Detecting and Analyzing Code Clones in HDL

Kyohei Uemura*†, Akira Mori†, Kenji Fujiwara‡, Eunjong Choi*, and Hajimu Iida*
*Nara Institute of Science and Technology, Japan
{uemura.kyohei.ub9@is, choi@is, iida@itc}.naist.jp
†National Institute of Advanced Industrial Science and Technology, Japan
a-mori@aist.go.jp
‡National Institute of Technology, Toyota College, Japan
fujiwara@toyota-ct.ac.jp

Abstract—**In this paper, we study code clones in hardware description languages (HDLs) in comparison with general programming languages. For this purpose, we have developed a method for detecting code clones in Verilog HDL. A key idea of the proposed method is to convert the Verilog HDL code into the pseudo C++ code, which is then processed by an existing code clone detector for C++. We conducted an experiment on 10 open source hardware products described in Verilog HDL, where we succeeded in detecting nearly 1,800 clone sets with approximately 80% precision. We compared code clones in Verilog HDL with those in Java/C based on the metrics to identify the differences among languages. We identified patterns on how code clones are created in Verilog HDL, which include cases for increasing stability and capability of parallel processing of the circuit.**

I. Introduction

Recently, there have been studies on hardware based computation, in addition to software development on general-purpose CPU, for improving efficiency and saving energy consumption [1], [2]. The field-programmable gate array (FPGA) is such hardware that can rewrite the structure of the circuit. In general, the structure of circuits on FPGAs is defined by hardware description languages (HDLs) such as VHDL and Verilog HDL.

The number of logic gates and clock frequency of FPGAs are growing. An FPGA with a large number of gates can constitute a single complex circuit. By improving performances of FPGAs, the usage of FPGAs is expanding. For example, Bing is an internet search engine operated by Microsoft using FPGA clusters [3]. However, the development of complex circuits takes a long time for validation and bug fixes. The growing development period will become more problematic in the future.

HDLs have characteristics similar to general programming languages. Applying software engineering methods to HDLs is important for improving efficiencies of circuit developments. Sudakrishnan et al. investigated the patterns of bug fixes in hardware development projects written in Verilog HDL [4]. Duley et al. proposed a differencing algorithm for Verilog HDL [5]. Analysis of the structures of the HDL code is not studied in depth as far as we know. In this paper, we focus on code clones in the HDL code.

A code clone is a duplicated code fragment in the source code. In general, a code clone is generated by a copy-and-paste operation when a developer adds a new feature similar to an existing feature [6], [7]. Although reusing code fragments facilitates software development, code clones are said to be one of the causes to reduce the maintainability of the source code [8], [9]. Code clone detection techniques [10], [11], [12] and code clone management techniques have been proposed [13], [14], [15].

Our contributions in this paper are as follows. Firstly, we developed a code clone detection method for Verilog HDL which is the most popular HDL. To the best of our knowledge, this is the first reported method on the subject. In the method, we use CCFinderX [10], [16] for detecting code clones. CCFinderX only can process the source code of C/C++, Java, C#, and COBOL. We proposed rules for converting the Verilog HDL code into the pseudo C++ code. Since the pseudo C++ code can be parsed by CCFinderX, CCFinderX can detect code clones in the pseudo C++ code. We conducted an experiment for 10 open source hardware products described in Verilog HDL, where we succeeded in detecting 1,798 clone sets.

Secondly, we analyzed the differences between code clones in general programming languages and Verilog HDL. We studied 10 products written in Verilog HDL, 5 in C, and 5 in Java. By comparing the metrics, we discovered that the code clones in Verilog HDL are longer and more complex than those in C/Java. In addition, the ratio of code clones in Verilog HDL is higher than the ratio in C/Java. We also discovered that the code clone in Verilog HDL are created for mitigating problems peculiar to circuits. For example, bit operations and reset circuits are described by similar statements and blocks.

The rest of the paper is organized as follows. Section 2 describes the overview of Verilog HDL. Section 3 describes the proposed method. Section 4 describes the investigation design. Section 5 shows the results of the experiment. Section 6 discusses the results and threats to validity. Section 7 introduces the related work. Finally, section 8 concludes with future work.

II. Overview of Verilog HDL

A hardware description language (HDL) is a language for defining the structure of circuits. HDLs are typically used for developing FPGAs and application specific integrated circuits (ASICs). Verilog HDL is the most popular HDL.

In Verilog HDL, a module is a base unit to describe a circuit. In hardware development, developers define a module for each

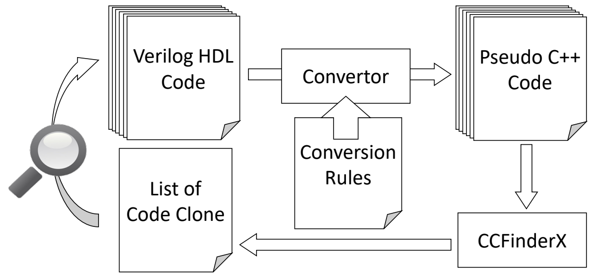

Fig. 1: Overview of Proposed Method

List 1: Sample Code of Verilog HDL

```
 1 module sample (ck, j, k, q, rb, sb );
 2   input ck, j, k, rb, sb;
 3   output q;
 4   reg q;
 5
 6   always @( posedge ck or negedge rb or negedge sb ) begin
 7     if ( rb == 1'b0 )
 8       q <= 1'b0;
 9     else if ( sb == 1'b0 )
10       q <= 1'b1;
11     else begin
12       case ( { j, k } )
13         2'b00 : q <= q;
14         2'b01 : q <= 1'b0;
15         2'b10 : q <= 1'b1;
16         2'b11 : q <= ~q;
17       endcase
18     end
19   end
20 endmodule
```

List 2: Sample Code of Pseudo C++

```
 1 class sample { sample (ck, j, k, q, rb, sb ) {
 2   input ck, j, k, rb, sb;
 3   output q;
 4   reg q;
 5
 6   always @( posedge ck or negedge rb or negedge sb ) {
 7     if ( rb == 1'b0 )
 8       q <= 1'b0;
 9     else if ( sb == 1'b0 )
10       q <= 1'b1;
11     else {
12       case ( { j, k } ) {
13         2'b00 : q <= q;
14         2'b01 : q <= 1'b0;
15         2'b10 : q <= 1'b1;
16         2'b11 : q <= ~q;
17       }
18     }
19   }
20 }};
```

function and connect modules to construct the whole circuit. A module contains definitions of input/output ports, wires, registers, and the behaviors of the circuits. The behaviors of the circuits are described by combinatorial circuits and sequential circuits. A combinatorial circuit is defined by sets of assignment statements and function declarations. A sequential circuit is defined by **always** blocks which describe events that should happen under certain conditions. A module may refer to other modules for specifying circuits that have already been defined.

III. Proposed Method

This section describes our proposed code clone detection method for Verilog HDL. Section III-A gives an overview and Section III-B describes the rules for converting the Verilog HDL code into the pseudo C++ code.

A. Overview

Fig. 1 shows an overview of the proposed method. In the method, the Verilog HDL code is converted into the pseudo C++ code, which is then processed by CCFinderX. CCFinderX parses the source code to identify syntactic units such as functions and statements. However, it does not perform strict parsing to construct the entire abstract syntax trees (ASTs). Therefore, we can use CCFinderX for detecting clones in the pseudo C++ code that may not follow the grammar of C++.

Although there are tools that convert Verilog HDL into C++ [17], [18], they are not appropriate for our approach. These tools are designed for simulating a circuit written in Verilog HDL as a program and does not preserve the structure of the original code. It is not meaningful to detect code clones in the converted code when the original structure is lost. Even if we detect code clones in the converted C++ code, there may not be matching instances in the original Verilog HDL code.

In our proposed method, the Verilog HDL code is converted into the pseudo C++ code in a way that the original structure is preserved. The approach is simpler and easier than developing a parser of CCFinderX for Verilog HDL since the conversion rules can be described by simple regular expressions. We believe that the proposed method can be easily applied to other languages than Verilog HDL.

B. Conversion Rules

Table I shows the proposed conversion rules. A Verilog HDL module is translated to a C++ class constructor whose arguments are the original input/output ports. **Always** and **case** blocks are translated to C++ function definitions and **if** and **else** statements are translated to the similar statements in a straight-forward manner.

List 1 and List 2 show an example of conversion by applying the rules. As we mentioned earlier, the converted pseudo C++ code is not compilable but can be parsed and processed by CCFinderX. At the first line in List 2, a module definition is replaced by a definition of a class and a constructor. CCFinderX recognizes a port list as an argument list of a constructor. A typical definition of an argument consists of the type and the name of the argument. Although the converted code does not contain type information, it does not cause any problem because the CCFinderX parser is not strict. The beginning of the **always** block at the sixth line and the beginning of the **case** block at the twelfth line are regarded as function definitions.

CCFinderX outputs the position information of each clone set and calculates the metrics for overall clone sets. The position information consists of file paths, start token/line numbers and end token/line numbers. The metrics are classified into file metrics and clone metrics. Since our conversion keeps the structure and line numbers of the original code, we can just use the outputs of CCFinderX for inspecting the cloned code regions. This is a very convenient feature of our approach.

The proposed conversion rules only replace the word which will be tokenized by the CCFinderX parser. This means that

TABLE I: CONVERSION RULES

Original Source Code	Converted Source Code
module < ModuleName > (< Port List >) begin < Description of Module > endmodule	class < Module Name > { < Module Name > (< Port List >) { < Description of Module > }};
always @(< Condition >) begin < Description of Always-Statement > end	always @(< Condition >) { < Description of Always-Statement > }
if (< Condition >) < Description of If-Statement > end	if (condition) { < Description of If-Statement > }
else < Description of If-Statement > end	else { < Description of If-Statement > }
case (< Condition >) < Description of Case-Statement > end	case (< Condition >) { < Description of Case-Statement > }

TABLE II: SELECTED VERILOG HDL SUBJECT PRODUCTS

Product Name	#Files	#LoC
mor1kx	48	22,335
ridecore	59	12,726
sd_card_mass_storage_controller	40	15,807
ethernet_tri_mode	43	43,153
nova	52	25,826
jpegencode	21	34,808
sdr_ctrl	16	10,552
hpdmc	22	5,738
aes_highthroughput_lowarea	12	2,362
aes-encryption	15	6,064

the conversion preserves the token level structures. Since CCFinderX is a token-based code clone detector, the proposed conversion does not create code clones which do not exist in the original code, nor misses the code clones which do exist in the original code.

C. Evaluations of the Proposed Method

In this section, we present evaluations of the proposed method. Table II shows the selected products for our evaluations. #Files represents the number of files. #LoC represents the total number of lines including empty lines and comment lines. We chose 9 Verilog HDL products from the Open Cores [1] and one Verilog HDL product from GitHub [2]. The Open Cores is a community for open source hardware and classifies circuits into several categories such as Processor, Communication Controller, Video Controller, Memory Core, and Crypto Core. We chose two products from each category exept for the Processors. For the category of Processors, we chose mor1kx from OpenCores and ridecore from Github. The ridecore is one of the several implementations of RISC-V which is a project for a novel open RISC processor. Companies such as Google, Oracle, and Hewlett Packard participate in this project. In all products, the status of the development is stable.

We detected code clones by three methods including the proposed method. Every method uses CCFinderX.

Method-1 (the proposed method):
Detecting code clones in the converted pseudo C++ code by the C/C++ mode.

Method-2:
Detecting code clones in the original Verilog HDL code by the C/C++ mode.

[1]http://opencores.org
[2]http://github.com

Method-3:
Detecting code clones in the original Verilog HDL code as plaintext by the plaintext mode.

Table III shows the detection results of the above three methods. In the mode of detecting as the C/C++ code, CCFinderX excludes empty lines, comments, and simple definitions such as consecutive variable declarations. LoC refers to the total number of lines. SLoC refers to the number of lines which are the target of detection. In this experiment, we delete empty lines and comment lines. Therefore, the difference between LoC and SLoC represents the number of lines which are excluded by CCFinderX as simple definitions. CLoC refers to the number of lines which have clone fragments. CVR is the ratio of lines including the clone fragments. LEN denotes the number of tokens in the source code. This value is measured by excluding simple definitions recognized by CCFinderX. #Clone set is the number of detected clone set.

The method-1 (the proposed method) detected more clone sets than the method-2. In addition, SLoC and LEN of the method-2 are smaller than those of the other methods. It means that CCFinderX failed to parse the Verilog HDL code and excluded many fragments from the detection target in the method-2. In the method-1, by converting the Verilog HDL code into the pseudo C++ code, CCFinderX could correctly tokenize and detect more code clones.

While detecting as plaintext, CCFinderX detected similar token lists from the entire code including comments and simple definitions. If the detection ability had been equal, the method-3 should have detected more code clones because the method-3 targets simple definitions, too. However, the method-3 has a lower number of detected clone sets and lower coverage (CVR) than the method-1. In tokenizing plaintext, CCFinderX divides texts by characters such as space, '-' and '_'. Thus, an identifier named by snake case is divided into two or more different tokens. Due to tokenization failures, the method-3 has a lower ability of detection than the method-1.

IV. INVESTIGATION DESIGN

This section explains our approach to code clone analysis for Verilog HDL. Section IV-A presents research questions and motivations behind them. Section IV-B shows subjects and procedures of the investigation.

A. Motivations of Research Question

In this study, we conducted an experiment to answer the following research questions.

TABLE III: COMPARISON OF METHODS

Method	LoC	SLoC	CLoC	CVR	LEN	#Clone Set
Method-1 (proposed method)	144,344	110,365	47,680	0.470	721,547	1,798
Method-2 (not converted, as C/C++)	144,344	46,843	23,043	0.523	295,166	1,066
Method-3 (not converted, as plaintext)	144,344	144,344	45,441	0.390	1,634,703	1,610

TABLE IV: FILE METRICS

Metrics	Definition
LEN	The number of tokens in a file.
CLN	The number of clone fragments included in the file.
NBR	The number of files which have clone pairs to other files.
RSA	The ratio of tokens that are clone fragments between other files.
RSI	The ratio of tokens that are clone fragments within only same file.
CVR	The ratio of tokens that are clone fragments.
RNR	The ratio of non repeated code.

TABLE V: CLONE SET METRICS

Metrics	Definition
LEN	The length (number of tokens) of the clone fragment.
POP	The number of clone fragments included in the clone set.
NIF	The number of files which have one or more clone fragments included in the clone set.
RAD	The range of clone fragments in the directory tree.
RNR	The ratio of tokens which are not included in a repeated fragment.
TKS	The number of kinds of the token included in the clone set.
LOOP	The number of loops included in the clone fragment.
COND	The number of conditional branches included in the clone fragment.

RQ1:
What is the difference between general programming languages and Verilog HDL with regard to code clones?

RQ2:
Why are code clones created in Verilog HDL?

RQ1 and RQ2 intend to clarify the characteristics of code clones in Verilog HDL, which is important to determine approaches for supporting code clone management in HDL. For RQ1, we focus on the clone metrics offered by CCFinderX. For RQ2, we examine detected instances to identify the situations where code clones are created.

B. Procedure of Investigation

For answering RQ1, we investigated code clone metrics and file metrics calculated by GemX, a GUI frontend of CCFinderX [19]. Table IV and Table V explain the meaning of the metrics. We focused on the difference of the metrics between Verilog HDL and general programming languages. As a general programming language, we selected two languages with different abstraction levels, C and Java.

TABLE VI: SELECTED C/JAVA SUBJECT PRODUCTS

Language	Product Name	#Files	#LoC
C	git	383	207,480
	How-to-Make-a-Computer-Operating-System	239	18,947
	Linux	23,485	15,870,993
	netdata	53	33,883
	redis	205	115,296
Java	elasticsearch	5,087	893,983
	okhttp	269	70,144
	react-native	583	70,455
	retrofit	134	18,304
	RxJava	602	153,685

For answering RQ2, we analyzed detected code clones in Verilog HDL. For this, we choose instances that are peculiar in terms of differences in the metrics.

For comparison, we collected 10 Verilog HDL products shown in TableII and the top 5 C/Java products in the numbers of stars in GitHub as shown in TableVI.

V. INVESTIGATION RESULTS

In this section, we show the investigation results on the code clone characteristics in Verilog HDL.

A. Quantitative Analysis

Table VII show the file metrics and clone-set metrics, respectively. The values are the averages of the medians of each product.

Let us look at the file metrics for analysis. Verilog HDL has the largest number of clone fragments per file. In addition, Verilog HDL has the highest coverage of code clones occupying files. On the other hand, the NBR values indicate that clone pairs are less distributed across files in Verilog HDL. One should be warned, however, that the result may have been affected by the fact that the Verilog HDL products we chose do not include large numbers of files.

Now let us turn to the clone metrics for analysis. Verilog HDL has the least number of clone sets, but the value of POP (the number of code clones contained in one clone set) is the largest. In addition, Verilog HDL has the largest length in the clone fragment. The COND values that indicate code complexity, in Verilog HDL and C are significantly larger than those in Java.

From these results, it can be said that the code in Verilog HDL products tends to be copied to multiple places in a big chunk across relatively fewer files. In addition, these code clones contain large numbers of conditional branches such as **if** or **case** sentences and hence are logically complex.

TABLE VII: File and Clone Set Metrics

		Verilog HDL	C	Java
File Metrics	LEN	2219.538	2228.777	820.619
	CLN	8.169	4.925	1.771
	NBR	1.107	3.028	1.207
	RSA	0.296	0.089	0.110
	RSI	0.136	0.084	0.078
	CVR	0.387	0.151	0.169
	RNR	0.703	0.919	0.901
Clone Set Metrics	#Clone Set	1798	70658	9279
	LEN	328.819	179.104	188.556
	POP	3.630	2.739	2.806
	NIF	1.539	1.360	1.412
	RAD	0.451	0.382	0.945
	RNR	0.384	0.568	0.669
	TKS	15.927	17.266	15.993
	LOOP	0.220	0.295	0.128
	COND	6.971	5.117	0.601

B. Qualitative Analysis

Let us examine detected code clone instances in Verilog HDL products based on the metrics values in comparison with Java and C to identify frequent code clone patterns in Verilog HDL. In the following sections, we analyze code clone instances classified by syntactical units.

1) Assign statements: Code clones involving assignment statements are frequently found in Verilog HDL. For example, in a code fragment shown in List 3, the rows of consecutive non-blocking assignments are detected as a clone set. An **assign** statement shown in List 4 also appears in different files. These code clones have high POP values and low RNR values. Bit array operations are usually modeled by lists of similar assignments. Developers often copy-and-paste such a code fragment for convenience. However, it can cause defects when variable names are inconsistently changed. In general, one has to edit all clone fragments when the original statements are changed. One may use macros to avoid such inconvenience. In addition, tools for consistent renaming identifiers are proposed [13].

2) Always block: List 5 contains an example of code clones having **always** blocks. An **always** block is a syntactic unit that describes a circuit that is executed asynchronously in accordance with the rising/falling edge of a signal. A circuit for reset is usually defined by an **always** block and is an obvious cause of code clones. These code clones also have high POP values and low RNR values.

3) If/case block: A circuit that changes its output depending upon a state of registers is common. The behavior of such a circuit is usually defined by conditional branches. There are many such code clones involving **if/case** statements as shown in List 6. This explains why the COND values for Verilog HDL products are much higher than those for Java products.

These code clones can be merged by rearranging conditional expressions. However, there is a side effect to it. A degree of parallel processing of the circuit may decrease. Also, the operation of the circuit may become unstable since there may be signals such as the reset signal that should not be used in logical operations with other signals.

List 3: nova/src/bs_decoding.v

```
mvx_H0_diff_a <= (MB_inter_size != 'I8x8)? 0:mvx_CurrMb2[7:0];
mvx_H0_diff_b <= (MB_inter_size != 'I8x8)? 0:mvx_CurrMb2[23:16];
mvx_H1_diff_a <= (MB_inter_size != 'I8x8)? 0:mvx_CurrMb2[15:8];
mvx_H1_diff_b <= (MB_inter_size != 'I8x8)? 0:mvx_CurrMb2[31:24];
mvx_H2_diff_a <= (MB_inter_size != 'I8x8)? 0:mvx_CurrMb3[7:0];
mvx_H2_diff_b <= (MB_inter_size != 'I8x8)? 0:mvx_CurrMb3[23:16];
mvx_H3_diff_a <= (MB_inter_size != 'I8x8)? 0:mvx_CurrMb3[15:8];
mvx_H3_diff_b <= (MB_inter_size != 'I8x8)? 0:mvx_CurrMb3[31:24];

mvy_H0_diff_a <= (MB_inter_size != 'I8x8)? 0:mvy_CurrMb2[7:0];
mvy_H0_diff_b <= (MB_inter_size != 'I8x8)? 0:mvy_CurrMb2[23:16];
mvy_H1_diff_a <= (MB_inter_size != 'I8x8)? 0:mvy_CurrMb2[15:8];
mvy_H1_diff_b <= (MB_inter_size != 'I8x8)? 0:mvy_CurrMb2[31:24];
mvy_H2_diff_a <= (MB_inter_size != 'I8x8)? 0:mvy_CurrMb3[7:0];
mvy_H2_diff_b <= (MB_inter_size != 'I8x8)? 0:mvy_CurrMb3[23:16];
mvy_H3_diff_a <= (MB_inter_size != 'I8x8)? 0:mvy_CurrMb3[15:8];
mvy_H3_diff_b <= (MB_inter_size != 'I8x8)? 0:mvy_CurrMb3[31:24];
```

List 4: mor1kx/rtl/verilog/mor1kx_lsu_cappuccino.v

```
assign next_dbus_adr = (OPTION_DCACHE_BLOCK_WIDTH == 5) ?
       {dbus_adr[31:5], dbus_adr[4:0] + 5'd4} : // 32 byte
       {dbus_adr[31:4], dbus_adr[3:0] + 4'd4}; // 16 byte
```

List 5: aes-encryption/aes_10cycle_10stage/aes_cipher_top.v

```
always @(posedge clk)
 begin
    text_out_stage9[127:120] <= sa00_next_round10;
    text_out_stage9[095:088] <= sa01_next_round10;
    text_out_stage9[063:056] <= sa02_next_round10;
    text_out_stage9[031:024] <= sa03_next_round10;
    text_out_stage9[119:112] <= sa10_next_round10;
    text_out_stage9[087:080] <= sa11_next_round10;
    text_out_stage9[055:048] <= sa12_next_round10;
    text_out_stage9[023:016] <= sa13_next_round10;
    text_out_stage9[111:104] <= sa20_next_round10;
    text_out_stage9[079:072] <= sa21_next_round10;
    text_out_stage9[047:040] <= sa22_next_round10;
    text_out_stage9[015:008] <= sa23_next_round10;
    text_out_stage9[103:096] <= sa30_next_round10;
    text_out_stage9[071:064] <= sa31_next_round10;
    text_out_stage9[039:032] <= sa32_next_round10;
    text_out_stage9[007:000] <= sa33_next_round10;

 end
```

List 6: sdr_ctrl/verif/model/mt48lc8m8a2.v

```
if ((Auto_precharge[0] == 1'b1) && (Read_precharge[0] == 1'b1)) begin
  if ((($time - RAS_chk0 >= tRAS) && // Case 2
     ((Burst_length_1 == 1'b1 && Count_precharge[0] >= 1) ||// Case 1
      (Burst_length_2 == 1'b1 && Count_precharge[0] >= 2) ||
      (Burst_length_4 == 1'b1 && Count_precharge[0] >= 4) ||
      (Burst_length_8 == 1'b1 && Count_precharge[0] >= 8))) ||
      (RW_interrupt_read[0] == 1'b1)) begin // Case 3
        Pc_b0 = 1'b1;
        Act_b0 = 1'b0;
        RP_chk0 = $time;
        Auto_precharge[0] = 1'b0;
        Read_precharge[0] = 1'b0;
        RW_interrupt_read[0] = 1'b0;
        if (Debug) $display ("at␣time␣%t␣NOTE␣:␣Start␣Internal␣
                Auto␣Precharge␣for␣Bank␣0", $time);
  end
end
```

4) Module/file: Files or **module** blocks are frequently cloned in Verilog HDL. This explains why the length of code clones tends to be longer and the coverage of code clones tends to be higher in Verilog HDL products than in C/Java products.

Some of these code clones are created for increasing the degree of parallel processing. However, most of these are created for implementing multiple specifications. In circuit development, similar circuits are often developed simultaneously. For example, circuits with different performance considerations such as the number of the bits or the speed are defined in the same Verilog HDL source files. This is particularly common in FPGA development where we have many variations with different specifications. Because of the low-level features of Verilog HDL for defining raw events in the circuits, it is difficult to merge this type of code clones.

VI. Discussion

This section answers RQs based on the previous investigations. In addition, we discuss threats to validity.

A. Answer to Research Question

1) RQ1:What is the difference between general programming languages and Verilog HDL with regard to code clones?: By comparing the metrics between C/Java products and Verilog HDL products, we could identify several differences. Code clones in Verilog HDL do not spread across a large number of files compared to those in C/Java. On the other hand, clone coverage of Verilog HDL files are higher than those of C/Java files. These results indicate that code clones in Verilog HDL are relatively large and spread within a single file or a small number of files. In addition, clone fragments in Verilog HDL contain numerous conditional branches. The overall trend of the Verilog HDL metrics is similar to C rather than Java. This may reflect the fact that both C and Verilog HDL are used for describing low-level aspects of the system, such as device drivers and circuit constructions, respectively. On the other hand, Java is an object-oriented language capable of abstract description.

By qualitative analysis, we found many code clones in Verilog HDL involving *if* statements where only conditions are different. From software developers' viewpoints, the code should be merged by organizing the conditions. However, if these *if* statements are merged, problems in circuit performance and stability may occur. The problems include increase in circuit footprints, decrease in processing speed, and unintended behaviors. This is an important difference between general programming languages and Verilog HDL. HDLs must deal with not only logical but also physical consequences.

2) RQ2:Why are code clones created?: As the result of observation, the following patterns of code clones were confirmed.

1: Code clones related to assignment statements.
2: Code clones which are **always** block units.
3: Code clones including **if/case** blocks.
4: Code clones which are **module**/file units.

The bit operations are peculiar in digital circuit design and they are repeated in many places in the form of assignment statements. This accounts for the code clones in the first category above. The **always** block specifying asynchronous events are often replicated by modifying conditions in its **if/case** statements to increase degrees of parallel processing and stability of the system. This accounts for the second and the third categories above. Multiple products having different performance requirements are often developed simultaneously. The lack of abstraction support in Verilog HDL forces duplication in the large. This accounts for the last category above.

B. Threats to Validity

Although we have argued the effectiveness of the conversion from Verilog HDL to the pseudo C++, we have not thoroughly evaluated the accuracy of code clone detection. Since no tokens recognized by CCFinderX are added or deleted, the accuracy of the proposed method depends on the ability of CCFinderX. The 1st author manually classified detected code clones and confirmed that 1,466 cases which are 80% of the detection results are actual code clones. The 20% of the detection results are unrelated code fragments where the sequence of grammatical elements happens to be identical. It is the characteristic of token base detection tools such as CCFinderX. In order to evaluate the accuracy, we need an oracle. However, it is hard to determine the complete set of clones as an oracle. Bellon et al. created such an oracle for evaluating multiple detection tools [20]. Roy and Cordy proposed a method to automatically evaluate recall by making an obviously correct answer using mutation/injection [21]. Evaluation of the proposed method, especially the recall, is left for a future work.

In this study, we investigated 10 Verilog HDL products and 10 C/Java products. They may not be sufficient for us to generalize the results reported in the current paper. However, we chose the Verilog HDL products in 5 categories, and the characteristics of Verilog HDL code clones should be well covered. Although the current results are based on observation of code clones, we plan to analyze statistical differences across languages in the future.

In this study, we relied on CCFinderX for code clone detection. CCFinderX can not detect the Type-3 code clone, which is the code fragment extensively modified after duplication. Although we believe that our analysis based on the Type-1 and the Type-2 code clones covers fundamental aspects of code clones in Verilog HDL, analysis of Type-3 code clones may offer new insights. Since our method is applicable to other code clone detection tools that allow non-strict parsing, we may be able to analyze Type-3 code clones in the future.

VII. Related Work

Sudakrishnan et al. investigated the bug fix patterns of the Verilog HDL code [4]. They reported that 29-55% of changes for the bug fix were related to assignment statements and 18-25% related to **if** statements. Their investigation used the bug reports and the change histories of the code. In our investigation, many code clones related assignments and **if** statements are found. It would be interested to investigate whether the code clone affects the bugs. Alternatively, appropriately management for copy and paste may help reduce bugs.

In our proposed method, the Verilog HDL code are converted into the pseudo C++ code and CCFinderX detects code clone from the pseudo C++ code. As a similar method, Roy and Cordy proposed the code clone detection tool NiCAD [22]. NiCAD parses and deforms the original code according to the rules and detects code clones. The rules are described in TXL which is a programming language designed for source code analysis. However, it is not easy to describe rules in TXL.

Our proposed method can detect code clones using CCFinderX and the conversion rules which consists of only a few regular expressions. This method can be applied quickly and easily to other languages.

Kapser and Godfrey classified intentions of code clones in software and analyze their advantages and disadvantages [23]. We plan to such classification of code clones in the Verilog HDL code as future work.

VIII. Conclusion

In this paper, we investigated code clones in Verilog HDL. Investigation of code clones in HDL had not been reported as far as we know.

We have proposed a simple but effective method for detecting code clones in Verilog HDL. To the best of our knowledge, this is the first reported method on the subject. The method converts the Verilog HDL code to the pseudo C++ and employs CCFinderX for detecting code clones in its C/C++ mode. Since the method does not add or delete any tokens recognized by CCFinderX, the detection capability genuinely depends on the ability of CCFinderX to detect C++ code clones. We have conducted experiment on 10 open source Verilog HDL products to demonstrate the effectiveness of the method. Since the proposed method is not limited to Verilog HDL, it enables code clone analysis for the languages not supported by existing tools.

We have compared code clones in Verilog HDL products with those in C/Java products based on the metrics to identify differences between them. Although the overall trend in the metrics for Verilog HDL products is similar to that of C, Verilog HDL products demonstrate a higher coverage of code clones than C/Java products.

We have noticed that code clones in Verilog HDL reflect how circuits are developed in the language. For instance, a bit string operation is often defined by a sequence of similar assignment statements, and reset circuits and input dependent circuits are often defined by cloning an **always** block and an **if/case** block, respectively. We have observed many cases of file/module cloning, which are the result of developing multiple products with similar, yet different specifications. We believe that the tool support for consistent modification over clone sets is also useful for Verilog HDL. On the other hand, we have noted that aggregating clone sets in Verilog HDL must take into consideration the trade-offs between computational parallelism and circuit footprints. We plan to work on this issue in the near future.

The future work also includes measuring recall of the proposed method. We plan to employ the clone mutation injection method [21] for this purpose. Regarding the fact that the proposed method cannot detect Type-3 code clones, we plan to employ methods based on abstract syntax trees.

Acknowledgment

The second author is supported by JSPS KAKENHI Grant Number 26280025.

References

[1] K. Neshatpour, M. Malik, and H. Homayoun, "Accelerating Machine Learning Kernel in Hadoop Using FPGAs," in *Proc. of CCGrid'15*, 2015, pp. 1151–1154.

[2] N. Katayoun, M. Maria, G. Mohammad, Ali, and H. Houman, "Energy-Efficient Acceleration of Big Data Analytics Applications Using FPGAs," in *Proc. of BigData'15*, 2015, pp. 115–123.

[3] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *Proc. of ISCA'14*, 2014, pp. 13–24.

[4] S. Sudakrishnan, J. Madhavan, E. J. Whitehead, Jr., and J. Renau, "Understanding Bug Fix Patterns in Verilog," in *Proc. of MSR'08*, 2008, pp. 39–42.

[5] A. Duley, C. Spandikow, and M. Kim, "Vdiff: a program differencing algorithm for Verilog hardware description language," *Automated Software Engineering*, vol. 19, no. 4, pp. 459–490, 2012.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proc. of ICSM'98*, 1998, pp. 368–377.

[7] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," in *Proc. of ISESE'04*, 2004, pp. 83–92.

[8] A. Zeller, *Why Programs Fail*. Morgan Kaufmann Publishers, 2005.

[9] B. S. Baker, "Finding Clones with Dup: Analysis of an Experiment," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 608–621, 2007.

[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[11] Z. Li, S. Myagmar, S. Lu, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on Software Engineering*, vol. 32, no. undefined.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *Proc. of ICSE'07*, 2007, pp. 96–105.

[13] P. Jablonski and D. Hou, "CReN: A Tool for Tracking Copy-and-paste Code Clones and Renaming Identifiers Consistently in the IDE," in *Proc. of OOPSLA Eclipse'07*, 2007, pp. 16–20.

[14] H. A. Nguyen, T. T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen, "Clone Management for Evolving Software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008–1026, 2012.

[15] E. Choi, N. Yoshida, and K. Inoue, "An Investigation into the Characteristics of Merged Code Clones during Software Evolution," *IEICE Transactions on Information and Systems*, vol. E97.D, no. 5, pp. 1244–1253, 2014.

[16] "CCFinder Official Site," http://www.ccfinder.net/ccfinderxos.html.

[17] W. Stoye, D. Greaves, N. Richards, and J. Green, "Using RTL-to-C++ translation for large soc concurrent engineering: a case study," *Electronics Systems and Software*, vol. 1, no. 1, pp. 20–25, 2003.

[18] " Verilog2C++ ," http://verilog2cpp.sourceforge.net/.

[19] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: maintenance support environment based on code clone analysis," in *METRICS'02*, 2002, pp. 67–76.

[20] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[21] C. K. Roy and J. R. Cordy, "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools," in *Proc. of ICSTW*, 2009, pp. 157–166.

[22] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *Proc. of ICPC*, 2008, pp. 172–181.

[23] C. J. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.