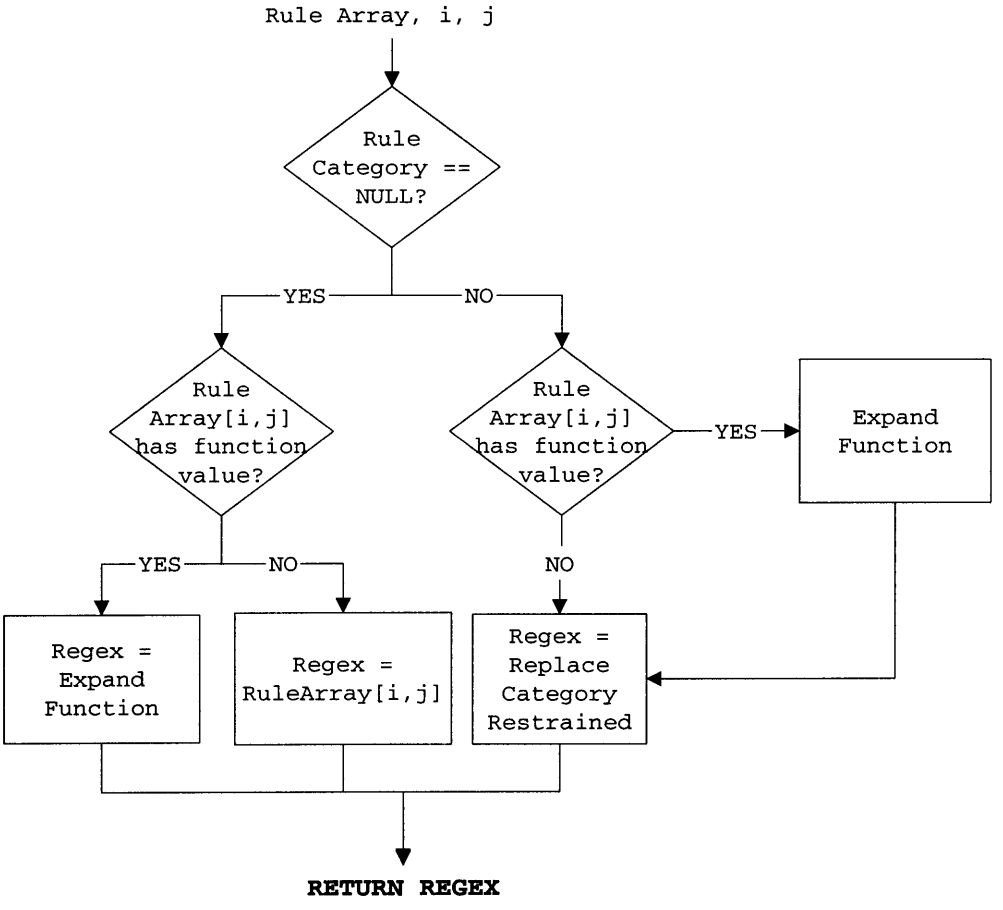

Figure 5-11: Get Regular Expression Flow Diagram

How Functions are Expanded

If a rule array node is a function definition, before the regular expression can be produced, the function must be evaluated for the given rule and module. A function called ExpandFunction performs this task. The flow diagram for this function is shown in Figure 5-12.

First, the function list is traversed to find the matching `function` struct for the function of interest. Next, based on the `function` struct's `input field`'s `regexType`, a matching column is found for the rule where this function was instantiated from. Based on the rule, the column that was just found, and the module, a list of inputs are generated by the function GetValues. The purpose of GetValues is to examine the module and determine what should be the correct inputs to the function. What determines this is the column header for the rule and column that are sent in as inputs. GetValues first finds the regexArray index that matches the regular expression of interest (the regexType of the column previously determined). Given this regular expression, it calls another variant of ReplaceCategories called ReplaceCategoriesCapture. The difference between this variant is an additional input of a category. When ReplaceCategoriesCapture traverses through the regular expression and finds a category, this category is first compared to the input category. If they are equal, the regular expression is modified to capture what this category matches to when Carp is run on the expression at a later time. Specifically, after expanding the category, it adds the string ``{capture,temp}'' immediately after. This is Carp syntax to store into a variable called `temp` the value that matched the element right before the capture statement. The regular expression is set to the returned expression from ReplaceCategoriesCapture. It is then determined if the column header deals with module names or wire names. If it deals with module names, the regular expression is matched to the module name and temp gets set to the value of the category from module name. This is put into a linked list and returned from GetValues. If the header deals with wire names, all the signals are traversed. For each signal, each pin is traversed and for each pin, each nexus signal is traversed. If the leaf status of

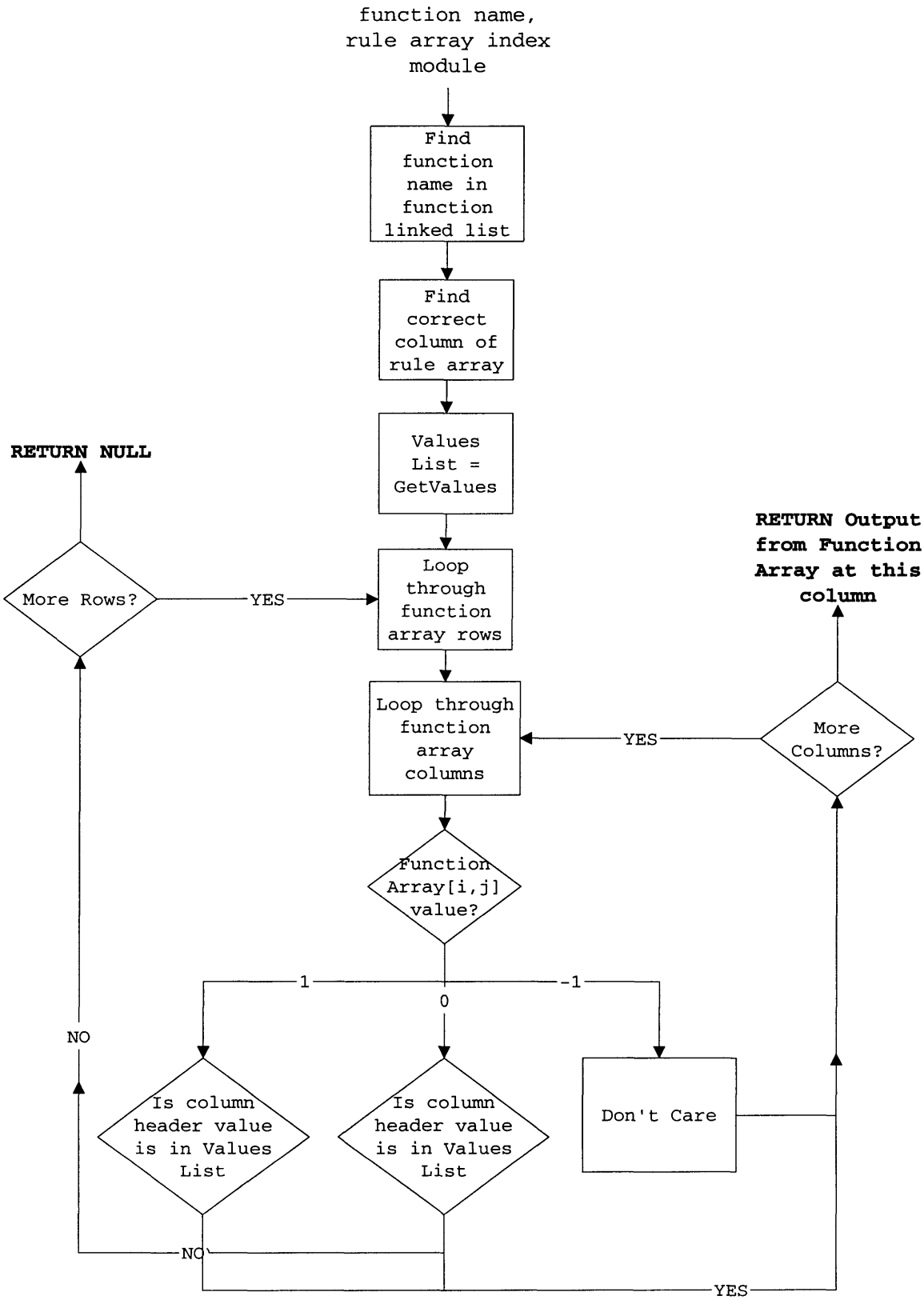

Figure 5-12: Expand Function Flow Diagram

the module which the nexus signal belongs to matches the leaf status of the column header, the signal name is matched against the regular expression using Carp and the value of the category is captured. If this category is not already present in the linked list (initialized to NULL), it is added. Once all signals have been traversed, the linked list is returned.

Take for example a case where the column header for the rule where this function was called from has a regexType of `NonLeafWire`, a category of `SignalSuffix`, and a `inout` value of input. Furthermore, the module that was sent into GetValues has four input signals: `clk_pulse`, `a_np`, `b_p`, and `c_p`. ReplaceCategories would replace the regexType to reflect that the category to capture is `SignalSuffix` and the new regular expression would be:

```
([0-9]|[a-z]|_)+_/SignalSuffix/
```

This would be replaced with the following:

```
([0-9]|[a-z]|_)+_(np|pn|p|n|pulse){capture,temp}
```

The values captured from the inputs would be `pulse`, `np`, `p`, and `p`. The linked list returned would have three elements in it: `pulse`, `np`, `p`.

Once the inputs to the function have been generated, the function array can be traversed. The rows are looped through. For each row, the columns in that row are then traversed. The function array is an array of integers and can contain the values 1, 0 and -1. For the current node, it is determined if the column header for the column of the node is in the input list returned from GetValues. If it is in the input list and the value of the node is a 1, this is considered a match for the node. Likewise, if the column header is not in the input list and the value of the node is a 0, this is also a match. If a match is found, the next column of that row is examined. If there are no columns left, the correct row has been found. The function output for that row is returned as the value of the function expansion. However, if the column header is not in the input list and the node value is 1 or if the column header is in the input list and the node value is 0, a match has not been found. The rest of the row is skipped and

the next row is examined. If there are no rows left, the function failed to match the inputs. This should *not* happen since it would signify a bad input function definition which is not defined for all inputs. If the value is a -1, this is a don't care indication and the next row is examined. If there are no rows left, the correct column has been found and the output for that column is returned.

How Categories are Replaced with Restraints

ReplaceCategoriesRestrained works very much like ReplaceCategories. ReplaceCategoriesRestrained takes two additional inputs: a category to be restrained and the values it should be restrained to. The function still replaces an * with the Carp equivalent of .+. Regular expression are still traversed and categories are found. However, when a category is found, the name first compared against the name of the category that is being constrained. If there is a match, the category is replaced with *only* the values that it should be restrained to. Otherwise, if there is no match, the category is looked up in the `Sets` Array (5.3.1) and all the members of the sets are returned as usual.

For example, the module name regular expression is:

```
/LeafCellCategory//LeafCellType/_([0-9]|[a-z]|_)+(_p[0-9]+n[0-9]+)?
```

A rule may restraint the category `LeafCellType` to be `cc`. The returned string from ReplaceCategoriesRestrained is:

```
(s|d|e|a)(cc)_([0-9]|[a-z]|_)+(_p[0-9]+n[0-9]+)?
```

5.5.3 How to Determine if there is a Match

After the correct regular expression has been resolved, the names to match against must be determined. The column header of the current rule array node is referenced to see whether the column deals with modules or wires.

Dealing with Module Names

If the column deals with modules names, the only name that must be matched is for that module. If the module name matches the regular expression, a match is made. If not, a match is not made.

Dealing with Wire Names

If the column deals with wire names, all the signals must be traversed and matched against the regular expression in order for a match to be made. For each signal, the actual names that are compared are not the signal names, but the nexus signal names for each pin. All nexus signal names need to be checked because a nexus represents a physical connection of actual wires. If a naming convention is to be followed, it must be consistent for all the wires at a connection. The overview for how this section is shown in Figure 5-13.

For each signal, all the pins are looped through. For each pin, all the nexus signals are looped through. If the nexus signal's `inout` status is equal to the `inout` status of the rule array node's column, then this signal name needs to be matched. If a match is not made, a non-match is returned as the result. If a match is made, the next nexus signal is checked. If there are no more nexus signals, the next pin is checked. If there are no pins, the next signal is checked. At the end, if there are no signals (and a non-match has not yet been returned), a match found is returned since all signals have been found to have matched.

5.5.4 How to Determine if Rule is met

Now it is known if this node of the rule array returns a match or not. If it does not produce a match, this row cannot be the right row and the rest of the row can be skipped to go onto the next column. If there are no additional columns, the rule has failed to match, an error message is written, and the next rule is examined. If this node did produce a match, the rest of the row must be checked. If there are no remaining columns, a match has been found for the rule. The next rule is then

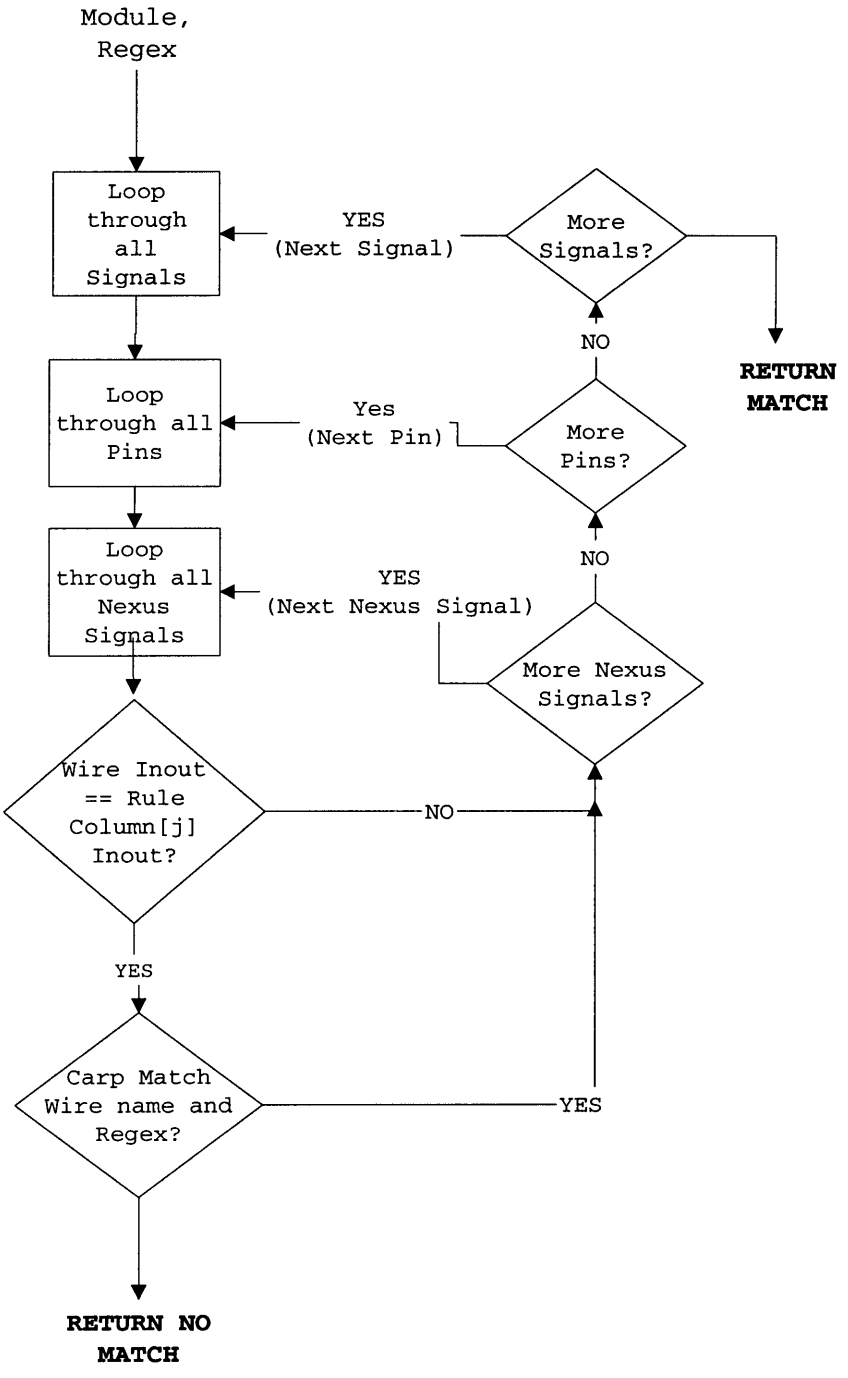

Figure 5-13: Wire Matching Flow Diagram

examined.

5.6 How to Run

The rule check unit is run from the command line as follows:

```
%rulescheck rulefile.xml moduledata.xml
```

The first input is the file name for the rule definition file. The second input is the file name for the module data information file. Error messages are output to the screen and verbose results are appended to *Results.out*.

Chapter 6

Program Validation

Extensive testing and validation was completed on the all parts of this program to ensure functionality. Most of the testing was done with sample files that were created to simulate the desired conditions. Rules that were used were derived from the SCALE-0 document.

Before starting on testing on the actual code written for this thesis, tests were first conducted with the two outside libraries, Carp and libxml. This was done to ensure that their behavior to inputs similar to ones that they would be invoked with in the actual code was correct and followed expectations.

6.1 Unit Testing

For both the *Parser* unit and the *Rules Check* unit, individual unit testing was conducted for all the functions in those two modules.

Black box testing was first done to ensure that the functions behaved as they should based on their specifications. All outcome possibilities were tested and all possible errors that could be generated were invoked by at least one test. Some glass box testing was also done when the functions had deeply nested looping or wide ranching. Different paths were invoked to ensure path completeness and high code coverage.

6.1.1 Parser Unit

The structural checker part of this unit was tested with sample Verilog code. It was tested with a Verilog design that followed structural Verilog style and one that did not. The error messages produced by the latter were checked for accuracy.

During the structural check, the module linked list which stores leaf/non-leaf information about all the modules traversed is populated. This data structure was validated via unit testing and debug step-through to ensure integrity.

The print output unit was also tested with sample Verilog code. It was tested with Verilog designs with multiple modules, each of which had many inputs and outputs. Validation of this function was completed by manually reading the XML file output and checking it for correctness against what should have been produced by the Verilog code. Signal and wire names were checked and nexus connection diagrams were drawn to check the nexus connections were all correct.

6.1.2 Rules Check Unit

The first task this unit performs is to parse both XML input documents. Both documents are parsed into internal data structures. All data structures were examined to ensure that they were being populated correctly. Since *ModuleData.xml* is produced by the *Parser* unit, it is assumed to be in the correct format. However, Rules.xml is checked for any violations in formatting. These violations are outlined in Section 4, at the end of element's individual section. All possible errors were simulated to ensure that the program indeed does catch them when they are made.

All the helper functions in this unit were then tested. This includes functions in the ReplaceCategories series and functions that manipulate data structures. In order to test these, wrapper functions were written around them with simulated inputs. All loops in these functions were tested with inputs which induced varying loop lengths. All possible errors in these functions were run to make sure they were caught.

The two functions that were the crux of this unit were the actual rule check function and the expansion of functions function. The rule check function was simulated

with many various rules, with some where a type was defined for the columns and some where there was not a type defined. The Expand Functions procedure was also tested with a variety of sample functions.

6.2 Integration Testing

Upon satisfactory completion of all unit tests integration testing began using a sample Verilog design for a MIPS processor. The naming conventions for the module names and wire names in this design follow the SCALE-0 conventions. The rules that are checked are also from the SCALE-0 design document. The *Rules.xml* file that was used can be found in Appendix A.

Chapter 7

Conclusion

This thesis presented the design and implemention of a Verilog rules check tool suite. The tool suite was designed to be used during the design of microprocessor chips in order to ensure that rules and conventions that were set early on in the design phase were actually being followed during the implementation phase. The rule check tool suite was implemented in C as two separate units. Though the base of this thesis was the SCALE-0 design, one of the main goals for this design was to have flexibility in allowing more rules and conventions to be added or changed to accommodate other designs in the future. The input file specification helps to achieve this goal by defining a way to describe new rules and conventions. Using this input file, as many rules as needed can be defined using the framework set up by the input file format. All the rules and conventions in the SCALE-0 design can be successfully represented and can be tested on any input Verilog files.

7.1 Future Improvements

The tool suite represents a framework that can be expanded in many ways in the future. While it was designed to be flexible, it was still designed with SCALE-0 design rules in mind and only considers rules that follow the same basic style. For example, it only handles structural Verilog designs. Therefore, the regular expressions that must be defined for module names and wires are based on if they are leaf or non-

leaf entities. This will not work for Verilog designs that follow some other structure. Many other types of rules that may come up in other design documents can be added to this basic framework.

Besides adding more rule formats, other built-in checks can also be added in the future as needed. The current tool only performs static checks on naming conventions and rules. It could be expanded to cover non-static checks. An example of such a check might be checking that a signal, which should become valid before the positive edge of a clock and remain valid until the negative edge, actually follows this timing restraint.

Appendix A

Sample Rules.xml Input File

```
<CheckRules>

<Sets>
        <category name=''LeafCellType''>
                <item>cc</item>
                <item>pf</item>
                <item>nf</item>
                <item>hl</item>
                <item>ll</item>
                <item>hv</item>
                <item>lv</item>
                <item>xx</item>
        </category>
        <category name=''LeafCellCategory''>
                <item>s</item>
                <item>d</item>
                <item>e</item>
                <item>a</item>
        </category>
        <category name=''SignalSuffix''>
                <item>pn</item>
                <item>np</item>
                <item>p</item>
                <item>n</item>
                <item>pulse</item>
        </category>
        <category name=''NonLeafCellCategory''>
                <item>m</item>
        </category>

</Sets>

<Regex>
        <Expression name=''LeafModuleName''>
                 /LeafCellCategory//LeafCellType/_([0-9]|[a-z]|_)+(_p[0-9]+n[0-9]+)?
        </Expression>
```

```
        <Expression name=''NonLeafModuleName''>
                /NonLeafCellCategory/_([0-9]|[a-z]|_)+
        </Expression>
        <Expression name=''NonLeafWire''>
                ([0-9]|[a-z]|_)+_/SignalSuffix/
        </Expression>
        <Expression name=''LeafWire''>
                *
        </Expression>
</Regex>

<Rules>
        <Rule category=''Leaf''>
                <ColumnDefinitions>
                        <Column type=''LeafModuleName''>
                                LeafCellType
                        </Column>
                        <Column type=''NonLeafWire'' inout=''input''>
                                SignalSuffixes
                        </Column>
                        <Column type=''NonLeafWire'' inout=''output''>
                                SignalSuffixes
                        </Column>
                </ColumnDefinitions>
                <Data>
                        <Row>
                                <Column>cc</Column>
                                <Column>*</Column>
                                <Column>%GLB</Column>
                        </Row>
                        <Row>
                                <Column>pf</Column>
                                <Column>np,pn,p</Column>
                                <Column>np</Column>
                        </Row>
                        <Row>
                                <Column>nf</Column>
                                <Column>np,pn,n</Column>
                                <Column>pn</Column>
                        </Row>
                        <Row>
                                <Column>hl</Column>
                                <Column>pn,np,n</Column>
                                <Column>np</Column>
                        </Row>
                        <Row>
                                <Column>ll</Column>
                                <Column>np,pn,p</Column>
                                <Column>pn</Column>
                        </Row>
                        <Row>
                                <Column>hv</Column>
                                <Column>pn</Column>
                                <Column>n</Column>
```

```
                        </Row>
                        <Row>
                                <Column>lv</Column>
                                <Column>np</Column>
                                <Column>p</Column>
                        </Row>
                        <Row>
                                <Column>xx</Column>
                                <Column>pulse</Column>
                                <Column>*</Column>
                        </Row>
                </Data>
        </Rule>
        <Rule category=''Leaf''>
                <ColumnDefinitions>
                        <Column type=''LeafModuleName''>LeafCellCategory</Column>
                        <Column type=''LeafModuleName''></Column>
                </ColumnDefinitions>
                </
                <Data>
                        <Row>
                                <Column>s</Column>
                                <Column>
                                s/LeafCellType/_([0-9]|[a-z]|_)+(_p[0-9]+n[0-9]+)
                                </Column>
                        </Row>
                        <Row>
                                <Column>d</Column>
                                <Column>
                                d/LeafCellType/_([0-9]|[a-z]|_)+(_p[0-9]+n[0-9]+)
                                </Column>
                        </Row>
                        <Row>
                                <Column>e</Column>
                                <Column>
                                e/LeafCellType/_([0-9]|[a-z]|_)+(_p[0-9]+n[0-9]+)
                                </Column>
                        </Row>
                </Data>
        </Rule>
</Rules>

<Functions>
        <Function name=''GLB'' category=''SignalSuffixes''>
                <Input type=''LeafWire'' inout=''input'' category=''SignalSuffix'' />
                <ColumnDefinitions>
                        <Column>pulse</Column>
                        <Column>p</Column>
                        <Column>n</Column>
                        <Column>np</Column>
                        <Column>pn</Column>
                </ColumnDefinitions>
                <Data>
                        <Row>
```

```
                        <Column>1</Column>
                        <Column>*</Column>
                        <Column>*</Column>
                        <Column>*</Column>
                        <Column>*</Column>
                        <Column>pulse</Column>
                    </Row>
                    <Row>
                        <Column>0</Column>
                        <Column>1</Column>
                        <Column>*</Column>
                        <Column>*</Column>
                        <Column>*</Column>
                        <Column>n,p</Column>
                    </Row>
                    <Row>
                        <Column>0</Column>
                        <Column>0</Column>
                        <Column>1</Column>
                        <Column>*</Column>
                        <Column>*</Column>
                        <Column>n,p</Column>
                    </Row>
                    <Row>
                        <Column>0</Column>
                        <Column>0</Column>
                        <Column>0</Column>
                        <Column>1</Column>
                        <Column>*</Column>
                        <Column>np,pn</Column>
                    </Row>
                    <Row>
                        <Column>0</Column>
                        <Column>0</Column>
                        <Column>0</Column>
                        <Column>0</Column>
                        <Column>1</Column>
                        <Column>np,pn</Column>
                    </Row>
                </Data>
            </Function>
</Functions>

</CheckRules>
```

Bibliography

[1] K. Asanovic, C. Batten, S. Laval, and A. Ma. Scale-0 vlsi design. Design document.

[2] J.P. Bergmann and M.A. Horowitz. Vex - a cad toolbox. In *Proceedings for the Design Automation Conference*, 1999.

[3] *IEEE Standard Verilog Hardware Description Language.*

[4] Michael K Montague. web: http://tintware.sourceforge.net/.

[5] Gong Ke Shen. A procedural layout library in java. Master's thesis, Massachusetts Institute of Technology, 2000.

[6] Daniel Veillard. web: http://xmlsoft.org/. The XML C parser and toolkit of Gnome.

[7] Steven Williams. web: http://www.icarus.com/eda/verilog/. Icarus Verilog Development Documents.