Pyrope Usability and Efficiency versus Verilog

Teddy Sudol
The College of New Jersey
2000 Pennington Road
Ewing, NJ
sudolt1@tcnj.edu

ABSTRACT

Pyrope is a new hardware description language designed to bring modern language features and expressiveness to HDLs. The goal of this project was to evaluate its usability for programmers who are not experienced with other HDLs. Use cases were written to compare Pyrope to Verilog, another popular HDL. Pyrope programs were found to require fewer lines of code and less development time than Verilog programs. These results suggest that Pyrope is an efficient and highly usable language for new HDL programmers.

1. INTRODUCTION

Hardware Description Languages (HDLs) are programming languages used to develop and simulate digital hardware systems. By providing language constructs that abstract away the basic aspects of digital design, designers can be more productive and focus on the big picture of large-scale digital systems.

However, current hardware description languages do not have the modern programming constructs of other modern programming languages. Pyrope, a new HDL, was created to address this issue. Inspired by Python and Ruby, Pyrope is a high-level, expressive language that incorporates modern language constructs such as global type inference. It includes features that are specific to the domain of HDLs; for example, Pyrope automates clock and reset signal management. The goal of the Pyrope project is to modernize hardware description languages and bring new features to this domain. Section 3.2 explores the features of Pyrope in depth.

Previous work with Pyrope compared it to Verilog and other HDLs in terms of program length (lines of code). (See section 2) Another angle of comparison is the usability of Pyrope. Specifically, is Pyrope easier to use than Verilog for someone who has never used either one?

These questions were investigated by writing use cases. Use cases are simple programs that succinctly show capabilities of a language or other system in a limited environment. They capture the pith of a language without requiring exhaustive coverage of its features. Pyrope and Verilog were compared by two metrics: the length of the programs in lines of code and the length of time required to write the program. The particular use cases are discussed in section 3.3, and section 4 compares the results thereof.

2. RELATED WORK

Much of the work on Pyrope is encapsulated in Blake Skinner and Dr. Jose Renau's paper *Pyrope* [5]. When compared with other hardware description languages, namely Verilog and Chisel, Pyrope was found to create shorter programs. The reduction compared to Verilog programs ranged from 37% (an implementation of a Greatest Common Divisor (GCD) circuit) to 90%. The reduction compared to Chisel was more dramatic and ranged from 11% (again the GCD circuit) to 92%. (In one case, a filter circuit, the Pyrope and Verilog programs were the same length while the Chisel program was longer.) On average, Pyrope showed an 81% reduction versus Verilog and a 56% reduction versus Chisel.

3. METHODOLOGY

3.1 Verilog

At first blush, Verilog's syntax bears a strong resemblance to C code. While the language does have an imperative feel, it has several core features that differentiates it from other programming languages. Verilog programs focus around modules and the interactions between them. A module defines a circuit or subsystem. The definition includes the circuit's inputs, outputs and internal registers. Another section of the module lists the circuit's behavior. This section is similar to C code and uses the usual paraphernalia of conditional statements, variables and function calls. Each module can be linked to other modules to make a complete system [2].

Verilog requires explicit typing for its variables. Each input, output and register is assigned a type, which is often a certain length of bits. For example, an 8-bit integer input called `x` is defined by
`input [7:0] x;`
Output values and registers are similarly defined with the `output` and `reg` keywords. Single-bit variables simply discard the `[size]` component. Every variable must be explicitly typed in the module.

The behavioral section of the module definition generally starts with a timing signal. For example, `always @ (posedge clk or posedge rst)` indicates that the module activates when the incoming clock (here `clk`) or reset signal (`rst`) is on its rising edge or transitions from `0` to `1`. This connects the behavior of the module to the clock the manages the whole system.

In addition to modules, Verilog provides functions and tasks. These two constructs have a few key differences. Both can have any number of inputs, but tasks can have any number of outputs while functions are limited to only one. Tasks can also include timing delays, i.e. they can be linked to the clock with `posedge` statements. Functions, on the other hand, cannot have timing delays.

3.2 Pyrope

Pyrope is designed around the pipeline programming model. In this model, data flows from operation to operation; the output of one operation is the input to the next. Each operation processes the data in some way. Digital systems are therefore represented as a series of transformations over the input [5].

Pyrope uses three top-level blocks to define the pipeline: stages, pipes and classes. Stages are analogous to the operations of the pipeline. The inputs and outputs of the stage are implicitly declared; any value that is read is an input, and any value that is written to is an output. For example,
`A = B + C`
implicitly declares `B` and `C` as inputs and `A` as the sole output. Unlike Verilog, the reset and clock inputs are handled automatically by Pyrope, and they do not need to be declared as inputs to the stage. The reset signal is occasionally defined explicitly to elucidate how the stage should be reset.

The second block type is the pipe. Pipes, as the name suggests, define the actual data pipeline of a program. A pipe connects stages together to make a full program by listing the order of the stages in the pipeline. Each stage is connected to the next using the `->` operator. This operator automatically connects the output of the first stage to the inputs of the second, though these connections may be listed explicitly. Pipes can connect stages in parallel and even link other pipes together in a nested configuration.

Classes, the third and final top-level block, are the core of Pyrope's object-oriented programming model. Classes can define class variables (preceded by an @ symbol) and methods, similar to Ruby classes. Pyrope also includes traits for further extending the functionality of classes and objects by adding single-instance functionality.

In stages, inputs and outputs are declared when they're used. In pipes, inputs and outputs are linked together stage-by-stage. However, the sizes of these variables are not explicitly defined anywhere! Thanks to Pyrope's global type inference, the programmer never needs to explicitly declare the size of variables. Instead, the Pyrope compiler determines the correct sizes for the variables while processing the file. If the compiler sees `A = B + C`, for example, it can infer that `A` must be large enough to contain both `B` and `C` and therefore makes `A` as large as the sizes of `B` and `C` combined. Pyrope has other operators to handle situations such as overflow; `A = B :+ C` is an overflow add of `B` and `C`, and `A` will be the same size as the larger of `B` and `C`. This allows, for example, 8-bit counters to always be 8-bits long instead of growing depending on the input size. Otherwise, the compiler would complain that the operation would change the size of the counter, which is a fatal error in a strongly-typed language like Pyrope.

Of course, the type inference process must have some base sizes to build upon. At some point the programmer must define the sizes of `B` and `C` if the compiler is to infer the size of `A`. For example, three stages `S0`, `S1` and `S2` are defined with no explicit types. Then two pipes are written. Pipe `P0` links the three stages together and declares their input values as 4-bits wide. Pipe `P1` also links the three stages, but instead lists their input values as 8-bits wide. The compiler can use the same three stages to make two different pipelines that only differ in terms of the range of values they can process. If the programmer wants to use 16-bit integers or some other size, they simply create a new pipe with a new list of input sizes. The compiler takes these input dimensions and infers the sizes of all the variables and values used in the stages themselves. That is the power of the global type inference.

3.3 Use Cases

3.3.1 Booth's Multiplication Algorithm

Booth's Multiplication Algorithm is a method for multiplying two signed integers using only addition and bit shifting [3]. Due to the simplicity of its operations and design, it is a useful algorithm for teaching hardware description languages.

The algorithm takes as input two numbers of arbitrary length—the multiplier and the multiplicand. The result has a length equal to the sum of the lengths of the two inputs. The calculation is performed by adding either the multiplicand or the negated multiplicand to the multiplier based on the final pair of bits of the multiplier. The result of the addition is then shifted once. This algorithm is easily represented in hardware by splitting the process into two stages, namely "add" and "shift".

3.3.2 Elliptic Curve Cryptography

Elliptic Curve Cryptography is a public-key cryptography method that uses the complexity of elliptic curves to secure information. Like the Diffie-Helman (DH) encryption method, it relies on the difficulty of finding discrete logarithms as its basis. ECC promises more security than other algorithms for the same key size, thus making it an important facet of security research. Please refer to *An Elliptic Curve Cryptography (ECC) Primer* for more information about this system [1].

An HDL implementation of the ECC algorithm can be somewhat complex. The algorithm is based around modular arithmetic, and special operations must be written for each step, up to and including the `mod` function itself. (While this function may be built into the language, a version that can handle negative numbers is required by algorithm and may not be available by default.) Fortunately, elliptic curve cryptography is based around discrete logarithms and thus does not require floating-point values, which reduces the overall complexity of the implementation. (Pyrope, for example, does not currently have floating-point capabilities.)

3.4 Metrics

As mentioned in section 1, the use cases evaluate Pyrope and Verilog by two metrics: lines of code and length of development time. These two metrics are directly related to the language characteristics that we are investigating, namely expressiveness and usability. A more expressive language requires fewer lines of code to accomplish some goal as compared to other languages. "Usability" for languages relates to how easily a programmer may begin using it and how easily a generic program is written in a particular language. For example, Verilog's explicit typing (i.e. `input [7:0] x`) may make the language seem less usable when compared to Pyrope's type inference. The programmer can accomplish more—and thus have a lower development time—if the language has high usability. Obviously, these two traits are connected, given that a more expressive language will enable to programmer to write less, thus making it more usable. The two chosen metrics can be seen as

	Pyrope LoC	Verilog LoC	Reduction
Booth	20	29	31%
ECC	84	45*	N/A

Table 1: The Pyrope use cases were shorter than their Verilog analogues.

	Pyrope Time	Verilog time	Reduction
Booth	12 min.	20 min.	40%
ECC	20 min.	20* min.	N/A

Table 2: The Pyrope use cases required less time than the Verilog implementations.

direct expressions of the expressiveness and usability of the target languages.

3.5 Development Tools: Syntax Highlighting

One mild difficulty of writing programs with Pyrope was the lack of syntax highlighting. Syntax highlighting is a common feature in many text editors and IDEs that colors or highlights different syntactical elements of the program. The highlighting is often accomplished by a script that identifies the various lexical elements and the colors that should be used. The simplest syntax highlighting scripts identify language keywords, but they can be expanded to include function and variable names, data type identifiers, comments, string literals and numeric values. Each IDE and editor usually has its own scripting language or other method for creating syntax highlighting scripts.

Because most Pyrope code is currently written with Vim, the first syntax highlighting script was written for that editor. Syntax highlighting scripts for Vim use the editor's default scripting language, VimScript (or VimL) [4]. `syn` statements (short for "syntax") define the lexical elements to be highlighted; for example, the line:

```
syn keyword pyropeBlock class stage pipe
```

defines a group of keywords called "pyropeBlock" that matches "class", "stage" and "pipe". This group is later used in a "`hi`" statement (short for "highlight") to indicate how the keywords should be colorized:

```
hi def link pyropeBlock Keyword
```

The above line tells the editor to use the "Keyword" colors to highlight the "pyropeBlock" set of lexical elements. The colors used are defined in a separate file that specifies a full color scheme.

`syn` statements can be used for more than just language keywords. The "match" keyword, when used in `syn` statements, allows regular expressions to define syntax patterns. These can be used, for example, to highlight numbers, strings and comments. These syntax matches are somewhat more complex than the simple keyword matches. For example, the pattern for hexadecimal numbers originally looked for the letters a-f and the digits 0-9. However, this would cause any variables named 'a', 'b', etc. to be colored as numbers. Additionally, Pyrope variable declarations could have an initial value for the variable by including a base indicator (such as 'h' for hexadecimal) and an initial value, e.g. `h10ab`, which simply reused the pattern declared for regular hexadecimal numbers. This was solved by redefining the hexadecimal number pattern to always look for a leading `0x` (e.g. `0x10ab`) and rewriting the variable declaration pattern with the number pattern included.

4. EVALUATION

4.1 Lines of Code

As explained in section 3.4, the length of the program in lines of code, or LoC, shows how expressive a language is over the length of a whole program. A more expressive language will require fewer lines of code to accomplish the same goal than a less expressive language. Table 1 contains the lines of code measurements for the two use cases.

In the Booth's Multiplication Algorithm example, the Pyrope implementation was found to be only 20 LoC versus 29 LoC for the Verilog version. The resulting 31% reduction is in line with the measurements found in other Pyrope vs. Verilog examples, as mentioned in section 2.

The Elliptic Curve Cryptography use case shows a very different result. The full Pyrope implementation was 84 lines long, while the Verilog implementation ran to only 45 lines. However, the Verilog implementation is essentially unfinished and, as such, the two implementations cannot be compared effectively. Section 4.4 discusses the issues associated with this use case.

Another metric for evaluating expressiveness in a programming language is line length. While it was not a goal of this project to evaluate the target languages for this metric, it is informally discussed in section 4.3.

4.2 Development Time

The development time for a program, or how much time is needed to write the whole program, is related primarily to how easy a language is to use. Easier languages mean more productive programmers which in turn means each program has a shorter development time. This metric can also correlate to the expressiveness and abstraction capabilities of a language. This can also be expressed by how "high-level" a language is. More expressive languages require less code to be written. Table 2 shows the comparison of this metric in the use cases.

The Pyrope implementation of Booth's Multiplication Algorithm was written significantly quicker than the Verilog implementation. On the other hand, the Elliptic Curve Cryptography programs were identical in terms of time. However, the caveat mentioned in section 4.1 also applies for this metric.

One obvious issue with this metric is bias introduced by solving the same problem twice. Verilog and Pyrope are not so dissimilar that code written in one is not easily translated into the other. This is more fully discussed in section 4.4.

It should be noted that development time as a metric can be highly relative. The resulting measurements are highly individualistic in that several factors arise from the programmer, not the language. The programmer's experience, both with programming in general and with hardware description languages, is by itself a large factor in how long a particular program will take. However, all of these factors should be constant across the use cases and thus give consistent results.

4.3 Line Length

Line length is the number of characters in a line of a program. As with the others, this metric is largely related to the expressiveness of the language in question. Expressiveness is effectively, "doing more while saying less" when it comes to programming languages. For the purposes of this project, the average line length of the use cases is being considered for informal purposes only as an interesting comparison between Pyrope and Verilog.

In the Booth's Multiplication Algorithm programs, the longest Pyrope line was 24 characters versus Verilog's 43, excluding leading whitespace. The lines in question:
Pyrope: `elif (P&1, b) == (1, 0):`
Verilog: `end else if ((p&1) == 1 && last == 0) begin`
Interestingly, these lines accomplish the same thing: Both are else-if conditional statements comparing a pair of bits. These lines also show how heavily language-syntax and -feature dependent line length is. The Pyrope example uses a tuple to succinctly compare the two pairs of bits while the Verilog example has to compare the bits individually. Additionally, where Pyrope can simply use `elif`, Verilog requires `else if` along with an `end` keyword to indicate the end of the previous if-block.

It should be expected that the two longest lines in the Elliptic Curve Cryptography programs are similar, as was seen for the Booth's Multiplication Algorithm use case. Due to the incompleteness of the Verilog example, this condition is difficult to follow. The Pyrope implementation uses an object-oriented approach, which lead to several long lines due to using qualified names when calling class functions. In the sections of the algorithm that both implementations have, the longest lines are nearly identical in length. This is due to the highly mathematical nature of the Elliptic Curve Cryptography algorithm. Considering that the operators are the same between the languages, the differences in the lines comes down to the variable names used. Considering that the lines are otherwise nearly identical, the line length metric provides no interesting insights into the languages for this use case.

4.4 Limitations

One issue of these use cases is the issue of repetition bias. The use cases are designed to evaluate how effectively a programmer can use the language to solve a given problem. They solve the problem once by writing the Pyrope implementation. When they go to write the Verilog version, they're no longer solving the problem and are instead translating their Pyrope code into Verilog. This introduces bias into the measurements. When writing the first implementation, some of the time might have been spent solving the problem, and that time is saved when developing the second version. This could make the second language seem better than the first language.

In the project, the bias was mitigated by alternating the first language used. For the Booth's Multiplication Algorithm use case, the program was first written in Verilog and then in Pyrope. The order was then switched for the Elliptic Curve Cryptography use case. While this cannot fully eliminate the bias, it can at least reduce its affect on the data.

The lack of experience in the languages is another possible concern. Verilog, in particular, is a mature and widely-used language. The code written by the programmers for these use cases could be longer and less efficient than the code written by an experienced user of Verilog. This largely applies to the lines of code metric, since an expert would be familiar with potential shortcuts and other efficiencies in the language. Development time could potentially go either way; if the programmer could not adapt to Pyrope's pipeline programming model, the development time data would show longer Pyrope development times than Verilog.

As mentioned in previous sections, the Elliptic Curve Cryptography use case presented some difficulty to the project. The complexity of the topic was the first hurdle. ECC requires a nontrivial amount of knowledge and familiarity with modular arithmetic and cryptography, which was outside the scope of the project. Development proceeded with guidance from online sources, which could be easily applied to Pyrope but not to Verilog. Due to these conditions and time constraints, the Verilog implementation was not finished. This lead to the incomplete data seen in tables 1 and 2 and the lack of comparisons between Pyrope and Verilog for the ECC use case. From the time comparisons we see that, in the same amount of time, the Pyrope implementation was completed but that Verilog one was not. This suggests that the, had the Verilog version been completed, the trend seen in the Booth's Multiplication Algorithm case would have continued.

5. CONCLUSION

From the data above it can be seen that, for programmers who have never used either language before, programs written in the Pyrope hardware description language are shorter and take less time than those written in Verilog. The trend with program length is corroborated in the original Pyrope paper [5], though this project added the metric of development time. While these use cases do have limitations, including some amount of bias, they argue for Pyrope as an efficient and highly usable language for HDL beginners.

6. REFERENCES

[1] An elliptic curve cryptography (ECC) primer. Tech. rep., Certicom Corp., June 2004.

[2] IEEE Standard for Verilog Hardware Description Language. *IEEE Standard 1364-2005* (2005).

[3] BOOTH, A. D. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics IV* (1951).

[4] MOOLENAAR, B. *Vim Manual*, April 2011.

[5] SKINNER, B., AND RENAU, J. Pyrope.