A web based tool for teaching hardware design based on the plain simple hardware description language

Karsten Becker
Technical University of Hamburg
Email: k.becker@tuhh.de

***Abstract*—In this paper, a new online learning tool is presented. The goal is to improve the learning experience for designing hardware on register transfer level (RTL)-level by facilitating collaboration and allowing a more agile development cycle. It also enables the inspection of the learning process of the students.**

When students have to learn RTL hardware-design, usually either Verilog or VHDL are used as hardware description language. Those languages do not just have an unfamiliar syntax and programming paradigm for the students (they are not sequential, but rather describe the wiring of components), they also come with complex tools. During the hands-on part of the lecture, a lot of time is spent in learning the functions (and bugs) of each tool involved. It also happens that VHDL was originally designed for a different purpose and thus accepts input that is not necessarily fit for synthesis for FPGAs, which are primarily used as target for those hardware designs. Much to the frustration of the students, it is also rather easy to create a mismatch between what the simulation does and how the design behaves on the actual hardware.

This does not only result in a highly unsatisfying learning experience, as "learning" is mostly about understanding the tools and their output, but it also distracts from the real goal of learning how to properly design hardware.

A novel programming language called PSHDL, the "plain simple hardware description language", keeps the parallelism paradigm of VHDL and Verilog, but provides a significantly improved workflow. This workflow and its features are outlined in this paper. The improvements are not just for the student side, but also allow the teacher to take a direct look at the students' progress and learning path. This information could then be used to improve the whole learning experience.

I. Related work

An idea similar to this paper is proposed in "Web-scale data gathering with BlueJ" [1]. The proposal contains ideas to capture similar data: file modifications, compiler invocations and IDE usage. Unlike with PSHDL, the collected data is aimed at getting a better understanding of the usage patterns for that IDE, rather than tracking students and their progress. It is also aimed at Java, a much better researched language than the area of hardware description languages.

In "Analyses of Student Programming Errors In Java Programming Courses" [3], students were given a certain task to complete and the occurring errors were logged. The course was aimed at students without prior knowledge in any programming language. The setup used in this paper tracks the progress of students. Also, the students investigated in this paper do have prior programming experience, although not necessarily with the parallel paradigm that is used in hardware description languages.

While there are quite a few papers [4] [5] [6] [2] that suggest the usage of remote laboratory setups for testing, this paper suggests to even run the simulation in the browser and rather create custom visualization. As PSHDL does not allow a mismatch between simulation and synthesis (unlike other hardware description languages), the simulation results can become very close to the real hardware.

In order to improve the learning experience, other languages such as hdl [7] or LOLA [8] have been proposed. Those languages are not well suited for large-scale projects. Either because they lack the ability to be implemented in hardware, or they are rather too low or high level. PSHDL can be compiled to synthesizable VHDL and even allows the instantiation of other components not written in PSHDL.

II. Features of PSHDL

PSHDL is a hardware description language that is designed to be easy to learn. This is accomplished by featuring a small language kernel with just 33 keywords and a C-like syntax. In contrast, Verilog has 106 and VHDL 95 keywords. The C-like syntax increases readability and writability for people with prior knowledge in another C-like language which, according to the TIOBE programming language [9], account for more than 50% of the most popular programming languages.

One feature to lower the barrier of entrance is an online editor. In order to use the online editor, a workspace has to be created. This workspace contains all files that are uploaded or created by the user.

The web interface [10] consists of a tabbed interface as can be seen in figure 1. The first tab, labeled 'learn', contains a short introduction into PSHDL. The introduction shows how to create a blinking LED, how to describe state machines and contains a few short examples to illustrate the parallel nature of PSHDL.

The second tab, labeled 'code', provides a listing of files within the workspace and the online editor. This editor provides features like syntax highlighting, code folding and other things that make editing online enjoyable. Additionally, there are a few common examples like a ripply-carry-adder, a test bench and more. The user can upload files by dragging them into the browser, perform edits and save the modified files. It is also possible to compile the workspace or to simulate the current file. On the top right above the editor is a link to the language reference where most syntax and semantic features are explained in detail.

978-1-4799-3190-3/14/$31.00 ©2014 IEEE 3-5 April 2014, Military Museum and Cultural Center, Harbiye, Istanbul, Turkey
2014 IEEE Global Engineering Education Conference (EDUCON)

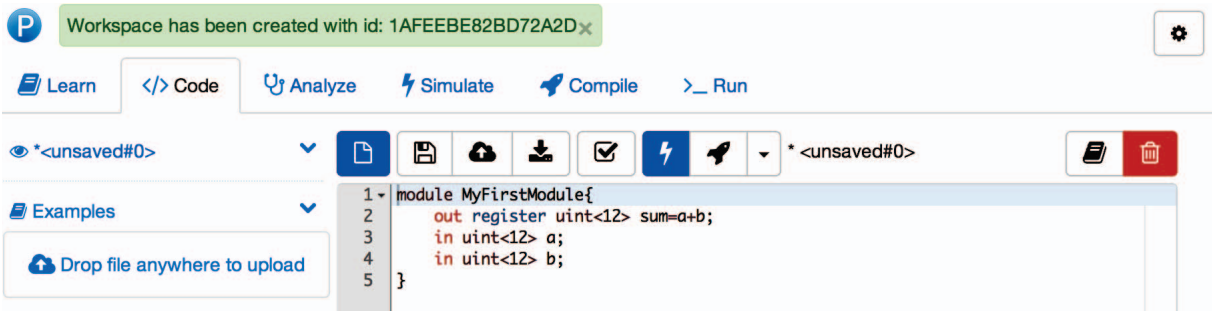

Fig. 1: beta.pshdl.org Screenshot

The fourth tab, labeled 'simulate', is where the currently edited file can be simulated. For this, the PSHDL file is converted to JavaScript, which is then executed with the user-provided inputs. To show the outputs, the user can choose between various number representations and LEDs. The fifth tab shows the resulting VHDL files when the user chose to run the compilation. He can then download either a single file, all generated files or the complete workspace.

The simulation part is a crucial component in lowering the barrier of entrance. When a user writes code, he probably wants to test whether it does what he hoped it should do. While one can edit VHDL files without installing the whole 20 GB tooling, the user has to install a simulator in order to verify his program. The online simulator allows the user to at least get a blinking LED running, before having to install the tools. While the last tab is called 'run', it is currently not functional and so the user has to install the tools locally for running synthesis and upload the results to the FPGA. In the future, all files could be generated from the web interface and a small local tool uploads them to the FPGA.

III. Data collection

In order to use PSHDL, a user is asked to create a workspace. Within this workspace, all files that are created or modified are stored on the server side. This workspace can later be re-opened by memorizing a unique workspace ID. When the user presses the 'save' button in the browser, the modified file will be sent to the server. There, the file will be checked into the versioning system and analyzed. The result of the analysis (syntax and semantic checks) will be sent back to the user and displayed as error marker in the editor. This implies that the user will get feedback upon each save, which he can then use to improve the code.

The general principle of the versioning system is shown in figure 2. When the user created a new document at T_1, he saved it, which created a new version A in the versioning system. At some point later in time, he added two more lines and introduced a syntax error. After saving(T_2), he gets feedback, sees the mistake, corrects it and saves again (T_3).

While the user is only able to see the latest version, all previous versions are available on the server. Those versions allow the analysis of the programming flow of a user.

A. Data sources

For this paper, the following user groups were analyzed:

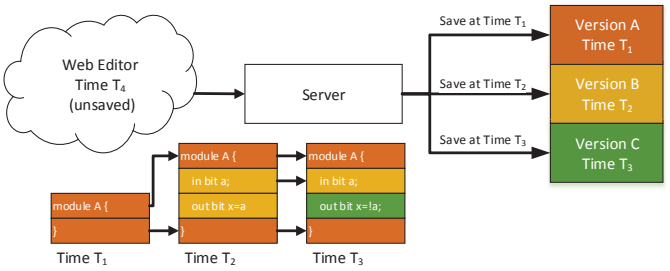

Fig. 2: File versioning

- Website users: those are random visitors of pshdl.org.
- Students of the hardware hands-on lab (HWP). During the HWP, those students have to solve increasingly difficult tasks. This is usually done in VHDL, but students were shown PSHDL and allowed to use it. This lab is mandatory for the students and takes place in their fifth semester of the Bachelor.

Generally, the students had attended lectures where VHDL or even Verilog were used. Most students that attended the HWP also had a VHDL hands-on lab the year before. The prior hands-on lab was based purely on simulation and did not focus on the synthesizable subset of VHDL.

B. Preliminary analysis

Website users generally loaded a few examples and hit the 'generate' button. From 440 analyzed workspaces, 159 did at least once save the default template. 106 workspaces loaded another example at least once. From those 440 workspaces, only 59 performed more than 10 save operations.

A first analysis pass on the data showed that some errors and warnings are occurring in groups. For example, when a new sub module is instantiated, several warnings about unused ports were generated. These warnings do not necessarily point to problems in understanding, but are rather an intermediate effect of working on the code. The same holds true for some *unresolved variables* errors which can easily be created in large numbers when a module is renamed. In one workspace, a peak of 54 *unresolved* errors and 51 *unused* warnings were generated in one version.

IV. Problem classification

The first step in validation is to parse the source. If an error occurs during parsing, the created errors are called syntax errors. After the source has been parsed successfully, an abstract syntax tree (AST) is created. As not all constraints can be placed in the grammar of a language, a semantic analysis has to be performed. While the failure to parse the source is always an error which terminates further processing, semantic problems can either be informative, warnings or errors.

A. Syntax and semantic errors

Examples of syntax errors:

```
//Missing semicolon
uint a
//Wrong order of register and type
uint register a;
//State names can not be numeric
enum states = {1,2};
//Unknown keyword, completely off syntax
typdef enum int{1,23,3};
```

Examples of semantic errors:

```
//Missing value for constant (Error)
//Unused variable (Warning)
const uint a;
//A port of direction in can
//      not be a register (Error)
//Duplicate variable a (Error)
in register uint a;
//Type names should start with upper-case
//Character (Information)
enum states = {A,B};
//Enums can not be used in arithmetic
//      operation (Error)
uint x=A+5;
```

Syntax and semantic errors are caught automatically, and are presented to the user. In some cases, good suggestions are possible on how to fix the error.

B. Logical errors

Unfortunately, a source that contains neither semantic nor syntax errors does not necessarily do what the user intended it to do. Those problems are called logical errors. For example, a user might want to develop a simple adder/subtractor:

```
module Adder {
    in int<8> a,b;
    in bit add;
    out int<8> sum;
    if (add)
        sum=a+b;
    else
        sum=a-b;
}
```

From just looking at this source, it is impossible to determine whether this code satisfies the requirements. For example, the signal *add* might be low-active. As such, the operations would be inverted. It might also be that the module is expected to use a register for the *sum* output. Thus, logical errors can only be caught by verification against the requirements. This can, for example, be done with a test-bench. Although this does not guarantee a 100% fulfillment of the requirements, it is a good indication of how well the design is meeting at least some requirements.

C. Paradigm related errors

A user with prior knowledge of a sequential programming language might make assumptions about the behavior of written code that differs from the actual behavior in a parallel paradigm language. An example is the following code:

```
module de.tuhh.ict.Timing {
    out uint a=1,b=2,c=3,d=4;
    a=b;
    b=c;
    c=d;
    d=5;
}
```

A user with prior knowledge in a sequential programming language might assume that the variables have the following values:

- *a* is first 1, then 2.
- *b* is first 2, then 3.
- *c* is first 3, then 4.
- *d* is first 4, then 5.

In fact, all variables have the value 5 all the time, because variables that are not registers can be thought of as a wire, rather than a storage element.

V. Data analysis

Syntax errors

With the versioning system, it is not only possible to take a look at how many errors/warnings occurred, but also to trace how they were dealt with. When you look at figure 2, you can see that the user introduced a syntax error in version *B* of his document. In version *C*, the problem has been fixed. An analysis of syntax errors showed that from 319 syntax errors found, the next edit fixed 229 of them. The remaining 90 were unsuccessful edits. This does not necessarily mean that the user understood the error and fixed the code, he could also have simply commented it out.

Examples of an unsuccessful edit history:

```
if LED_reg{2} == 1
  ERROR: missing '(' at 'LED_reg'.
if LED_reg{2}() == 1
  ERROR: missing '(' at 'LED_reg'.
if LED_reg(2) == 1
  ERROR: extraneous input 'LED_reg'
                    expecting '(' .
if 1==1
  ERROR: missing '(' at '1'.
if 1==1()
```

```
  ERROR: missing '(' at '1'.
if 1()==1()
  ERROR: missing '(' at '1'.
```

What he meant was:

```
if (LED_reg{2} == 1)
```

This edit history may seem odd for a C programmer, but it makes sense for a person that knows VHDL. In VHDL, the parentheses are used for accessing bits as well as array dimensions, also, in VHDL, parentheses around the if expression are optional. This is a clear case where the C-likeness did not increase writability, but caused the opposite effect. Also, the automatically generated error message can certainly be improved.

From all syntax errors, the missing semicolon accounted for 31%.

Semantic errors

In order to provide a rough overview of what errors have occurred, they have been combined into categories. The distribution of those errors can be found in table I. *Unused local* warnings are created by variables that are declared within the module, but are either written or read just once, or are not used at all. *Unused interface* warnings are generated for ports (variables with direction in/out/inout/parameter) that are either not read or written, depending on their direction. In contrast to local variables, those ports can be declared in another file and thus are not necessarily visible to the user. The category of *unfixable* warnings contains warnings that the user could not fix. The following example generates the warning: *possible range overlap*. For the following example, this means that PSHDL is unable to determine that the left expression OFF+8 is bigger than OFF, and thus cannot guarantee that the direction of the bit access is downwards. A new syntax has been introduced to solve this problem, but it has not been publicly documented yet.

```
module A {
    param uint OFF;
    in bit<32> inData;
    out bit<8> data=inData{OFF+8:OFF};
}
```

Unresolved errors are created when a referenced type cannot be found. This may happen for misspelled variables, types, or incorrect imports. *Unsupported operations* can occur when, for example, a bit vector which does not have a numerical representation is used in an arithmetic operation. *Duplicate names* are generated when a variable name is not unique. *Array* errors can occur for accessing non-existing dimensions, or exceeding the size of the array.

VI. Student Performance Analysis

The question is, can this data be used to evaluate the performance of students? In this section, three workspaces of students are analyzed. The first two students A and B participated in the HWP where they had to develop an LED that switches between 8 different levels of intensity every second. The intensity should be realized using pulse-width-modulation. The most simplistic solution in PSHDL can be done in about 5 lines of code.

Category	Occurrence
Unused local warnings	36,5 %
Unused interface warnings	36,0 %
Unfixable warnings	10,7 %
Unresolved errors	7,2 %
Unsupported operations	4,7 %
Duplicate names	1,8 %
Array errors	1,5 %

TABLE I: Relative occurrence of semantic errors

The third student C developed his diploma thesis in PSHDL. This took over four months, and the final code base had a substantial size of around 2500 lines of code.

For all students, a graph was plotted. The x-axis is measured in saved versions. That means it does not matter whether a user worked 10 seconds or 30 minutes between two save operations. The motivation for this is that some users might think more carefully about their modifications, while others prefer a quick feedback by saving more frequently. In the end, the user cannot determine for sure whether his modification is successful until he performs a save operation.

On the y-axis the following data is presented:

- Lines of code: this is the total number of lines of code that existed at that version. Empty and commented lines were ignored. The right-hand side axis applies to this data.
- Semantic errors: the number of semantic errors found in that version. The number of semantic errors is filtered from errors that occur regularly during development, such as *unused* warnings, *unfixable* warnings and *unresolved* errors.
- Syntax errors: the number of syntax errors in that version. It is important to know that when a syntax error occurred, the file is not further analyzed for semantic errors.
- Severe syntax errors: those are syntax errors that were not fixed with the next edit as outlined in section V.

Student A participated in the HWP. Within about 120 versions, he accomplished the goal of developing the dimmed LED as described in the exercise. The resulting code size of around 30 lines of code is bigger than required, and he did not perform a table lookup for the intensities. The workspace graph is shown in figure 3.

Student B also participated in the HWP. The solution as required per exercise was achieved with about 220 edits and 40 lines of code, and even some useful comments. The workspace graph is shown in figure 4.

Student C did not participate in the HWP, but was using PSHDL for his diploma thesis. He is included because the number of lines of code he produced differs significantly from student A and B, even for about the same amount of versions. The graph shown in figure 5 is scaled to match the graphs of students A and B. Notice how the number of lines of code is

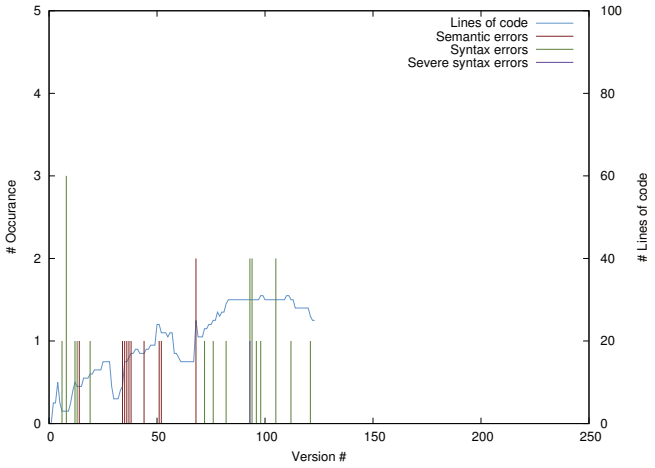

Fig. 3: Workspace of student A

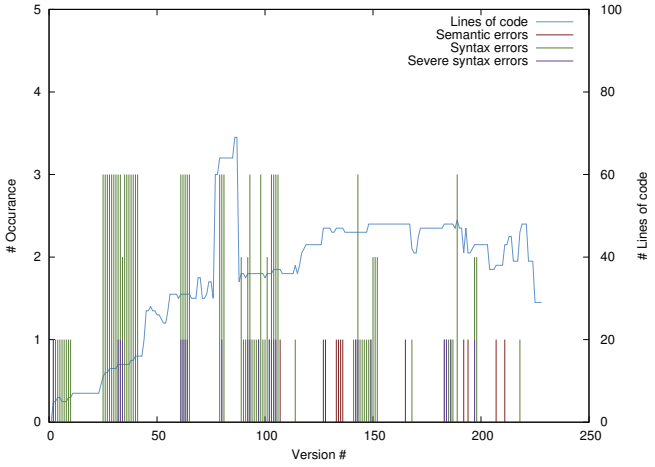

Fig. 4: Workspace of Student B

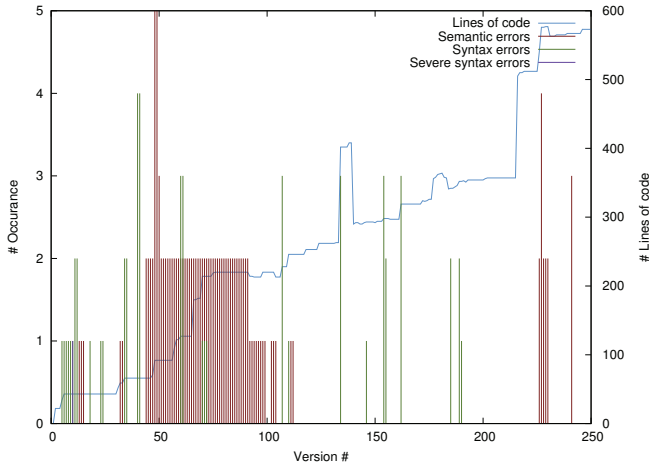

Fig. 5: Workspace of student C
(trimmed to match students A and B's graph)

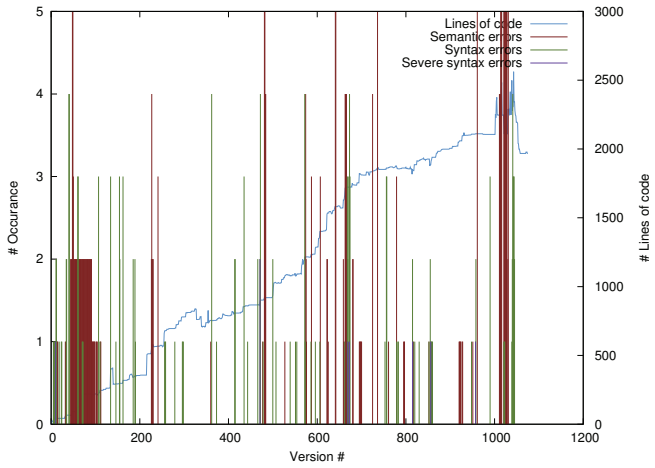

Fig. 6: Workspace of student C

about 10 times higher at the same version. The figure 6 shows the full workspace history.

A. Analysis

Student A and at least one other student created a paradigm-related error. In both cases, setting the duty cycle for the intensity was done in a for-loop.

```
module de.tuhh.hwp.PWMsmall{
    uint<8> dutyCycle=0;
    out bit<4> LED;
    register uint<8> period=period+1;
    for (i={1:8}) {
        dutyCycle=32*i;
        if (dutyCycle>=255)
            dutyCycle=0;
        LED{0}=period.counter
    }
}
```

When the student was asked why he thought that this might work, he said that he knew it wouldn't work in VHDL, but expected that PSHDL might do some magic to it. The problem appears to be the serial nature that a for-loop implies. In theory, it would be possible to present a warning or error if the index variable or a variable derived from it is not used on the left-hand side. This, however, has not been implemented in PSHDL yet.

Most students did use the online simulation at least once in order to test their code during development.

All students did encounter the majority of syntax errors in the beginning, which is expected, as they have not used PSHDL before. However, the amount of errors per lines of code varies significantly among the students. Student B, for example, frequently suffered from failed attempts to fix some syntax errors, while student C was very quickly recovering from most syntax problems. In general, however, it can be seen that syntax errors occur more frequently in the beginning. With the full workspace of student C as shown in figure 6, it can be seen that even after producing a substantial amount of code, there are still syntax errors occurring, however, most are small typos, or temporarily generated incomplete code to be

filled in later.

The graphs and manual analyses show that learning the syntax is the first hurdle in learning a new language, it is thus very important to provide good feedback when such an error is encountered. The error messages that are produced by standard parser generators such as ANTLR do not necessarily help the user to recover from the syntax error. For those cases, a good documentation is necessary. In the access logs of the web server, it could be found that the 'documentation' button was in fact accessed by the students. While the tutorial contains an example of a blinking LED, no evidence could be found that code from the tutorial has been used, although access to it can be found in the log files. Also, students have different means of dealing with an error. Student B, for example, saved quite frequently, sometimes up to 4 times per minute, to get feedback for each modification. Other students such as student C performed additional code modifications in other areas with each save and thus generally have a longer period in-between saves.

VII. Summary

The data also showed that providing clear error messages, especially related to syntax errors, is really important. As a result of this, the PSHDL parser has been improved to detect a few common mistakes and improve those messages. An interactive documentation that scans for keywords surrounding the cursor position could provide help about the language feature the user is probably most interested in.

Some errors, such as the *unresolved* errors or *unused* warnings that occur due to code reorganization, can be avoided by integrating refactoring tools.

Another lesson learned from analyzing the workspaces is that users are reluctant to report errors. Some versions in a workspace caused the compiler to crash, which is clearly visible in the workspace as error message. Yet none of the users reported a bug via the provided bug reporting mechanism. When feedback was provided, it was a rather casual oral feedback.

A. Future work

In the future, we plan a separate test setup with students unfamiliar with any hardware description language. This could provide more interesting data regarding the paradigm-related issues. Also, an increased number of subjects and a tighter, more strictly controlled environment would allow to create statistically relevant data.

B. Teaching with PSHDL

The server infrastructure of PSHDL provides interesting insights into the status of students. It can be seen what problems the students are struggling most with and what code is currently worked on most. In addition to the analysis, the web-based interface eases the interaction with the student. When a student has a question on a certain problem or phenomenon, he can send a link of the current workspace to the teacher or other students and ask for help. This could become even more helpful with an integrated chat system and multi-editing capability as provided by frameworks like shareJS.

As the simulation of PSHDL code can occur in the browser, custom visualizations can be created and made interactive to illustrate a point. A simple example is the LED, but it is also possible to visualize critical signals within a CPU, complete with an instruction de-compiler and online assembler to modify the code for that CPU.

From the data it could also be seen that students are willing to work outside of their lab times, which can be difficult when a campus-wide license is required in order to run the software. A direct collaboration between students by, for example, accessing another group's workspace could not be detected. With a continued development of the web interface, this collaboration could be encouraged by including a chat infrastructure.

The collected data allows to identify students that are struggling to achieve the desired results, and identify students that are doing well. In order to improve the learning experience, those students could be brought together, so that they could teach each other.

The version history can also provide insights into whether a student was working on the solution on his own, or whether he was simply copying the solution.

With all this data available, it is important to explain to the students what data is collected and how it is going to be used.

Acknowledgment

I would like to thank the students that worked on their workspaces. I also would like to thank Cem Bassoy, Henning Holm and Inez Mischitz for their valuable feedback on this paper.

References

[1] I. Utting, N. Brown, M. Kölling, D. McCall, and P. Stevens, "Web-scale data gathering with bluej," in *Proceedings of the ninth annual international conference on International computing education research*, ser. ICER '12. New York, NY, USA: ACM, 2012, pp. 1–4. [Online]. Available: http://doi.acm.org/10.1145/2361276.2361278

[2] (2014, Jan). [Online]. Available: http://www.edaplayground.com

[3] I. T. C. Mow, "Analyses of student programming errors in java programming courses," *Journal of Emerging Trends in Computing and Information Sciences*, vol. 3, no. 5, pp. 739–749, 2012.

[4] A. Etxebarria, I. J. Oleagordia, and M. Sanchez, "An educational environment for vhdl hardware description language using the www and specific workbench," in *Frontiers in Education Conference, 2001. 31st Annual*, vol. 1. IEEE, 2001, pp. T2C–2.

[5] Y. Rajasekhar, W. V. Kritikos, A. G. Schmidt, and R. Sass, "Teaching fpga system design via a remote laboratory facility," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 2008, pp. 687–690.

[6] J. S. Pastor, I. Gonzalez, J. Lopez, F. Gomez-Arribas, and J. Martinez, "A remote laboratory for debugging fpga-based microprocessor prototypes," in *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on*. IEEE, 2004, pp. 86–90.

[7] N. Nisan and S. Schocken, *The elements of computing systems: Building a modern computer from first principles*. The MIT Press, 2005.

[8] S. Gehring, S. Ludwig, and N. Wirth, "A laboratory for a digital design course using fpgas," in *Field-Programmable Logic Architectures, Synthesis and Applications*. Springer, 1994, pp. 385–396.

[9] (2013, 11). [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[10] (2013, 11). [Online]. Available: http://beta.pshdl.org