On the Naturalness of Hardware Descriptions

Jaeseong Lee*
UT Austin (USA)
jason.lee27@utexas.edu

Pengyu Nie*
UT Austin (USA)
pynie@utexas.edu

Junyi Jessy Li
UT Austin (USA)
jessy@austin.utexas.edu

Milos Gligoric
UT Austin (USA)
gligoric@utexas.edu

ABSTRACT

Mining software repositories (MSR) has been shown effective for extracting data used to improve various software engineering tasks, including code completion, code repair, code search, and code summarization. Despite a large body of work on MSR, researchers have focused almost exclusively on repositories that contain code written in imperative programming languages, such as Java and C/C++. Unlike prior work, in this paper, we focus on mining publicly available hardware descriptions (HDs) written in hardware description languages (HDLs), such as VHDL. HDLs have unique syntax and semantics compared to popular imperative languages, and learning-based tools available to hardware designers are well behind those used in other application domains. We assembled large HD corpora consisting of source code written in several HDLs and report on their characteristics. Our language model evaluation reveals that HDs possess a high level of naturalness similar to software written in imperative languages. Further, by utilizing our corpora, we built several deep learning models for automated code completion in VHDL; our models take into account unique characteristics of HDLs, including similarities of nearby concurrent signal assignment statements, in-built concurrency, and the frequently used signal types. These characteristics led to more effective neural models, achieving a BLEU score of 37.3, an 8–14-point improvement over rule-based and neural baselines.

CCS CONCEPTS

• Hardware → Hardware description languages and compilation; **• Software and its engineering → Software maintenance tools**.

KEYWORDS

Naturalness, Hardware Description Languages, VHDL, Verilog, SystemVerilog, Natural Language Processing, Code Completion

ACM Reference Format:
Jaeseong Lee, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2020. On the Naturalness of Hardware Descriptions. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368089.3409692

*The first two authors led this work with equal contributions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409692

1 INTRODUCTION

Mining software repositories (MSR) has been an active area of research for over a decade. Researchers and practitioners have been mining various artifacts available in (or associated with) repositories, e.g., code, license files, images, natural language elements, pull requests, open issues, and have shown that MSR can be effective for extracting data that is valuable for numerous software engineering tasks, e.g., [5, 9, 17, 24, 35].

Recently, the extracted data from software repositories was used to show that *source code is natural*, i.e., it is repetitive and predictable. The pioneering paper on the topic [17] found that the level of naturalness in Java code is even higher than that of English, and built a simple code completion tool based on an n-gram language model. A follow-up work [44] revisited the original results and explored naturalness of code written in several programming languages, including C, C#, Java, Python, Ruby, and Scala. The naturalness of source code has led to the advancement of powerful statistical models useful in a wide range of tasks, including code completion [27], code repair [4], code search [56], and code summarization [26]. However, despite some great progress on MSR in general and the naturalness of software in particular, *researchers have focused almost exclusively on general purpose (imperative) programming languages*, e.g., Java and C/C++ [2].

In this paper, we present the first work on *mining hardware descriptions* (HDs)[1] written in one of the three popular hardware description languages (HDLs): VHDL, Verilog, SystemVerilog. HDLs have unique syntax and semantics compared to popular imperative languages, e.g., they are data-flow programming languages where many statements are executed simultaneously. Moreover, HDs have unique properties, e.g., most of the signals are logic bits or vectors of logic bits, and the similarity of nearby statements is high.

We hope that our work will inspire more research on improving engineering practices and tools used by hardware designers, a different target audience than developers using traditional imperative programming languages. This improvement in development practices and tools is very much needed; the state-of-the-art in hardware description is well behind practices and tools used by software engineers. For example, even the most advanced Integrated Development Environment (IDE), Sigasi [48], is well behind IDEs for Java in terms of adopting learning-based code completion and code checking techniques. Thus, hardware designers need help right now to speed up their development, enforce coding conventions [25, 32, 47, 61], and ensure correctness of their HDs. We make several contributions to start off the research in this direction.

First, we *mine* publicly available repositories on GitHub to extract the most popular (in terms of the number of stars) repositories that contain HDs for VHDL, Verilog, and SystemVerilog. We then

[1]Hardware descriptions == *software* written in hardware description languages; they are also frequently referred to as hardware designs or hardware design models.

analyze these repositories to extract various metrics of the code that characterize the languages. The creation of this large corpus of HDLs stands to motivate other researchers to study and improve available HDs.

Second, we conduct a comparative evaluation of the *naturalness of HDs* written in aforementioned languages against software written in imperative languages by building language models over the mined corpora and report standard cross entropy measures [17] on the statistical regularity of HDs. Our results show that HDs have a high level of naturalness similar to software written in imperative languages like Java, which should motivate researchers and practitioners to develop code automation techniques based on (deep) learning for HDLs.

Finally, we develop the first learning-based models for HDLs utilizing the extracted data. We design several deep learning models for automatically *completing the right hand side of a concurrent signal assignment statement* (concurrent assignment for short) in VHDL; concurrent assignments are among the most common components for data-flow and structural descriptions in HDs. Our models are designed to exploit unique properties of HDs and HDLs, including similarities of nearby concurrent assignments, in-build concurrency, and a relatively small number of frequently used signal types. We reveal that these properties contribute to the performance of our most advanced model, measured in terms of accuracy and BLEU [38], which are metrics commonly used in the related literature [18].

In summary, we make the following key contributions:

- ⋆ We release a large corpus for learning over HDs, mined from GitHub. This corpus can be used by researchers, practitioners, and tool developers to improve engineering practices in this important domain.
- ⋆ We perform the first study of the naturalness of HDs written in three popular languages: VHDL, Verilog, SystemVerilog. By training and testing statistical language models, we find that HDs written in VHDL are more natural than software written in Java, and HDs written in other HDLs also have a high level of natural regularities in code.
- ⋆ We design and implement deep learning models for predicting the right hand side of concurrent assignments in VHDL. We extensively evaluate our models using 100 popular repositories. Our best model achieves 48.0% accuracy and 37.3 BLEU, which outperforms rule-based and neural baselines by 8–14 BLEU.

Our code and data is available on GitHub: https://github.com/EngineeringSoftware/hdlp.

2 BACKGROUND

This section provides a brief background on hardware description languages (HDLs) and introduces several VHDL [3] constructs via an example. We focus on VHDL due to our familiarity with the language. Because it is not feasible to describe all aspects of VHDL, we focus on the constructs required in later sections.

Hardware designers use HDLs to write *hardware descriptions* (HDs) that describe logic circuits on the Register Transfer Level (RTL); RTL is a design abstraction of the flow of digital signals between hardware registers. An HD is commonly tested and debugged in an event-driven simulation program. Once the HD is

```
1  entity fpga64_sid_iec is
2          port(...
3                  clk32 : in std_logic;
4                  uart_txd : out std_logic;
5                  uart_rts : out std_logic;
6                  uart_dtr : out std_logic;
7                  uart_ri_out : out std_logic;
8                  uart_dcd_out: out std_logic; );
9  end fpga64_sid_iec;
10 architecture rtl of fpga64_sid_iec is ...
11         signal cia2_pao: unsigned(7 downto 0);
12         signal cia2_pbo: unsigned(7 downto 0);
13         signal vicAddr: unsigned(15 downto 0); ...
14 begin
15 -- setting local signals, etc., not shown due to space constraints
16       process(clk32)
17       begin
18           if rising_edge(clk32) then
19               if sysCycle = sysCycleDef'high then
20                   sysCycle <= sysCycleDef'low;
21               elsif sysCycle = CYCLE_CPU6 then
22                   sysCycle <= CYCLE_CPU8;
23               else
24                   sysCycle <= sysCycleDef'succ(sysCycle);
25               end if;
26           end if;
27       end process;
28       iec_data_o <= cia2_pao(5);          \
29       iec_clk_o <= cia2_pao(4);            |
30       iec_atn_o <= cia2_pao(3);            |
31       uart_txd <= cia2_pao(2);             |
32       uart_rts <= cia2_pbo(1);              > concurrent assignments
33       uart_dtr <= cia2_pbo(2);             |
34       uart_ri_out <= cia2_pbo(3);          |
35       uart_dcd_out <= cia2_pbo(4);         |
36       vicAddr(14) <= (not cia2_pao(0));    |
37       vicAddr(15) <= (not cia2_pao(1));   /
38 end architecture;
```

Figure 1: An example VHDL code snippet from MiSTer-devel/C64_MiSTer repository, which is a part of our corpus.

completed, it is processed by a synthesis program and translated onto a programmable logic device, e.g., FPGA.

We show an example of an HD written in VHDL in Figure 1. (Syntax of VHDL is the closest to the Ada programming language.) We extracted this example from MiSTer-devel/C64_MiSTer repository, which aims to recreate Commodore 64 using modern hardware [33]. This project is publicly available on GitHub (SHA: `0efb9e1b`) [34] and is part of our corpus that is used in later sections.

In VHDL, each HD consists of one or more *modules*; only one module is shown in Figure 1. For each module, there is code that describes its interface (i.e., what comes into the module and comes out of the module) and behavior (i.e., how values on the output depend on the inputs).

An *interface* starts with the keyword `entity` (line 1) followed by the name of the module (`fpga64_sid_iec`); the interface ends with the keyword `end` followed by that module name (line 9). An interface defines *ports* (lines 2–8) available on the module. Each port has a name (e.g., `uart_txd`), the direction that information is allowed to flow through the port (`in`, `out`, or `inout`), and the type of the port (e.g., `std_logic`) that defines the set of values that can flow through the port.

A *behavior* starts with the keyword `architecture` (line 10) followed by the name for the architecture (`rtl`), the keyword `of`, and then the name of the module for which the behavior is specified; a sequence of keywords `end architecture` (line 38) closes the

behavior section. At the beginning of each architecture (lines 11–13), a designer can define an arbitrary number of local signals that can be helpful when defining the behavior; each signal definition starts with keyword `signal`, followed by the signal's name (e.g., `cia2_pao`) and type (e.g., `unsigned(7 downto 0)`, which is an unsigned 8 bit value). Between `begin` and `end architecture` (lines 15–38), there can be an arbitrary number of concurrent processes. In our example, there is one explicit process (lines 16–27). This process is executed whenever the input `clk32` signal changes. The statements within the process (lines 18–26) are executed sequentially. The process in our example is followed by several *concurrent assignments* (lines 28–37). Each concurrent assignment is an implicit process (i.e., w/o the `process` keyword) with a single statement. *All processes execute in parallel.* Each concurrent assignment is executed whenever any of the signals on its right hand side is changed. For example, on line 28, whenever `cia2_pao` is changed, the concurrent assignment is executed and a new value for `iec_data_o` is computed.

VHDL is strongly typed. Unlike VHDL, Verilog is a weakly-typed language and its syntax is inspired by C and Fortran. However, the base structure and language constructs available in Verilog are similar to those in VHDL. It is commonly accepted that VHDL is more verbose than Verilog. SystemVerilog was initially introduced as an extension of Verilog with object-oriented features and the goal to improve verification of HDs; the most common usage of SystemVerilog is to write code for verifying HDs written in VHDL and Verilog. (We do not show code snippets for Verilog and SystemVerilog as that would take too much space.)

In Section 3, we mine GitHub to find and characterize available HDs written in VHDL, Verilog, and SystemVerilog. We then, in Section 4, study the naturalness of HDs written in HDLs, which have substantially different syntax and semantics than imperative programming languages studied in the past. Finally, in Section 5 we develop deep learning models to predict the right hand side of concurrent assignments in VHDL, e.g., on line 30 in Figure 1 we want to predict `cia2_pao(3)`. We focus on VHDL but our models are generalizable to other HDLs. Our learning-based models are designed based on our intuition that (1) nearby assignment statements are similar and provide important local context, (2) types of ports and local signals provide important global context, and (3) the order of concurrent assignments, based on the parallel execution semantics, is irrelevant. We will show that these observations contribute to the performance of our models.

3 HARDWARE DESCRIPTION CORPORA

In this section, we describe the procedure that we followed to assemble the corpora of HDs; these HDs are used in later sections to study naturalness (Section 4) and train/evaluate models for completing the right hand side of concurrent assignments (Section 5). Moreover, we provide statistics for several features unique to HDLs in the assembled corpora, which could motivate other research directions. In addition to the HD corpora, we obtained two Java corpora in order to compare the naturalness of HDs with that of Java code; we followed prior work on software naturalness to obtain the Java corpora. We obtained all repositories from GitHub, which is the most popular repository hosting service. We downloaded all the repositories on Oct 4, 2019.

HD Corpora. First, we created a corpus for each HDL using the top 100 repositories from GitHub, excluding forked repositories, and ranked by the number of stars (we used the number of stars as a proxy for projects' popularity following recent literature [20, 36]). The number of stars for the VHDL repositories in our corpus is, on average, 129.4 and ranges from 17 to 979; the number of stars for the Verilog repositories in our corpus is, on average, 206.8 and ranges from 61 to 1217; the number of stars for the SystemVerilog repositories in our corpus is, on average, 73.2 and ranges from 28 to 746. Second, we processed all the files in all the repositories to find the set of files that can be parsed by the open-source parsers generated using the ANTLR parser generator [39]. We observed that some files that have a correct file extension (e.g., `.vhd`) are not parsable because they do not actually contain valid code but rather binary data (e.g., RSA keys and data blocks), for example, the file blk_mem_gen_v8_4_vhsyn_rfs.vhd from Xilinx/PYNQ-DL project or they use latest language standards not supported by available tools. We also filtered out duplicate files. Duplicate files are present in repositories with imperative languages too, and they can introduce noise in experimental results [1].

Java Corpora. In order to compare our findings with prior work that studied Java repositories, we use two Java corpora: (1) Java (Naturalness), which contains exactly the same repositories and revisions as Hindle et al. [17]'s study, and (2) Java (Popular), which contains the most popular 10 repositories at the time of our experiments. We used only 10 repositories in the Java (Popular) corpus in order to match the number of repositories used in the Java (Naturalness) corpus, and to remain close to the number of tokens in the HDLs corpora. We used the same procedure as for HDLs to clean duplicate files from the Java corpora; we used Eclipse JDT Core version 3.12.2 [11] to process Java files.

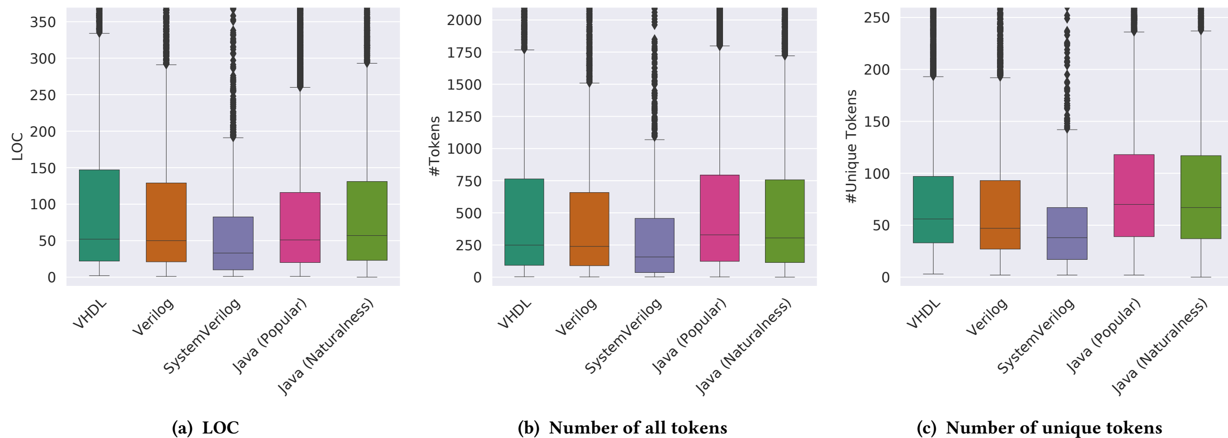
Statistics. Table 1 shows the statistics of the HD and Java corpora. The first column shows the corpus name, and the second column shows the number of repositories in each corpus. Columns 3-5 show the number of parsable files, the percentage of duplicate files, and the number of unique files. Columns 6-8 show the statistics of the unique files of each corpus, including lines of code (LOC), total number of tokens, and vocabulary size (i.e., number of unique tokens). We also use box plots to show the distribution of lines of code for the files in each corpus in Figure 2a, the distribution of number of tokens in Figure 2b, and the distribution of vocabulary size in Figure 2c.

From the table and figures, we can observe that HDs written in VHDL are more verbose than Verilog, as commonly accepted in the community, although the difference is not statistically significant[2]. Also, we can see that code (per file) written in SystemVerilog is somewhat shorter—though not significantly—than code written in VHDL and Verilog, because SystemVerilog is commonly used for writing test benches to verify HDs. Finally, we can observe that the number of total and unique tokens, taking into account the number of repositories, is substantially higher in Java repositories than in HDL repositories, and the number of tokens in SystemVerilog is significantly smaller than that of VHDL and Verilog.

[2]Under significance level $p < 0.05$ using the bootstrap method by Berg-Kirkpatrick et al. [6]. The same method applies to other significance tests in this paper.

Table 1: Statistics of Our HD Corpora and Java Corpora: Number of Repositories, Number of Parsable Files, Percentage of Duplicate Files, Number of Unique Files, Lines of Code (LOC), Number of Tokens, Vocabulary Size.

Corpus	#Repos	Files			LOC	#Tokens	Vocab. Size
		#Parsable	%Duplicate	#Unique			
VHDL	100	13,554	15.5%	11,459	4,759,308	14,572,639	227,117
Verilog	100	7,219	4.8%	6,869	3,433,764	8,238,560	273,893
SystemVerilog	100	2,021	6.5%	1,890	317,886	925,656	28,693
Java (Popular)	10	32,294	3.2%	31,264	6,672,160	23,502,694	387,812
Java (Naturalness)	10	9,886	2.6%	9,630	2,457,854	6,926,953	147,682



(a) LOC (b) Number of all tokens (c) Number of unique tokens

Figure 2: Lines of code (LOC) per file and number of tokens per file distributions in our HD corpora and Java corpora.

Table 2: Statistics of HDL Code Elements in Our Corpora: Number of Entities (Modules), Number of Functions, Number of Processes ("always" Blocks), Number of Input Ports, Number of Output Ports, and Number of Inout Ports. "Avg" Columns are the Numbers Per File and "Sum" Columns are the Total Number for All Files from All Repositories.

Corpus	#Entity/Module		#Function		#Process/Always		#Input Port		#Output Port		#Inout Port	
	Avg	Sum	Avg	Sum	Avg	Sum	Avg	Sum	Avg	Sum	Avg	Sum
VHDL	1.06	12,191	0.70	7,981	2.17	24,824	12.65	144,997	9.08	104,089	0.46	5,248
Verilog	1.24	8,511	0.20	1,399	3.22	22,123	21.66	148,806	27.43	188,395	1.13	7,739
SystemVerilog	0.97	1,837	0.28	531	1.26	2,381	4.46	8,425	2.54	4,792	0.06	116

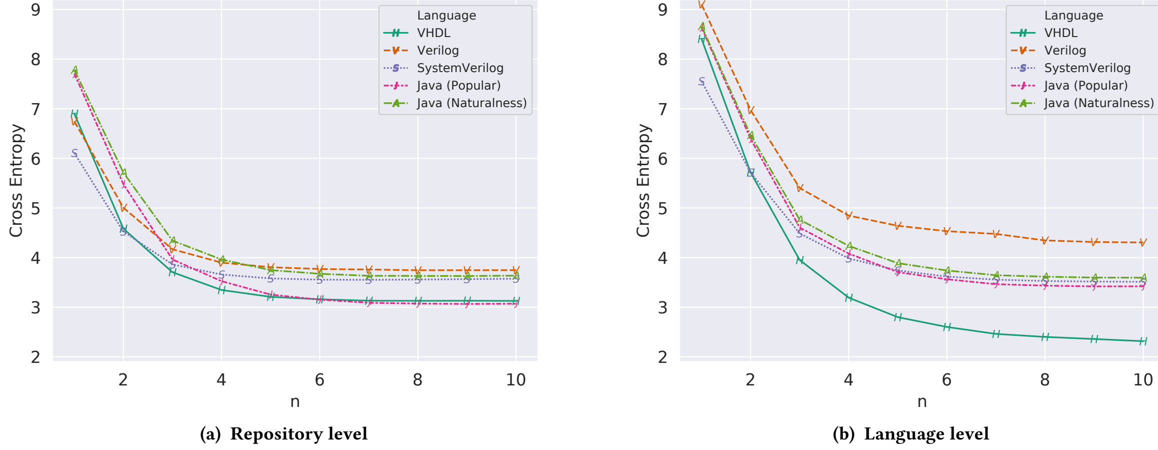
Table 2 shows the statistics of the numbers of several HDL code elements in our corpora. The first column shows the corpus name. Columns 2-3 show the number of entities (in VHDL) or modules (in Verilog and SystemVerilog) per file (Avg) and total in each corpus (Sum). Note that number of modules per file for SystemVerilog could be ≤ 1 because some files may consist of only program blocks but not modules. Columns 4-5 show the number of functions. Columns 6-7 show the number of processes (in VHDL) or "always" blocks (in Verilog and SystemVerilog). Finally, columns 8-13 show the number of input ports, output ports and inout ports. Although we do not report the details about each of these code elements, we find that a large number of them is available in popular open-source repositories, which can be exploited by various learning-based techniques to build better software tools for HDLs.

4 NATURALNESS OF HDS

This section presents an assessment of the naturalness of HDs, in comparison with software written in Java, using our corpora. To make sure the results are compatible with existing literature, we follow prior work on the naturalness of software [17, 44] and use n-gram language models for this analysis. We reveal that HDLs have comparable levels of naturalness to Java software.

4.1 Methodology

Language Modeling and Naturalness. At a high level, language models are generative models that capture statistical regularities in terms of style and vocabulary in a corpus. Their learned parameters in turn can be used to determine whether a document $\mathcal{D}$ fits in the corpus, by estimating the probability $P(\mathcal{D})$ of generating $\mathcal{D}$ under



(a) Repository level

(b) Language level

Figure 3: Line plots of the cross entropy measurements on our HD and Java corpora.

a trained model. Following [17, 44], we use the document's cross entropy $\mathtt{H}(\mathcal{D})$ as a fitness measure (also called "SelfCrossEntropy" in Musfiqur et al. [44]); cross entropy is the power term of perplexity, which is the standard measure for language model evaluation [22]:

$$\mathtt{H}(\mathcal{D}) = -\frac{1}{\mathtt{len}(\mathcal{D})} \log_2 P(\mathcal{D}) \qquad \mathtt{ppl}(\mathcal{D}) = 2^{\mathtt{H}(\mathcal{D})}$$

where $\mathtt{len}(\mathcal{D})$ is the number of tokens in $\mathcal{D}$. These are information-theoretic measures that capture the length-normalized inverse probability of $\mathcal{D}$ in the corpus, thus lower values entail more "fit"; note that $\mathcal{D}$ should be a test document that does not appear in the training set of the language model. As cross entropy and perplexity are monotonically correlated by definition, we report only cross entropy as in prior work [17, 44].

The concept of "naturalness" captures predictability and repetitiveness, and is an aggregated version of cross entropy over the whole corpus. Namely, we iteratively partition the corpus into training and testing sets; the lower the cross entropy over all partitions, the more "natural" the corpus is. To obtain the aggregated measures, we perform *10-fold validation* through the following steps, reimplementing Hindle et al. [17]:

(1) Randomly partition the corpus into 10 equally sized folds; this is done by first shuffling the files in the corpus, then splitting into 10 folds with equal number of lines of code.
(2) Train a language model on 9 folds and apply it on the remaining fold; repeat this step for each fold.
(3) Report the average cross entropy as a measurement of naturalness for the corpus.

N-Gram Language Model. The probability of document $\mathcal{D}$ is modeled by the joint probability of its token sequence:

$$P(\mathcal{D}) = P(w_1^m) = \prod_{i=1}^{m} P(w_i | w_1^{i-1})$$

In an n-gram language model, the Markov assumption is applied for the conditional probabilities for a given n:

$$P(w_i | w_1^{i-1}) \approx P(w_i | w_{i-n+1}^{i-1})$$

Therefore, the model directly counts the frequencies of token sequences of lengths n and $n-1$. Since our goal in this section is a *comparative* analysis of naturalness rather than building the best language model, using n-grams here makes sure that our results are comparable to prior work [14, 17]; in addition, compared to neural language models, n-gram models are transparent and straightforward in their probability estimates. This is particularly appealing given a large vocabulary, and it is also easy to vary the length of the sequence n, an analysis that led to important conclusions of source code naturalness which we also show below.

We used Software Language Processing library (SLP) [14, 49] to build the n-gram language model. The library had been used for building language models for imperative languages such as Java, and we modified it to accept HDLs. To handle zero probabilities of n-grams, we used Jelinek-Mercer smoothing as Hellendoorn and Devanbu [14] found it to be the most appropriate for source code.

Repository Level and Language Level Naturalness. Hindle et al. [17] measured *naturalness on the repository level*: they considered each Java repository as a (mini) corpus, computed naturalness measures on each repository, and reported the average among all repositories as the naturalness for Java software. In contrast, Musfiqur et al. [44] reported *language level naturalness*: they considered all repositories of one programming language as a single corpus. Our work reports naturalness on both levels, to gain insights into regularities in each language as a whole, while accounting for variabilities across different repositories written in the same HDL.

4.2 Analysis

Figure 3 shows the line plots of cross entropy for our HD corpora and Java corpora, with 3a and 3b showing the repository and language level measures respectively. In each line plot, the x-axis is the value of n for the n-gram language model, and the y-axis is the cross entropy, and each line corresponds to one corpus.

For all the corpora, the cross entropy monotonically drops as n increases, which indicates that a higher order n-gram language model (larger n) is better at capturing statistical regularity in both

HDLs and Java. The decline of cross entropy saturates around 4-grams for HDs, a similar finding as in Hindle et al. [17] for Java software. At the repository level, code written in all HDLs show lower cross entropy values than Java software at lower n. With larger n, VHDL code has the lowest cross entropy values among HDLs, similar to that of Java (Popular) corpus, while code written in Verilog and SystemVerilog have similar cross entropy values compared to that of Java (Naturalness) corpus. At the language level, VHDL code has the lowest cross entropy compared to other language corpora. SystemVerilog code shows lower cross entropy than Verilog code, with values similar to the two Java corpora. However, Verilog code has more regularities within repositories than at the language level. Our results indicate that Verilog code is more diverse across repositories compared to VHDL code.

We verify whether the differences between the cross entropy of the languages are statistically significant or not, by performing Wilcoxon rank-sum tests [59] under significance level $p < 0.05$ on each pair of the languages at both repository and language level when n = 10. At the repository level, the cross entropy differences between ⟨VHDL, Java (Popular)⟩ and ⟨SystemVerilog, Java (Naturalness)⟩ pairs—and only these pairs—are not statistical significant. At the language level, the cross entropy differences between ⟨Verilog, SystemVerilog⟩, ⟨SystemVerilog, Java (Popular)⟩, and ⟨SystemVerilog, Java (Naturalness)⟩ pairs—and only these pairs—are not statistical significant.

From these observations, we conclude that similar to software written in imperative languages, HDs also show clear properties of naturalness, i.e., predictability and repetitiveness, as captured by n-gram language models (with $n \geq 4$). VHDL code has the highest naturalness among the HDLs we study, and is higher than that of Java software at the repository level. This is due to VHDL being more verbose thus more repetitive. The findings for Verilog may be attributed to a larger vocabulary used in a smaller corpus compared to VHDL (Table 1).

The cross entropy of Java (Naturalness) at the repository level matches that reported in Hindle et al. [17], while at the language level this corpus has higher cross entropy values. Compared to that, the Java (Popular) corpus which represents recent and trending repositories has lower cross entropy values at both repository and language levels.

5 ASSIGNMENT COMPLETION FOR VHDL

The high level of naturalness of HDs that we revealed over our large corpora entails that many learning-based code automation techniques can be built to support hardware designers, including automated code completion, code search, etc. In this section, we start off the research in this direction by designing and implementing the first technique for an automated code completion task in VHDL. Our technique leverages unique semantics of VHDL.

5.1 Task

We tackle code completion in concurrent signal assignment statements (concurrent assignments for short). Namely, given a left hand side of a concurrent assignment, we predict the value on the right hand side to be assigned. **Example** (line 37 in Figure 1):

```
vicAddr(15)<= [ ? ] ;
```

Our task is inspired by prior work on code synthesis that looked at completing the right-hand side for imperative languages [13, 31]. In a personal communication, we also confirmed with a Sigasi IDE developer the relevance of the task. We focus on VHDL because of its high level of naturalness among HDLs we have studied, the abundance of data on GitHub, and our familiarity with the language.

A concurrent assignment can be divided into its left hand side (LHS) and right hand side (RHS). The LHS consists of a signal name or an array indexing expression which specifies the signal to be assigned to. The RHS consists of some operations over signals and literals, which is to be the new value of the LHS. Our task is to design an *assignment completion* technique that automatically completes RHS code fragments conditioned on the LHS and other context.

5.2 Neural Architecture

The underlying framework of our models is a sequence-to-sequence architecture that encodes a sequence into a deep representation, and predicts a target sequence. Originally used in natural language generation such as machine translation [52], this type of model has widely been used in language–code tasks such as code summarization [21, 26], comment generation [18, 37], and code generation [29, 57].

In contrast to traditional approaches which usually consider little context beyond the tokens that are to be encoded with a single encoder, the unique aspects (e.g., concurrent assignments) of HDLs motivate us to design a richer, context-driven model. Namely, our model utilizes the context in the source code before the concurrent assignment under consideration (assuming that the code is written in top-to-bottom order); this setting mimics how in practice developers would use a code completion tool. For example, when predicting the RHS of the concurrent assignment at line 37 in Figure 1, our model can utilize all the previous context in the file, including the *previous concurrent assignments* at lines 28–36, and the *signal type declarations* at lines 2–8 and lines 11–13.

Below, we first introduce our base sequence-to-sequence model, then describe our extensions that capture HDL-specific characteristics: (1) a multi-source architecture, i.e., having multiple encoders rather than one; (2) utilizing type embeddings; and (3) ensembling multiple sequence-to-sequence models. Figure 4 depicts the sequence-to-sequence model with the extensions (1) and (2).

Base Architecture. Sequence-to-sequence models are designed specifically for transduction tasks, i.e., given an input token sequence $\mathbf{x} = x_1, x_2, ..., x_m$, the model predicts a target token sequence $\mathbf{y} = y_1, y_2, ..., y_n$. This is achieved by an encoder—usually a recurrent neural network (RNN)—which encodes the input into a deep semantic vector representation $\mathbf{z} = \texttt{encoder}(\mathbf{x})$, and a decoder—another RNN—predicts the target $\mathbf{y}|\mathbf{x} \sim \texttt{decoder}(\mathbf{z})$. The entire architecture is trained end-to-end using back-propagation through time, maximizing the conditional log-likelihood

$$\log p(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^{n} \log p(y_i|\mathbf{y}_1^{i-1}, \mathbf{z})$$

over the training set. The output is predicted via a decoding algorithm, such as beam search, rather than predicting tokens one at a time as in language models. Our work uses bidirectional GRU [7] (an advanced RNN often used for sequence data [26, 62]) for both the encoder and the decoder.

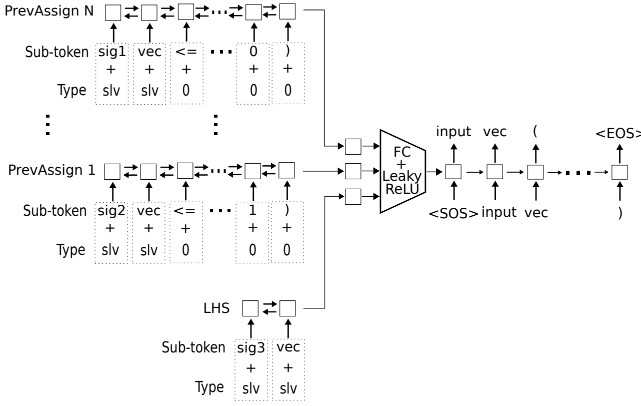


Figure 4: Neural architecture for our assignment RHS prediction model, which is a multi-source sequence-to-sequence model with type embedding concatenated with sub-token embedding. The left part shows several encoders for LHS or previous concurrent assignments; each encoder is a bidirectional GRU and the input at each time step is concatenation of the sub-token embedding and the type embedding (slv = ***std_logic_vector*****). The right part shows the decoder which predicts the sub-tokens in the output sequence (RHS). Attention mechanism is not shown in this figure to avoid unnecessary complexity but described in the main text.**

To improve the model's ability in capturing long-range dependencies among its tokens, we incorporate an attention mechanism using the global attention model [30] which is commonly used with sequence-to-sequence models. When predicting a target token y_i, the attention mechanism informs the model how much information should be pulled from each input token $x_j, j \in \{1, .., m\}$, which is then aggregated via a learned weighted sum over the encoder representation $\mathbf{z}$ constituting another layer over the decoder.

Sub-Tokenization. In VHDL, identifier names (e.g., signal names and function names) are usually composed of multiple sub-tokens separated by underscores (_). CamelCase is rarely used because VHDL is case-insensitive. We split the LHS and previous concurrent assignments into sub-tokens, normalize them to lower case, and use the obtained sequence of sub-tokens as the inputs to the models. Sub-tokenization helps the model to capture more semantics in the identifier names and to generalize across different repositories.

Multi-Source Sequence-to-Sequence Model. As described in Section 2, one of the characteristics of HDLs that is very different from imperative languages is that the processes—including concurrent assignments—are executed in parallel, despite the fact that they are still presented sequentially in a text editor. In addition, concurrent assignments that are similar to each other are often written close together. With this in mind, we design a multi-source architecture such that prior context—and the fact that concurrent assignments can be freely shuffled—is taken into account.

We denote each input sequence with a superscript, i.e., $\mathbf{x}^1, ..., \mathbf{x}^k$ for k sequences. The multi-source encoder is formulated as:

$$\mathbf{z}^j = \texttt{encoder}^j(\mathbf{x}^j), \forall j \in \{1, ..., k\}$$
$$\mathbf{z} = \texttt{merge}(\mathbf{z}^1, ..., \mathbf{z}^k)$$

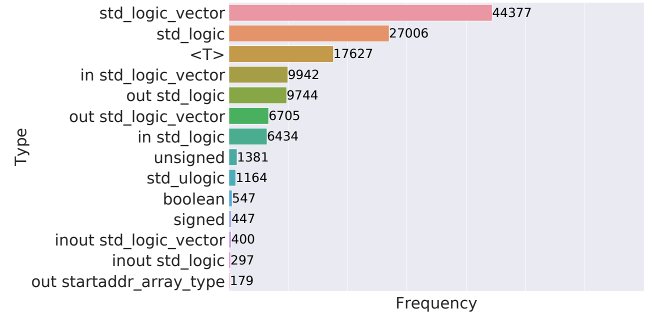


Figure 5: Histogram of VHDL signal types in our dataset. Each bar represents a type that we use for the type embedding and its value is the frequency of that type.

The `merge` function consists of a fully connected layer followed by a Leaky ReLU regularization. The input sequences are concatenated for the attention mechanism in the decoder.

In our preliminary study, the closest 5 previous concurrent assignments showed the most similar context 70% of the times (measured by Jaccard similarity among up-to 200 previous concurrent assignments). Thus we experiment with using 1–5 previous concurrent assignments. This leads to 2–6 encoders (one for LHS and others for previous concurrent assignments).

Type Embedding. Clearly, the expressions on the RHS have to match the type of the signals on the LHS. Thus, we condition on type information in the encoder(s) such that the model learns what expressions would be appropriate for the type of the LHS[3]. Therefore, in addition to the local context in LHS and previous concurrent assignments, we utilize the types of signals as *global context* as they are extracted from the port and local signal declarations at the beginning of entity/architecture definitions. This idea aligns with the recent work that uses types for code completion in imperative languages [27, 53].

There are many possible types in VHDL, but only few of them are frequently used. To reduce sparsity and improve the generalizability of the model, we focus on 13 types that are most frequently used in concurrent assignments. We replace all other types with a special token ⟨T⟩. Figure 5 shows the histogram of the signal types, where each bar represents one signal type or ⟨T⟩, and its value shows the frequency of the type in our dataset.

In our model, types are represented as a one-hot 14-bit vector, where each bit represents one of the 13 plus ⟨T⟩. We concatenate this type embedding to each sub-token of the LHS and previous concurrent assignments. If the sub-token is a part of a signal, the respective bit in the type embedding corresponding to the signal type is set to one; otherwise the type embedding is a zero vector.

Ensembling. Our preliminary experiments with the multi-source sequence-to-sequence models revealed that utilizing different previous statements, e.g., LHS + 1st previous concurrent assignment, LHS + 2nd previous concurrent assignment, etc., led to complementary model behavior. In particular, they make better/worse predictions for different portions of the development set of the dataset, and comparable performance on the entire development

[3] An alternative approach would be to perform an extensive static analysis and ensure that our model never suggests RHS with an incorrect type, but we wanted to focus on an end-to-end neural architecture in this work.

Table 3: Statistics of Our Collected Assignments Dataset.

Statistic	All	Train	Dev	Test
#Assignments	49,982	39,986	4,998	4,998
Avg. LHS length	4.10	4.11	4.06	4.10
Avg. RHS length	8.55	8.56	8.51	8.51

set. Furthermore, we also observed the RHS of the current concurrent assignment can be frequently the same as the RHS of a previous concurrent assignment, especially when the LHS of the two concurrent assignments are similar. Thus the rule-based models that copy RHS from 1st-5th concurrent assignments could complement the aforementioned sequence-to-sequence models.

Because of the parallelism of HDLs, the order in which previous concurrent assignments are encoded does not matter. Thus, we ensemble these sequence-to-sequence and rule-based models to leverage their complementary prediction powers: given a set of inputs, we use all the models to predict the output sequence individually, which gives us a confidence score from each sequence-to-sequence model. We train a regression model, using the data from development set, that assigns a score on each prediction. The input features include the similarity of each prediction to the RHS of each concurrent assignment from 1st-5th concurrent assignments, the similarity of the LHS of current concurrent assignment with the LHS of each concurrent assignment from 1st-5th concurrent assignments, and the confidence scores of each sequence-to-sequence model. The similarity of two sequences of sub-tokens is measured as one minus the Jaccard distance between the sets of their bag of sub-tokens. We then rerank the predictions and select the one with the highest score as predicted by the regression model.

5.3 Data

We extracted all concurrent assignments from our VHDL corpus. We extracted each assignment's LHS and RHS and the type declarations in each entity/architecture, and performed post-processing to obtain the type for each signal in the concurrent assignments. Table 3 shows the statistics of our collected data. We collected 49,982 concurrent assignments (across files) in total, where LHS has 4.10 sub-tokens on average and RHS has 8.55 sub-tokens on average. Note that 92.5% and 94.9% of LHS and RHS sequences in our dataset have lengths within the range of one standard deviation. This means that long LHS and RHS sequences are rare. We observed that all long RHS sequences contain a number of logic operators.

To obtain the training, development, and testing sets, we randomly shuffle the list of all files and then take enough files to obtain ~10% of assignments for the testing set and ~10% of assignments for the development set; assignments from other files (~80%) go into the training set. Duplicates that contain same assignment and same context are removed.

5.4 Baselines, Models, and Training Details

Rule-Based Baseline. We compare to a baseline model which outputs the RHS of the 1st previous concurrent assignment, or an empty sequence if there is no previous concurrent assignment.

Language Model Baseline. This model utilizes the inputs as existing tokens $\{t_1, ..., t_m\}$, and repeatedly predicts the next token using the language model LM as $t_{i+1} = LM(t_1, ..., t_i)$, until the predicted token is the end-of-statement (i.e., a semicolon ";"). The model outputs all the predicted tokens (except end-of-statement) as the prediction for RHS.

We use RNN language model because prior work [45, 58] showed its power on code completion tasks. We use three variants: *RNNLM* is the RNN language model trained with the concurrent assignment (LHS + RHS) sentences; in *RNNLM+PA(1)*, each sentence is a concurrent assignment with its 1st previous concurrent assignment; in *RNNLM+PA(1-5)*, each sentence is a concurrent assignment with its 1st-5th previous concurrent assignments. For example, the sentence extracted from the concurrent assignment on line 37 in Figure 1 for *RNNLM* is

```
vicAddr(15)<= (not cia2_pao(1));
```

the sentence for the same assignment for *RNNLM+PA(1)* is

```
vicAddr(14)<= (not cia2_pao(0));
vicAddr(15)<= (not cia2_pao(1));
```

and for *RNNLM+PA(1-5)* is

```
uart_rts <= cia2_pbo(1); uart_dtr <= cia2_pbo(2);
uart_ri_out <= cia2_pbo(3); uart_dcd_out <= cia2_pbo(4);
vicAddr(14)<= (not cia2_pao(0));
vicAddr(15)<= (not cia2_pao(1));
```

We also experimented with n-gram language model for completeness. Similar to RNN language model, we use three variants for n-gram language model: *10gramLM* is trained with the concurrent assignment (LHS + RHS) sentences; in *10gramLM+PA(1)*, each sentence is a concurrent assignment with its 1st previous concurrent assignment; in *10gramLM+PA(1-5)*, each sentence is a concurrent assignment with its 1st-5th previous concurrent assignments.

Sequence-to-Sequence Models. The *S2S* model is the sequence-to-sequence model with only LHS as input. *S2S+PA(1)* utilizes the 1st previous concurrent assignment as additional input, using multi-source architecture. *S2S+PA(1)+Type* adds type embedding based on *S2S+PA(1)*. *S2S+PA(1-k)+Type* utilizes the 1st-*k*th previous concurrent assignments (e.g., *S2S+PA(1-4)+Type* utilizes 1st-4th previous concurrent assignments), using the multi-source architecture with type embedding. *S2S+PA(Ensemb-1-5)+Type* utilizes 1st-5th previous concurrent assignments by ensembling models and also uses type embedding. *S2S+PA(Concat-1-5)+Type* uses only the basic sequence-to-sequence architecture and utilizes the 1st-5th previous assignments by concatenating them with the LHS as one input sequence, and it also uses type embedding. All the sequence-to-sequence models have attention mechanism enabled.

Hyper-Parameters. We set hyper-parameters of all neural models by tuning on the development set. For sequence-to-sequence models, both encoder and decoder bidirectional GRU have 2 layers and hidden state dimensions of 512; the sub-token embedding dimension is 512, randomly initialized. We limit the lengths of input sub-tokens sequences to be at most 200. We train each model with batch size 32 for at most 60,000 steps (one batch per step), and use an early stop mechanism which stops the training if the validation loss does not improve for the subsequent 10 checkpoints (one checkpoint every 75 steps). During training time, we train the sequence-to-sequence model with teacher forcing [60]; during inference time, the model performs beam search with beam size 5.

Table 4: BLEU, Accuracy (Acc), and Exact-Match Accuracy (xMatch) Scores of Assignment Completion Models. All Improvements are Statistically Significant.

Model	BLEU	Acc [%]	xMatch [%]
Rule-based Baseline	29.4	38.1	8.8
RNNLM+PA(1)	18.0	22.0	8.2
S2S+PA(Ensemb-1-5)+Type	**37.3**	**48.0**	**19.1**

Table 5: BLEU, Accuracy (Acc), and Exact-Match Accuracy (xMatch) Scores of the RNN Language Model Variants.

Model	BLEU	Acc [%]	xMatch [%]
RNNLM	5.9	7.3	3.5
RNNLM+PA(1)	18.0	22.0	8.2
RNNLM+PA(1-5)	12.3	14.9	6.2

The RNN language models have the same hyper-parameters except that they use single-directional GRU. Our models are implemented using Pytorch [41] and OpenNMT [23].

5.5 Results

Evaluation Metrics. We use three automatic metrics to evaluate our models:

- BLEU: originally proposed for Machine Translation [38], BLEU is now widely used in language–code tasks [18]. This metric calculates the percentage of n-grams in the predicted output that also appear in human-written RHS, averaging across $n \in \{1, 2, 3, 4\}$ and using a brevity penalty to eliminate the impact of the number of tokens predicted. The range of values is 0–100. Following settings from code-related tasks, e.g., [18], we use sentence-level BLEU implementation in NLTK library [55] with the smoothing method proposed by Lin and Och [28].
- Accuracy (Acc): we report the accuracy of sub-tokens, averaged across the testing set. The accuracy for each concurrent assignment is calculated using the following formula:

$$accuracy = \frac{\texttt{len}(\{i \mid pred[i] = tgt[i]\})}{\texttt{max}(\texttt{len}(pred), \texttt{len}(tgt))}$$

  where $pred$ is the sequence of sub-tokens in the predicted RHS, tgt is the sequence of sub-tokens in the human-written RHS, and `len` function calculates the length of a sequence or size of a set.
- Exact-match accuracy (xMatch): we also report the accuracy in statement-level, i.e., the number of predicted RHS that exactly match the human-written RHS divided by total number of concurrent assignment across the testing set.

We perform statistical significance testing to compare the metrics between models under significance level $p < 0.05$ using the bootstrapping method [6].

Quantitative Results. Table 4 compares the performance of the baselines and our best model evaluated on the testing set. We ran each model 3 times and show the averaged metrics; each time we use a different random seed for initializing the model as well as for splitting the training/development/testing sets [12]. The best model is *S2S+PA(Ensemb-1-5)+Type* with 37.3 BLEU, 48.0% accuracy, and 19.1% exact-match accuracy. This is the model that ensembles multi-source sequence-to-sequence models for 5 previous assignments. It significantly outperforms the rule-based and language model baselines.

Table 6: BLEU, Accuracy (Acc), and Exact-Match Accuracy (xMatch) Scores of N-gram Language Modeling Based Assignment Completion Models.

Model	BLEU	Acc [%]	xMatch [%]
10gramLM	14.5	16.4	10.4
10gramLM+PA(1)	17.8	20.0	13.5
10gramLM+PA(1-5)	10.6	10.6	5.6

Table 7: Ablation Study. The Best Scores are in Bold Text and Statistically Significantly Outperform All Others.

Model	BLEU	Acc [%]	xMatch [%]
S2S+PA(Ensemb-1-5)+Type	**37.3**	**48.0**	**19.1**
S2S+PA(1-5)+Type	24.4	28.2	11.4
S2S+PA(1-4)+Type	24.7	29.0	11.6
S2S+PA(1-3)+Type	26.1	30.9	13.4
S2S+PA(1-2)+Type	25.0	29.6	13.1
S2S+PA(1)+Type	25.8	30.4	14.1
S2S+PA(1)	25.4	30.0	14.4
S2S	19.6	21.9	12.3
S2S+PA(Concat-1-5)+Type	23.2	26.9	12.2

Table 5 compares the three variants of the RNN language model. None of the language models outperform the rule-based baseline. *RNNLM+PA(1)* is better than *RNNLM+PA(1-5)*, despite the latter one having more context as inputs. This is likely because the language model was trained with too much context that is less relevant to the current assignment thus distracts the model from the task of assignment completion. We also observed that the language models usually cannot predict the end-of-statement token correctly and end up predicting very long sequence.

Table 6 compares the three variants of the n-gram language model, where $n = 10$ because it gives the best results among $n \in \{1, ..., 10\}$ on the development set. Both models *10gramLM* and *10gramLM+PA(1)* are worse than the rule-based baseline, although *10gramLM+PA(1)* achieves a reasonable performance (20.0% accuracy and 17.8 BLEU). Moreover, *10gramLM+PA(1-5)* has much worse performance, despite the fact that higher order n-grams consistently shows lower cross entropy values (i.e., higher naturalness). This again shows that using too much context to train language model is not helpful and may distract the model from the task, and compared to RNN language model, n-gram model is more likely to be distracted.

Ablation Study. Table 7 compares the variants of sequence-to-sequence models, all of which (except for the basic *S2S* model) are better than the RNN language model baseline. The best model, *S2S+PA(Ensemb-1-5)+Type*, significantly outperforms all sequence-to-sequence models on all computed metrics. In preliminary experiments, we tried using 6+ previous concurrent assignments, but

found them significantly decreasing the performance, likely due to the low similarity of those assignments to the current assignment. Notably, all the multi-source models significantly outperform the models without multi-source architecture (*S2S* and *S2S+PA(Concat-1-5)+Type*), which shows that multi-source architecture is effective.

For type embedding, comparing *S2S+PA(1)* and *S2S+PA(1)+Type*, there is an improvement, although not statistically significant. The ensembling model *S2S+PA(Ensemb-1-5)+Type* has 37.3 BLEU, 48.0% accuracy, and 19.1% exact-match accuracy, which is significantly better than *S2S* (not including any context) and significantly better than *S2S+PA(1-5)+Type* which uses one multi-source architecture to include all previous concurrent assignments.

From these observations, we conclude that the using of ensembling model to include multiple previous concurrent assignments (local context) and the using of type embedding (global context) have positive effect on the performance of the models. Language model is not a good option for this task because it cannot effectively utilize the context information, such as those available in previous concurrent assignments.

Qualitative Analysis. We showcase two examples in Table 8 from our corpus to illustrate our models. For each example, we show its inputs (LHS, previous concurrent assignments, and relevant signal types), its expected output RHS, and the predictions of *RNNLM+PA(1)*, *S2S*, *S2S+PA(1)+Type*, and *S2S+PA(Ensemb-1-5)+Type* models.

In the first example, the five previous concurrent assignments have the same pattern `m_7seg_? <= disp_7seg_segment(?)`, where `?` represents a missing character or number. The input LHS is similar to the LHS of previous concurrent assignments, thus the output RHS should also be similar to the RHS of previous concurrent assignments and follow the pattern. *RNNLM+PA(1)* and *S2S+PA(1)+Type* models learn the `disp_7seg_segment(?)` pattern, but cannot predict the correct number in the parenthesis. The *S2S+PA(Ensemb-1-5)+Type* detects the pattern where the number in the parenthesis is ascending and predicts the correct number "5" following the pattern. Due to the lack of longer local context, *RNNLM+PA(1)* guesses a wrong pattern to repeat the numbers, and *S2S+PA(1)+Type* guesses a wrong pattern to decrease the number. Due to lack of context (and especially global context), *S2S* predicts an irrelevant variable.

In the second example, there are several patterns in the RHS of previous assignments but no clear pattern in the LHS. *RNNLM+PA(1)*, *S2S+PA(1)+Type*, and *S2S+PA(Ensemb-1-5)+Type* are still able to detect the right pattern `sseg_edu_cathode_out(?)` for RHS, and the latter two models predicts the right number in the parenthesis. Without any context, the *S2S* model could not learn this pattern and predicts both a wrong variable and value.

Table 8: Example Predictions of Our Models.

Repository: f32c_f32c	
LHS: `m_7seg_f`	
Previous Concurrent Assignments:	
1st	`m_7seg_e <= disp_7seg_segment(4);`
2nd	`m_7seg_d <= disp_7seg_segment(3);`
3rd	`m_7seg_c <= disp_7seg_segment(2);`
4th	`m_7seg_b <= disp_7seg_segment(1);`
5th	`m_7seg_a <= disp_7seg_segment(0);`
Types:	
`m_7seg_f`	*out std_logic*
`m_7seg_2`	*out std_logic*
`m_7seg_d`	*out std_logic*
`m_7seg_c`	*out std_logic*
`m_7seg_b`	*out std_logic*
`m_7seg_a`	*out std_logic*
`disp_7seg_segment`	*std_logic_vector*
Expected Output RHS: `disp_7seg_segment(5)`	
Predictions:	
RNNLM+PA(1)	`disp_7seg_segment(4)`
S2S	`state_cur(234)`
S2S+PA(1)+Type	`disp_7seg_segment(3)`
S2S+PA(Ensemb-1-5)+Type	`disp_7seg_segment(5)`
Repository: fpga-logi_logi-projects	
LHS: `pmod3(0)`	
Previous Concurrent Assignments:	
1st	`pmod2(1)<= sseg_edu_cathode_out(1);`
2nd	`pmod2(5)<= sseg_edu_cathode_out(0);`
3rd	`sys_sda <= 'z';`
4th	`sys_scl <= 'z';`
5th	`vga_clk <= clk_50mhz;`
Types:	
`pmod3`	*inout std_logic_vector*
`pmod2`	*inout std_logic_vector*
`sseg_edu_cathode_out`	*std_logic_vector*
`sys_sda`	*inout std_logic*
`sys_scl`	*inout std_logic*
`vga_clk`	*std_logic*
`clk_50mhz`	*std_logic*
Expected Output RHS: `sseg_edu_cathode_out(2)`	
Predictions:	
RNNLM+PA(1)	`sseg_edu_cathode_out(0)`
S2S	`sseg_edu_anode_out(6)`
S2S+PA(1)+Type	`sseg_edu_cathode_out(2)`
S2S+PA(Ensemb-1-5)+Type	`sseg_edu_cathode_out(2)`

6 THREATS TO VALIDITY

Internal. As our goal was to compare different characteristics of the models under the same environment, we did not fine-tune the hyper-parameters for each assignment completion model individually. The models might achieve better performance if their hyper-parameters are fine-tuned, but we do not expect changes to our reported observations. We plan to evaluate the benefit of such fine-tuning in future work.

Our scripts and implementation may contain bugs. To mitigate this threat, at least two authors reviewed each script and the output logs. We also automated each step of the process to remove a chance for human errors while executing the experiments.

We used only parsable files in our HD corpora and experiments. To parse the files, we initially generated parsers using the ANTLR parser generator framework [39] and publicly available grammars that appeared to be the most complete [40, 42]. However, those grammars are somewhat obsolete and do not support the latest versions of HDLs. To mitigate this threat, we extended these grammars in several ways to increase the number of files that can be parsed by supporting more recent language standards. Of all files in VHDL

Table 9: VHDL Parsing Errors Categories.

Error	Count
Mismatched input keyword	27,529
Extraneous input keyword	10,979
Unrecognized token	2,628
Not interpretable syntax	609
Missing input keyword	321

corpus, 20% still cannot be parsed. Table 9 shows the error categories and the count for each of them; note that there are multiple errors per file. Note that parsing one file could generate multiple parsing errors. We observed five types of errors: mismatched input keyword (27,529), when the parser is expecting a keyword but got a different one; extraneous input keyword (10,979), when the parser is not expecting a keyword but got one; unrecognized token (2,628), when the file contains binary data that the parser can not handle; not interpretable syntax (609), when the parser could not find a valid parsing rule for a code snippet; and missing input keyword (321), when the parser is expecting a keyword but got none (due to end of file).

External. We constructed our dataset from HDL repositories available on GitHub. We chose GitHub because of its popularity and our prior experience with mining the repositories available on this repository hosting service. As many HDs are not publicly available, the HDs used in our study might not be representative of proprietary HDs. We mitigate this threat by ranking the repositories on GitHub by the number of stars, which might indicate that these repositories are also widely used by various companies. Additionally, nowadays many HDs are made publicly available on GitHub by hardware companies.

7 RELATED WORK

We briefly discuss related work on (1) the naturalness of software, (2) software engineering for HDs, and (3) code completion.

The Naturalness of Software. Hindle et al. [17] were the first to study the naturalness of software written in Java and C. Recently, Musfiqur et al. [44] replicated the study of naturalness for several other imperative languages and on much larger corpora. Our Section 4 replicates the study of naturalness for HDs, and we have discussed our similarities and differences to these two works throughout the text. Hellendoorn et al. [15] studied the naturalness of proofs written in Coq and HOL and found that proofs are also repetitive and predictable. Unlike prior work, we studied the naturalness of HDs.

Software Engineering for HDs. Clarke et al. [8] developed a program slicing technique for VHDL based on mapping its operational semantics to imperative languages. Sudakrishnan et al. [50] analyzed the bug fix history of four HD repositories written in Verilog and grouped them into 25 bug fix patterns. They found that 29-55% of the bug fix pattern instances involve assignment statements, which reflects the need for automating completion for assignment statements to reduce a chance for manual errors. Duley et al. [10] developed Vdiff, a program differencing tool for Verilog that finds syntactic differences of two versions of code. Notably, their differencing algorithm is position-independent to robustly handle language constructs whose relative orderings do not matter, e.g., concurrent processes. Uemura et al. [54] developed a clone detection tool for Verilog by first converting Verilog modules into pseudo C++ code, then applying code clone detectors for C++. Schkufza et al. [46] developed the first just-in-time compiler for Verilog to speed up program execution on FPGAs. These software engineering techniques for HDs have not considered using learning-based approaches. Our study reveals the naturalness of HDs which stands to motivate other researchers and practitioners to improve previously studied techniques based on deep learning for HDLs. We presented the first work in this direction by designing and implementing an assignment right hand side completion technique for VHDL.

Code Completion. Code completion is a task that recommends upcoming code elements given the code context. Hindle et al. [17] developed a code completion tool using n-gram language models. Proksch et al. [43] used Bayesian Networks for code completion which can utilize "global context" from methods, class, etc. Li et al. [27] developed a neural code completion technique with attention mechanism and pointer networks to better handle out-of-vocabulary words. Raychev et al. [45] proposed an RNN language model for completing holes in partial programs with the most likely sequences of API method calls. Svyatkovskiy et al. [53] proposed a neural code completion model that incorporates type information, instantiated as a tool called Pythia for Python. Hellendoorn et al. [16] applied several code completion tools on real-world data and analyzed the real-world efficacy of those tools. Hu et al. [19] proposed LSTM based code completion tool by inducing tokens at character and token levels, thereby reducing vocabulary size. Sun et al. [51] developed neural code generation model that implements attention mechanism and tree-based AST reader. These prior efforts targeted imperative languages. We introduced the first code completion technique for concurrent assignments in VHDL utilizing its unique semantics. Furthermore, unlike prior work on code completion, we used a multi-source sequence-to-sequence model with type embedding which we found suitable for our task.

8 CONCLUSION

We assembled large HD corpora consisting of source code written in VHDL, Verilog, and SystemVerilog and reported on their characteristics. We studied the naturalness of HDs, and our language model evaluation reveals that HDs possess a high level of naturalness similar to software written in imperative languages. Further, we built several deep learning models for automated code completion in VHDL utilizing unique characteristics of HDLs (e.g., semantics of concurrent signal assignment statements). These characteristics led to effective neural models, achieving a BLEU score of 37.3. Our study stands to motivate other researchers and practitioners to develop code automation techniques based on deep learning for HDLs to provide powerful tools to hardware designers.

ACKNOWLEDGMENTS

We thank Nader Al Awar, Hendrik Eeckhaut, Kush Jain, Owolabi Legunsen, Kayvan Mansoorshahi, Dusan Matic, Sasa Misailovic, Raymond Mooney, Jiyang Zhang, and the anonymous reviewers for their feedback on this work. This research was partially supported by a Google Faculty Research Award.

REFERENCES

[1] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!* 143–153.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *CoRR* (2017). arXiv:1709.06182

[3] Peter J. Ashenden. 2001. *The Designer's Guide to VHDL* (2nd ed.). Morgan Kaufmann Publishers Inc.

[4] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 159.

[5] Stephany Bellomo, Robert L. Nord, Ipek Ozkaya, and Mary Popeck. 2016. Got Technical Debt? Surfacing Elusive Technical Debt in Issue Trackers. In *International Conference on Mining Software Repositories*. 327–338.

[6] Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An Empirical Investigation of Statistical Significance in NLP. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. 995–1005.

[7] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Empirical Methods in Natural Language Processing*. 1724–1734.

[8] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. 1999. Program Slicing of Hardware Description Languages. In *Correct Hardware Design and Verification Methods*. 298–313.

[9] Massimiliano Di Penta, Daniel M German, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2010. An Exploratory Study of the Evolution of Software Licensing. In *International Conference on Software Engineering*. 145–154.

[10] Adam Duley, Chris Spandikow, and Miryung Kim. 2010. A Program Differencing Algorithm for Verilog HDL. In *Automated Software Engineering*. 477–486.

[11] Inc. Eclipse Foundation. 2020. *JDT Core Component | The Eclipse Foundation.*

[12] Kyle Gorman and Steven Bedrick. 2019. We Need to Talk about Standard Splits. In *Annual Meeting of the Association for Computational Linguistics*. 2786–2791.

[13] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *Conference on Programming Language Design and Implementation*. 27–38.

[14] Vincent J. Hellendoorn and Premkumar T. Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 763–773.

[15] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Mohammad Amin Alipour. 2018. On the Naturalness of Proofs. In *International Symposium on the Foundations of Software Engineering, NIER*. 724–728.

[16] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When Code Completion Fails: A Case Study on Real-World Completions. In *International Conference on Software Engineering*. 960–970.

[17] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *International Conference on Software Engineering*. 837–847.

[18] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *International Conference on Program Comprehension*. 200–210.

[19] Xing Hu, Rui Men, Ge Li, and Zhi Jin. 2019. Deep-AutoCoder: Learning to Complete Code Precisely with Induced Code Tokens. *Computer Software and Applications Conference* 1 (2019), 159–168.

[20] Oskar Jarczyk, Blazej Gruszka, Szymon Jaroszewicz, Leszek Bukowski, and Adam Wierzbicki. 2014. GitHub Projects. Quality Analysis of Open-Source Software. In *International Conference of Social Informatics*. 80–94.

[21] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Automated Software Engineering*. 135–146.

[22] Dan Jurafsky and James H. Martin. 2019. *Speech and Language Processing* (3rd draft ed.).

[23] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M. Rush. 2017. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In *Annual Meeting of the Association for Computational Linguistics-System Demonstrations*.

[24] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2019. PathMiner: A Library for Mining of Path-based Representations of Code. In *International Conference on Mining Software Repositories*. 13–17.

[25] Jeremiah C Leary. 2020. *vhdl-style-guide Documentation.*

[26] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *International Conference on Software Engineering*. 795–806.

[27] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *International Joint Conference on Artificial Intelligence*. 4159–25.

[28] Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: A Method for Evaluating Automatic Evaluation Metrics for Machine Translation. In *International Conference on Computational Linguistics*. 501–507.

[29] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *International Conference on Language Resources and Evaluation*.

[30] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Empirical Methods in Natural Language Processing*. 1412–1421.

[31] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *Conference on Programming Language Design and Implementation*. 48–61.

[32] Microsemi. 2020. *Microsemi HDL Coding Style Guidelines*. https://www.microsemi.com/document-portal/doc_download/130823-hdl-coding-style-guide.

[33] MiSTer-devel. 2020. *MiSTer - An Open Project That Aims to Recreate Various Classic Computers, Game Consoles and Arcade Machines, Using Modern Ardware*. https://github.com/MiSTer-devel/Main_MiSTer/wiki.

[34] MiSTer-devel. 2020. *MiSTer-devel/C64_MiSTer*. https://github.com/MiSTer-devel/C64_MiSTer/tree/0efb9e1b7eb380e3cbd87bce770901802b8d4615.

[35] Parisa Moslehi, Bram Adams, and Juergen Rilling. 2018. Feature Location Using Crowd-based Screencasts. In *International Conference on Mining Software Repositories*. 192–202.

[36] Sheena Panthaplackel, Milos Gligoric, Raymond J. Mooney, and Junyi Jessy Li. 2020. Associating Natural Language Comment and Source Code Entities. In *AAAI Conference on Artificial Intelligence*. 8592–8599.

[37] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. 2020. Learning to Update Natural Language Comments Based on Code Changes. In *Annual Meeting of the Association for Computational Linguistics*. 1853–1868.

[38] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Annual Meeting of the Association for Computational Linguistics*. 311–318.

[39] ANTLR/Terence Parr. 2020. *The Antlr Programming Language Modelling Tool*. https://www.antlr.org/.

[40] ANTLR/Terence Parr. 2020. *antlr/grammars-v4: Grammars written for ANTLR v4; expectation that the grammars are free of actions.* https://github.com/antlr/grammars-v4.

[41] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *NIPS Autodiff Workshop*.

[42] Eirik Prestegårdshus. 2020. *eirikpre/VSCode-SystemVerilog: SystemVerilog support in VS Code.*

[43] Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent Code Completion with Bayesian Networks. *ACM Trans. Softw. Eng. Methodol.* 25, 1 (2015), 3:1–3:31.

[44] Musfiqur Rahman, Dharani Palani, and Peter Rigby. 2019. Natural Software Revisited. In *International Conference on Software Engineering*. 37–48.

[45] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. *SIGPLAN Not.* 49, 6 (2014), 419–428.

[46] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. 271–286.

[47] Lattice Semiconductor. 2020. *Lattice HDL Coding Style Guidelines*. http://www.latticesemi.com/~/media/LatticeSemi/Documents/UserManuals/EI/HDLcodingguidelines.PDF?document_id=48203.

[48] Sigasi. 2020. *Home - Sigasi*. https://www.sigasi.com/.

[49] SLP-team. 2020. *SLP-team/SLP-Core: Your Library for Dynamic Language Modeling*. https://github.com/SLP-team/SLP-Core.

[50] Sangeetha Sudakrishnan, Janaki Madhavan, E. James Whitehead, Jr., and Jose Renau. 2008. Understanding Bug Fix Patterns in Verilog. In *International Conference on Mining Software Repositories*. 39–42.

[51] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A Tree-Based Transformer Architecture for Code Generation. In *AAAI Conference on Artificial Intelligence*. 8984–8991.

[52] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Annual Conference on Neural Information Processing Systems*. 3104–3112.

[53] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted Code Completion System. In *International Conference on Knowledge Discovery & Data Mining*. 2727–2735.

[54] K. Uemura, A. Mori, K. Fujiwara, E. Choi, and H. Iida. 2017. Detecting and Analyzing Code Clones in HDL. In *International Workshop on Software Clones*. 1–7.

[55] Wiebke Wagner, Steven Bird, Ewan Klein, and Edward Loper. 2010. Natural Language Processing with Python, Analyzing Text with the Natural Language Toolkit. *Language Resources and Evaluation* 44, 4 (2010), 421–424.

[56] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S Yu. 2019. Multi-modal Attention Network Learning for Semantic Source Code Retrieval. In *Automated Software Engineering*. 13–25.

[57] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *Annual Conference on Neural Information Processing Systems*. 6559–6569.

[58] Martin White, Christopher Vendome, Mario Linares Vásquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *International Conference on Mining Software Repositories*. 334–345.

[59] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[60] Ronald J. Williams and David Zipser. 1989. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation* 1, 2 (1989), 270–280.

[61] Xilinx. 2020. *Xilinx HDL Coding Style Guidelines*. https://wiki.electroniciens.cnrs.fr/images/Xilinx_HDL_Coding_style.pdf.

[62] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander J. Smola, and Eduard H. Hovy. 2016. Hierarchical Attention Networks for Document Classification. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 1480–1489.