Understanding Bug Fix Patterns in Verilog

Sangeetha Sudhakrishnan
sangeetha@soe.ucsc.edu

Janaki T. Madhavan
janaki@soe.ucsc.edu

E. James Whitehead Jr.
ejw@soe.ucsc.edu

Jose Renau
renau@soe.ucsc.edu

University of California, Santa Cruz CA

Abstract

Today, many electronic systems are developed using a hardware description language, a kind of software that can be converted into integrated circuits or programmable logic devices. Like traditional software projects, hardware projects have bugs, and significant developer time is spent fixing them. A useful first step toward reducing bugs in hardware is developing an understanding of the frequency of different types of errors. Once the most common types are known, it is then possible to focus attention on eliminating them. As most hardware projects use software configuration management repositories, these can be mined for the textual bug fix changes. In this project, we analyze the bug fix history of four hardware projects written in Verilog and manually define 25 bug fix patterns. The frequency of each bug type is then computed for all projects. The frequency of bug types in these Verilog projects is compared with similar patterns in Java, allowing a comparison of bug frequencies between hardware and software projects. We find that $29-55\%$ *of the bug fix pattern instances in Verilog involve assignment statements, while* $18-25\%$ *are related to if statements.*

1 Introduction

During the 1990's, the design of integrated circuits shifted from the traditional labor intensive process of schematic layout towards the use of hardware description languages (HDLs), such as Verilog and VHDL. Similar to the tradeoff of early programming languages, HDLs produced less efficient ciruits, but at higher designer productivity. One consequence is that hardware development now has a heavy software development flavor.

Verilog is one of the most widely used hardware description languages. The language supports the design, verification, and implementation of analog, digital, and mixed-signal circuits at various levels of abstraction. Its syntax is similar to the C programming language, has a preprocessor like C and the control flow keywords are similar to C. The language differs from C in the following. It uses Begin/End to specify a block instead of the curly braces. The definition of constants requires a bit width along with their base and Verilog does not have structures, pointers, or recursive subroutines. The concept of time is important in HDLs, and is not found in C. Another major difference from a conventional language is that the statements are not executed strictly linearly.

Just as in software, a hardware project goes through design, development, and testing phases. A designated team of testers usually checks whether the project exhibits the required external behavior. When undesirable behavior is detected, a log is maintained, similar to bug tracking in software projects. This undesirable behavior is called a bug and fixing the bug involves modification of the HDL source code.

Software configuration management repositories are widely used to record the evolution of hardware and software projects. Similar to SourceForge, there is a publically available repository of open source HDL software, www.opencores.org, that contains over 450 HDL projects including arithmetic cores, communication controllers, memory cores, video controllers, and microprocessor cores. As in software projects, changes made to HDL projects include a brief description of the change, recorded in the SCM change log. More often than not, words like "fixed" and "bug" are used to describe the changes related to bug fixes. Hence, these repositories can be mined for log messages containing these words to obtain the bug fix changes. Indeed, it appears that many of the techniques developed to exploit information found in the development history of software projects can be applied to the development history of HDL projects too.

Given this similarity between HDL and high-level programming languages, one question that arises is whether the types and frequencies of bugs in an HDL like Verilog are similar to those in, say, Java. In prior work, Kai Pan manually developed a series of fine-grained bug fix patterns in Java software [1]. These bug fix patterns are automatically extractable from the history of a software project. The bug fix patterns can be viewed as a kind of error taxonomy. Unlike existing work to categorize software errors into specific categories, the bug fix patterns are automatically categorized; prior work involved a human manually performing the categorization work, resulting in poor scalability and unreliable categorizations. This paper reports on an effort to translate this idea of bug fix patterns over to Verilog.

We present a series of 25 bug fix patterns observed in four open source Verilog projects, downloaded from www.opencores.org. The patterns we found were seen to be related to assignments (blocking, non-blocking and assign), "if" statements, module declaration and instantiation, "always" and "switch" statements. We have obtained the frequency of their occurence in the four projects we have analyzed. Errors in assignment were found to be the most common pattern, followed by the "if" related ones. In order to understand them in greater detail, we have characterized them further into many sub-patterns.

The remainder of the paper is organized as follows. Section II discusses the methodology used to obtain the bug fix changes from the change history of the projects analyzed. Section III describes the identified Verilog bug fix patterns in detail. The frequency of the observed patterns in the four projects are presented in Section IV. Section VI discusses the related work in this area. And finally, Section VII presents ideas for future work and concludes the report.

2 Methodology for Identifying Bug Fixes

In this paper, we use the term bug for a mistake made in the source code by the developer, which results in an undesirable external behaviour during the execution of the code. When the developers commit the changes that repair a bug to the software configuration management repository, they usually log a brief note about these changes. The bug fixes are typically denoted by the words "bug" and "fixed". Therefore, by going over the change history, we can obtain the revision that contains the changes that fix a bug (bug fix revision) and the one that has the bug (buggy).

There are two main approaches to get the bug fix revisions and their corresponding buggy revisions. One of the techinques involves looking for references to bug reports; this was introduced by Mockus and Votta [3]. The other one, introduced by Fischer et al [4] and by Cubranic and Murphy [5], involves scanning the change logs for keywords like "fixed" or "bug"; the revisions, whose change logs contain these words are considered to be the bug fix revisions and the revisions prior to them are their respective buggy revisions. We follow the second approach and manually look for the above mentioned keywords in the change logs of the Verilog projects, we analyzed.

Table 1. Projects Analyzed

Project Name	# of Revisions	# of Bugs
Uart_16550	32	128
OC_8051	22	121
OR1K	17	108
Mem_ctrl	45	169

After obtaining the buggy and the bug fix revisions, we compute the changes to each file involved in the latter by doing a "diff" with their counterparts in the former. Adopting the terminology used by [1], we call this textual difference a bug fix hunk pair. A bug hunk refers to the code section in the buggy version, that is modified or deleted in the bug fix revision. The corresponding fix hunk comprises the modifications in the bug fix version that fixes the bug. These two together form the bug fix hunk pair. A bug fix change is typically a modification, addition or deletion of some specific lines of the source code.

To get some insight into the kind of changes developers make to repairs bugs in Verilog, we have looked for syntax-based patterns in the bug fix hunk pairs. The patterns found by us are explained in detail in the next section.

3 Bug fix patterns

To obtain all the bug fix patterns in Verilog, we manually analyzed the four open source projects downloaded from www.opensource.org. They are listed in Table I. By following the methodology described in the previous section, we obtained the bug fix hunk pairs and identified the bug fix patterns present in them. These patterns are grouped into the following major categories: if-related (IF), module-related (MD), assignment (AS), switch (SW), always (AL) and class field (CF).

IF-RELATED (IF)

1. Addition of an if statement (IF_APC): This pattern is observed when an if statement is added to fix a bug. This statement is usually added to ensure that a certain condition is met before performing an operation.

Example:

$$+ \qquad if(rst)\ ma\_issue \leq 1'b0$$

2. Removal of an if statement (IF_RMV): In this pattern, the buggy version has an unnecessary if statement that is removed in the bug fix version. After this fix, the code previously enclosed by the "if" gets executed without requiring any condition to be met.

Example:

$$- \qquad if(rst)\ ma\_issue \leq 1'b0$$

3. Addition of an else branch (IF_ABR): This pattern involves an addition of an else branch to an if statement in the bug fix version to repair a bug. This bug fix is performed when a condition that had been previously ignored has to be considered for the program to exhibit the desired external behaviour.
Example:

$$+ \qquad else\ \{ma\_issue \leq 1'b0$$

$$OR$$

$$+ \qquad else\ if(mst)\ \{ma\_issue \leq 1'b0$$

4. Removal of an else branch (IF_RBR): As the header suggests, in this pattern, an else branch present in the buggy version is removed in the bug fix version, thereby freeing the code in the else body.
Example:

$$- \qquad else\ \{ma\_issue \leq 1'b0$$

$$OR$$

$$- \qquad else\ if(mst)\ \{ma\_issue \leq 1'b0$$

5. Change of if condition expression (IF_CC): In this pattern, the condition logic of the if expression is changed to fix a bug.
Example:

$$\begin{array}{ll} - & if(mst)\ \{ma\_issue \leq 1'b0 \\ + & if(!mst)\ \{ma\_issue \leq 1'b0 \end{array}$$

$$OR$$

$$\begin{array}{ll} - & if(mst)\ \{ma\_issue \leq 1'b0 \\ + & if(mst\ \&\&\ rst)\ \{ma\_issue \leq 1'b0 \end{array}$$

The if related patterns of Verilog are similar to those of any conventional programming language but the remainder of the patterns are more specific to Verilog.

MODULE-RELATED

A Verilog design consists of a heirarchy of modules. Each module defines a set of inputs and output connections called ports. These ports act as connection points with other modules in the system. They are somewhat similar to the formal parameters of a procedure in a high level language. Ports can be of mode *input*, *output* or *inout*. A module receives its input from its input ports and uses the output ports to send out the generated signals. The inout port can at times receive input, at other times generate output but cannot do both simultaneously. We now present the patterns where the bug fixes involve making changes to the modules.

Change in the number of ports in a module (MD_DNP): This bug fix involves changing the number of ports (input and output included), while declaring a module, i.e., the buggy version has more or fewer number of ports in a module declaration than the fix version.
Example:

$$\begin{array}{ll} - & module\ decode(clk, rst, jp, dr, isb)\{ \\ + & module\ decode(\ clk, rst, jp, dr)\{ \end{array}$$

$$OR$$

$$\begin{array}{ll} - & module\ decode(clk, rst, jp, dr, isb)\{ \\ + & module\ decode(clk, rst, dpr, dr, isb)\{ \end{array}$$

Modules in Verilog can incorporate a copy of another module by the process of module instantiation, thus creating a heirarchy of modules. The communication between the parent module and the instantiated one is done through the ports of the latter. We also found some bug fix patterns related to module instantiation. They are listed with examples below.

Module instantiation with different number of ports (MI_DNP): This bug fix consists of a change to the number of ports in the instantiated module.
Example:

$$\begin{array}{ll} - & df\ u2(.clk(clock), .q(q[1])); \\ + & df\ u2(.clk(clock), .q(q[1]), .data(d[1])); \end{array}$$

$$OR$$

$$\begin{array}{ll} - & df\ u2(.clk(clock), .q(q[1]), .data(d[1])); \\ + & df\ u2(.clk(clock), .q(q[1])); \end{array}$$

Module instantiation with different values for the ports (MI_DCP): In this pattern, the number of ports in the instantiated module is the same across the buggy and the fix version but the signals passed through the ports is changed.
Example:

$$\begin{array}{ll} - & dff\ u2(.clk(clock), .q(q[1])); \\ + & dff\ u2(.clk(clock), .q(q[2])); \end{array}$$

ASSIGNMENT-RELATED

There are two types of assignments in Verilog: continuous and procedural. The former is a statement that gets executed every time the right hand side of the assignment changes. It starts with the "assign" keyword. The procedural assignments can be either blocking (operator used: =) or non-blocking (operator used: <=). A blocking statement must be executed before the execution of the statements that follow it in a sequential order, whereas a non-blocking statement allows one to schedule assignments without blocking the procedural flow. We have found bug fix patterns in the continuous as well as the procedural assignments. These patterns are given below.

Addition of "assign" statement (ASG_ADD):
Example:

$$+ \qquad assign\ mux\_out = sel?\ a : b;$$

Removal of "assign" statement (ASG_RMV):
Example:

$$- \qquad assign\ buf\_out = en?\ in :\ 16'bz;$$

Change in "assign" statement expression (ASG_CE):
Example:

$$\begin{aligned} - &\qquad assign\ F = (\ (A\&B) \parallel (C\&D)\ ); \\ + &\qquad assign\ F = (\ (A\&B)\ \&\ (C\&D)\ ); \end{aligned}$$

Addition of a non-blocking assignment (NB_ADD):

Example:

$$+ \qquad buf\_out <= sel?\ a : b;$$

Removal of a non-blocking assignment (NB_RMV):

Example:

$$- \qquad mux\_out <= sel?\ a : c;$$

Change in non-blocking assignment expression (NB_CE):

Example:

$$\begin{aligned} - &\qquad mux\_out <= a; nonumber \qquad (1) \\ + &\qquad mux\_out <= a\&b; \end{aligned}$$

Addition of a blocking assignment (B_ADD):

Example:

$$+buf\_out = sel;$$

Removal of a blocking assignment (B_RMV):

Example:

$$- \qquad mux\_out = sel;$$

Change in blocking assignment expression (B_CE):

Example:

$$\begin{aligned} - &\qquad mux\_out = a; \\ + &\qquad mux\_out = a\&b; \end{aligned}$$

CASE-RELATED

Verilog has a case statement that works exactly like the switch statement of C. The instruction decoder for a CPU is implemented using the case statement. The bug fix pattern observed in this statement is given below. We have grouped addition/deletion of case statements with the addition/deletion/modification of case branches because the incidence of case-related patterns is quite rare.

Addition or removal of case branches (SW_ABRP):

Example (addition of case statement):

$$\begin{aligned} + &\qquad case(\ select\ ) \\ + &\qquad\quad 0 :\ q = d[0]; \\ + &\qquad\quad 1 :\ q = d[1]; \\ + &\qquad\quad 2 :\ q = d[2]; \\ + &\qquad endcase \end{aligned}$$

Example (removal of case statement):

$$\begin{aligned} - &\qquad case(\ select\ ) \\ - &\qquad\quad 0 :\ q = d[0]; \\ - &\qquad\quad 1 :\ q = d[1]; \\ - &\qquad\quad 2 :\ q = d[2]; \\ - &\qquad endcase \end{aligned}$$

Example (addition of a case branch):

$$+ \qquad 3 :\ q = d[3];$$

Example (removal of a case branch):

$$- \qquad 4 :\ q = d[4];$$

Example (change in the body enclosed in a case branch):

$$\begin{aligned} - &\qquad 0 :\ q = d[0]; \\ + &\qquad 0 :\ q = d[0]\&sel; \end{aligned}$$

ALWAYS-RELATED

This structure is Verilog-specific and it is hard to find an equivalent structure in a conventional programming language. As the name suggests, an always block excecutes always, unlike other blocks that execute only once. In addition, the always block has a sensitivity list that tells this block when to execute the code in its body. The bug fix patterns observed in this statement are as follows.

Addition of an always block (AL_ADD):
Example:

$$
\begin{array}{ll}
+ & always\ @(\ a\ or\ sel\ or\ b\ ) \\
+ & \quad begin \\
+ & \quad\quad f = fb\&fd; \\
+ & \quad end
\end{array}
$$

Removal of an always block (AL_RMV):
Example:

$$
\begin{array}{ll}
- & always\ @(\ a\ or\ sel\ or\ b\ ) \\
- & \quad begin \\
- & \quad\quad f = fb\&fd; \\
- & \quad end
\end{array}
$$

Change in the sensitivity list of the always statement (AL_SE):
Example:

$$
\begin{array}{ll}
- & always\ @(\ a\ or\ sel\ or\ b\ ) \\
+ & always\ @(\ a\ or\ sel\ or\ b\ or\ selb)
\end{array}
$$

$$OR$$

$$
\begin{array}{ll}
- & always\ @(\ a\ or\ sel\ or\ b\ ) \\
+ & always\ @(\ a\ or\ sel\ )
\end{array}
$$

CLASSFIELD-RELATED

The roles of wires and registers in Verilog are similar to the class fields in Java. So we call this category classfield-related. The bug fix patterns observed in this category are listed below.

Addition of a register (RG_ADD):
Example:

$$+ \qquad reg\ q;$$

Removal of a register (RG_RMV):

Example:

$$- \qquad reg\ p;$$

Addition of a wire (WR_ADD):

Example:

$$+ \qquad wire[1:0]\ select;$$

Removal of a wire (WR_RMV):

Example:

$$- \qquad wire[3:0]\ d;$$

We have ignored changes related to comments, synthesis statements, include statements, param statements, code cleanup and code formatting. Excluding these, we have classified all the changes observed in the four projects and the results are summarized in Table 2. In the next section, we characterize the bug fix patterns obtained in these projects.

4 Characteristics of Bug Fix Patterns

In this section, we show the frequency of bug fix patterns in the four verilog projects. analyzed. Figure 1 gives the frequencies of all the bug fix patterns across projects.
It is clear that the assignment related patterns are the most common. They account for 29-55% of all observed patterns. This is followed by the if related patterns. They account for 18-25% of the observed patterns. The combined frequency of module declaration and module instantiation is seen to be 4.7-25%. The class field related patterns occur with frequency 4.7-16%. The frequency of the always related patterns is about 2.8-21%. And finally, case related patterns account for only about 1.8-4.6%. Apart from the assignment and the if related patterns, which follow similar trend across all the four projects (most common and second most common respectively), we cannot say anything conclusive about the rest of the projects.

5 Related Work

Bugs in software are difficult to find and costly to fix. Therefore, static code analysis to detect bugs is a very active area of research in software engineering.
The closest work to the work presented in this paper is by [14]. The authors study 4 student hardware design projects and present error distributions. The major difference with the work presented here is that designers were asked to create new revision via CVS whenever a design error was corrected or whenever the design process was interrupted. In this paper, we mine the revision history after the project was created and looked for revision logs containing the words "Bug" or "Fix", thereby considering only changes made to code to correct a design bug. Further, the taxonomy used by Campenhout et al [14] is ambiguous when compared to the taxonomy presented in this paper.
Many techniques have been developed over the years to automatically detect bugs in software. These techniques are usually based on syntactic pattern matching, data flow analysis, type systems and theorem proving. FindBugs [6], JLint [7, 8], ESC/Java 2 [9], PMD [10] are some examples. These tools are applied to find bugs in Java, which has many

CATEGORY	PATTERN NAME	SHORTNAME	UART 16550		OC8051		OR1K		MEM_CTRL	
			#	%	#	%	#	%	#	%
If related	Addition of else related branch	IF_ABR	2	1.56	12	10	9	8.3	6	3.5
	Addition of if branch	IF_APC	6	4.7	2	1.7	0	0	12	7.1
	Change of if condition expression	IF_CC	14	11	9	7.4	6	5.5	20	11.8
	Removal of else branch	IF_RBR	4	3.1	1	0.8	3	2.8	1	0.6
	Removal of if branch	IF_RMV	5	4.6	3	2.5	1	0.92	4	2.3
	TOTAL		31	24	27	22.3	19	18	43	25
Module instantiation	Module instantiation with different number of ports	MI_DNP	2	1.56	4	3.3	7	6.5	3	1.8
	Module instantiation with different values for the ports	MI_DCP	0	0	2	1.65	9	8.3	7	4.1
	TOTAL		2	1.6	6	4.9	16	15	10	6
Module declaration	Module declaration with different number of ports	MD_DNP	4	3.1	5	4.1	11	10	7	4.1
	TOTAL		4	3.1	5	4.1	11	10	7	4.1
Assignment related	Change of expression for non_blocking assignment	NB_CE	21	16.4	9	7.4	6	5.5	5	3
	Change of expression for blocking assignment	B_CE	0	0	4	3.3	1	0.92	6	3.5
	Change of expression for assign statement	ASG_CE	6	4.7	13	10.7	27	25	5	2.95
	Change of expression for assign statement	ASG_CE	6	4.7	13	10.7	27	25	5	2.95
	Assign statement added	ASG_ADD	28	21.8	5	4.1	4	3.7	7	4.1
	Assign statement removed	ASG_RMV	4	3.1	5	4.1	1	0.92	8	4.73
	Non_blocking statement added	NB_ADD	10	7.8	5	4.1	1	0.92	3	1.78
	Non_blocking statement removed	NB_RMV	1	0.8	8	6.6	0	0	0*	1.18
	Blocking statement added	B_ADD	0	0	3	2.5	0	0	7	4.14
	Blocking statement removed.	B_RMV	0	0	1	0.8	0	0	6	3.55
	TOTAL		70	54.6	53	44	40	37	49	29
Switch Case related	Additionor removalofcase branches	SW_ABRP	5	4.6	3	2.5	2	1.9	3	1.77
	TOTAL		5	4.6*	3	2.5	2	1.9	3	1.77
Class field related	Additionofreg	RG_ADD	4	3.1	10	8.26	5	4.7	11	6.5
	Additionof wire	WR_ADD	1	0.8	4	3.3	8	7.4	7	4.14
	Removal of reg	RG_RMV	1	0.8	4	3.3	1	0.92	0	0
	Removalof wire	WR_RMV	0	0	0	0	3	2.8	4	2.366
	TOTAL		6	4.7	18	15	17	16	22	13
Always related	Sensitivity list changes	AL_SE	1	0.8	2	1.65	1	0.92	11	6.5
	Always added	AL_ADD	5	4.6	5	4.1	2	1.9	20	11.83
	Always removed	AL_RMV	4	3.1	2	1.65		0	4	2.36
	TOTAL		10	7.8	9	7.5	3	2.8	35	21
	TOTAL		128		121		108		169	

Table 2. The count and frequency of the extracted pattern instances across projects

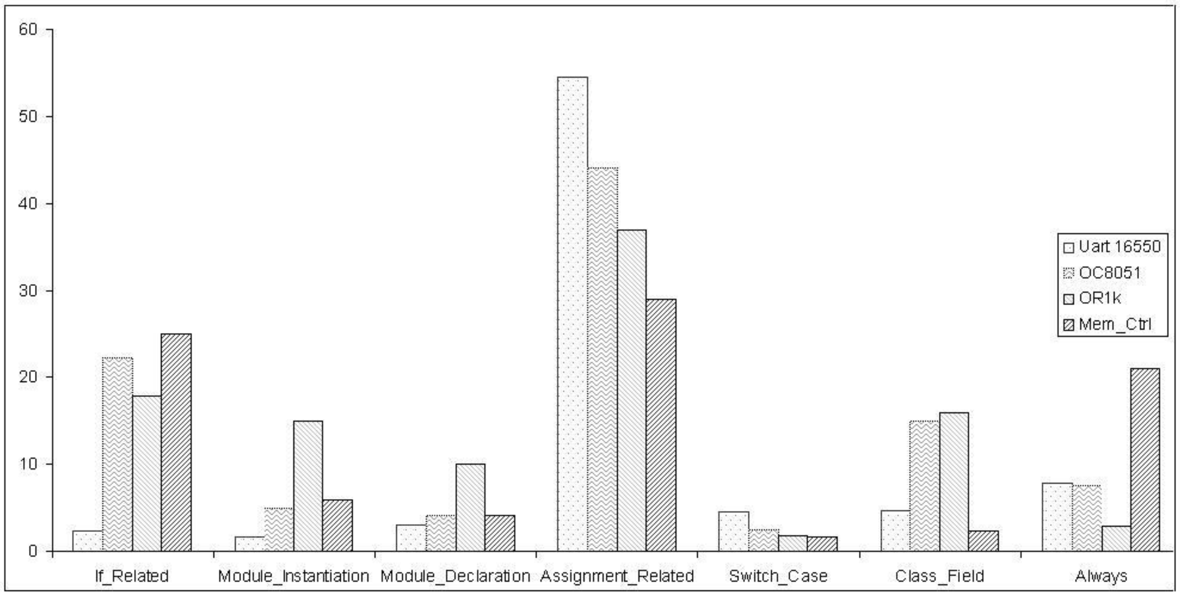

Figure 1. Bar chart of the frequency distribution of bugs in the projects analyzed

potential coding pitfalls [11]. There has also been work that compares the effectiveness of these tools [13]. All these tools are based on static analysis and the bug patterns are based on known program errors. These tools do not use the change history of a software project to find the real bugs reported by the developers.
FindBugs is a tool to detect bug patterns in Java. Its main technique is to analyze the bytecode and syntactically match source code to known suspicious programming practice. This approach is similar to ASTLog [12]. It also uses data flow analysis to look for bugs. JLint is also based on analysing bytecode, performing syntactic checks and dataflow analysis. PMD also performs syntactic checks on program source code and does not have the dataflow component. ESC/Java is based on theorem proving and performing formal verifications of properties of Java source code.
The change history of a software project can give considerable information about the common bugs and the patterns found in their fixes. This information has been exploited to develop an extractor tool to automatically extract 27 bug fix patterns in Java projects [1]. All the tools described above have been developed for Java; to our knowledge, there has been no work on statically analyzing bugg code patterns in hardware description languages. Our approach, based on [1], looks for the bug fix patterns in the software change history of Verilog projects. We have thus categorized all the observed bug-fix related changes into 25 bug fix patterns.

6 Conclusions and Future Work

In this paper, we obtained the bug fix patterns in a hardware description language, Verilog, observing 25 categories of bug fix patterns. Among them, the assignment related patterns are found to be the most common, followed by if related ones. The former accounts for 29-55% of all observed patterns, while the latter accounts for 18-25%. In contrast, the bug fix patterns in Java are dominated by the if statements and the method calls [1]. They account for 45-60.3% of all defined bug fix patterns. The correlation analysis on the extracted pattern instances on the four projects indicates that most of them have similar bug fix pattern distributions.
We could only examine four Verilog projects as our analysis was manual. Also, finding Verilog projects with long history of evolution was difficult. In our future work, we would like to examine more projects to investigate whether the trend found in this paper holds across other projects. If it does, then it should be possible to leverage this pattern knowledge in the creation of static analysis bug finders for Verilog. These patterns can be used to develop bug prediction mechanisms for Verilog. A similar analysis can be done for other hardware description languages to see whether the bug fix patterns are similar to those of Verilog.

References

[1] Pan, Kai, ”Using Evolution Patterns to Find Duplicated Bugs,” Ph.D Dissertation, Computer Science, UC Santa Cruz, 2006. Available at `http://www.soe.ucsc.edu/~ejw/dissertations/pan-dissertation.pdf`

[2] S. Kim, K. Pan, E. J. Whitehead, Jr., *Memories of Bug Fixes*, In Proceedings of the fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE2006), Portland, Oregon, November 5-11, 2006.

[3] A. Mockus and L. G. Votta, *Identifying Reasons for Software Changes Using Historic Database*, In Proceedings of International Conference on Software Maintenance (ICSM 2000), San Jose, California, 2000, pp 120-130.

[4] M.Fischer, M. Pinzger and H. Gall, *Populating a Release History Database from Version Control and Bug Tracking Systems*, In Proceedings of 2003 International Conference on Software Maintenance (ICSM'03), Amsterdam, Netherlands, 2003, pp 23-32.

[5] D. Cubranic and G. C. Murphy, *Hipikat: Recommending Pertinent Software Development Artifacts*, In Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, 2003, pp 408-418.

[6] D. Hovemeyer and W. Pugh, *Finding Bugs is Easy*, ACM SIGPLAN Notices, vol. 39, no. 12, pp 92-106, 2004.

[7] C. Artho, *Finding Faults in Multi-threaded Programs*, Master's Thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001.

[8] JLint, http://artho.com/jlint .

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, *Extended Static Checking for Java*, In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 234-245, Berlin, Germany, June 2002.

[10] PMD/Java, http://pmd.sourceforge.net .

[11] E. Allen, *Bug Patterns in Java*, APress, 2002.

[12] R. F. Crew, *ASTLOG: A Language for Examining Abstract Syntax Trees*, In Proceedings of the Conference on Domain-Specific Languages, Santa Barbara, California, Oct 1997.

[13] N. Rutar, C. B. Almazan and J. S. Foster, *A Comparison of Bug Finding Tools for Java*, in The 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04), pp 245-256, Saint-Malo, Bretagne, France, November 2004.

[14] D. Van Campenhout and H. Al-Asaad and J. Hayes and T. Mudge and R. Brown, *High-level design verification of microprocessors via error modeling*, Trans. Design Automation of Electronic Systems, 3(4):581–599, October 1998.