Comparing Hardware Description Languages

Matthijs Hofstra
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.hofstra@student.utwente.nl

ABSTRACT
We describe a comparison of several existing hardware description languages to determine their advantages and disadvantages with regards to abstraction, code reuse, and usability. To make this comparison we will implement a FIR-filter and a FIFO-queue in every language and evaluate the result according to these criteria. The languages that will be compared are VHDL, CλaSH, Kansas Lava and MyHDL.

Keywords
hardware description language, vhdl, clash, kansas lava, myhdl

1. INTRODUCTION
There are many different hardware description languages (HDLs) in use today. Some of them, such as Verilog and VHDL, have been around for decades. Others, such as CλaSH have been created much more recently. While the choice of language does not generally restrict the functionality of the end-product, it may affect the speed of development readability, re-usability, and the number of bugs. Moreover, the choice of language may encourage or discourage certain styles of programming.

1.1 Hardware Description Languages used in this paper
In the following sections the HDLs VHDL, CλaSH Kansas Lava and MyHDL will be introduced, with a quick description of each. While many other HDLs exist, we cannot evaluate them all and will limit our scope the these four. We will include our motivation for choosing each language in the descriptions below.

1.1.1 VHDL
The development of VHDL was initiated by the U.S. Department of Defense, with the first standardized version appearing in 1987. The syntax of VHDL is strongly influenced by Ada, a programming language language that was also created by the Department of Defense, and it was initially intended as a documentation language, with the options of simulating and synthesizing a VHDL description being added later. Since its inception VHDL has undergone several revisions. VHDL has some support for generic types.[6] VHDL was chosen because it is one of the most popular HDLs in use today (along with Verilog) and because it has been in use for a long time. We consider it to be important to include at least one widely used language so that those who are unfamiliar with less popular languages have a point of reference.

1.1.2 CλaSH
The CλaSH (CAES Language for Synchronous Hardware) was created in 2009. It uses the syntax and semantics of the functional programming language Haskell and can translate a functional description of the hardware to VHDL, which can then be synthesized by existing VHDL tools[1]. Because the hardware description is also executable Haskell, testing and simulating functions is easy and can be performed without translating to VHDL. Using polymorphism and higher-order functions is possible. CλaSH was chosen because it is very young compared to VHDL, and is based on a functional programming language. Because CλaSH offers very powerful abstractions that VHDL does not, we expect a comparison between the two to yield interesting results.

1.1.3 Kansas Lava
Like CλaSH, Kansas Lava is a HDL based on Haskell, and the two have many benefits in common, such as being able to test the hardware descriptions by executing them in Haskell. Kansas Lava also produces VHDL code.[3] Kansas Lava was also chosen because it is based on a functional program language. By including two such languages we hope to get a better illustration of what functional programming has to offer in the field of hardware description languages.

1.1.4 MyHDL
MyHDL is a hardware description language that is implemented as a Python[5] package. It models components as Python procedures and can translate to both VHDL and Verilog. Data is passed between components as signals, which have an immutable current state, and a mutable next state. Concurrency of components is modeled with resumable functions, and a description in MyHDL is executable Python code that can be simulated without translating to VHDL or Verilog.[2]
MyHDL was chosen because we suspect it might be an interesting direction that HDLs might take. It is more closely related to VHDL than to CλaSH, but dissimilar enough that we expect to find significant differences in usability. Because Python is a relatively popular programming language, it has a large number of potential users.

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. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2012, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
2. PROBLEM STATEMENT

Because of the large number of different HDLs available it can be hard to decide which one is suitable in a specific situation. VHDL and Verilog have become the default language when it comes to hardware description, and without data that compares these languages to alternatives it is difficult to make an informed decision to deviate from this standard. This is the area where we wish to contribute by answering the following research question:

- How do differences in syntax and semantics between hardware description languages affect their usability by programmers?

This question could be answered by describing the same component in several languages and comparing the results. We intend to evaluate usability on these criteria:

The conciseness of the description. A simple measure of conciseness would be to count the lines of code and the number of words used in each description, and that is the measure we intend to use.

The comprehensibility of the description. The comprehensibility could be measured relative to similar designs in other languages, by asking people to choose which of the designs they find easiest to understand.

The re-usability of the description. To judge the re-usability of a defined component we will attempt to instantiate it with different parameters. If a component can easily be reused with different parameters (such as the length of a queue, or by changing some constants) that component is more useful than a component that cannot be reused.

For this research we will not be considering the performance and size of the hardware produced by the different languages.

3. RESEARCH APPROACH

For the case study we will produce hardware descriptions of a FIR-filter and a FIFO-queue. For the FIR filter we will attempt to create a version where the user can specify the type of the input, the number of samples, and the weights. For the FIFO queue we will aim to create a generic version that will allow the user to specify the length of the queue and the type of data.

When we have created the hardware descriptions in each language, we will compare them based on the criteria mentioned in our problem statement. Conciseness will be tested by counting lines and words of the resulting descriptions. For the comprehensibility we will ask two groups, one consisting of people with prior experience in the field of hardware design, and one group without such experience. To make sure these people focus on the syntax of the language we will not include any comments that suggest how a particular action is being performed. However, we will attempt to use descriptive names where possible.

After we have collected all the data we will present the resulting hardware descriptions and the results of the comparison. If significant differences are detected, we will attempt to explain these differences by highlighting the relevant capabilities and limitations of the languages involved.

3.1 FIR-filter

A FIR (Finite Impulse Response)-filter is a type of filter that is used in signal processing. The general structure of a FIR-filter can be observed in figure 1.

\[
x[n] \rightarrow z^{-1} \rightarrow \sum b_i \rightarrow z^{-1} \rightarrow \sum b_i \rightarrow z^{-1} \rightarrow \sum b_i \rightarrow y[n]
\]

Figure 1. The structure of a FIR-filter

In the figure, \(x[n]\) is the input of the filter, and \(y[n]\) its output. The output depends on the current value of \(x\), the \(n\) values that preceded it, and \(n\) filter coefficients \(b_0, b_1, ..., b_N\). The output is calculated as follows:

\[
y[n] = \sum_{i=0}^{N} b_i x[n - i]
\]

3.2 FIFO-queue

A FIFO (First In, First Out)-queue is a common type of queue from which items emerge in the same order that they entered the queue. FIFOs are often used as buffers.

4. RESULTS

First we will for each language a short description of the impressions we had while implementing the components. We will also show the most important parts of the code. Details such as importing of libraries will be omitted. At the end of this section all languages will be compared to one another.

4.1 VHDL

Overall the implementation of both components was relatively easy in VHDL. While it is by far the oldest language of the four we have evaluated, we did not find it to be inferior in any way. The definitions of the components were somewhat long, but not to the point where this reduced clarity. Surprisingly we also found VHDL to be easiest language to use. The VHDL components took the least time to write by far. VHDL has a relatively simple syntax and very decent documentation. There were no complicated new concepts to learn and the language was easy to pick up.

In the FIR-filter the verbosity of VHDL is most obvious. The actual behavior of the component is quite simple and can, as is shown below, be described in 14 lines, but the actual source file is 36 lines in length. The other 24 lines are used to include libraries, define types, describe the generic parts of the component, and name the ports that will be used as input and output. The component accepts input from a signal \(in1\), and outputs to \(out1\). Note that while this component will accept integers of any size, it cannot handle floating point numbers. As far as we are aware VHDL generics are not powerful enough to define a component that can be used with both integers and reals. It would be possible to define a component that is an exact copy of the one below except that it would accept real numbers. We do not consider this a desirable solution as it results in code duplication.
Listing 1. The behavior of the FIR-filter
1 architecture Behavioral of fir is
2 signal data: std_logic_vector (sig_width *
3   coefs ’length downto 0) := (others => ’0’);
4 begin
5   data <= in1 & data(coefs ’length −
6   sig_width downto 0) when rising_edge(
7   clk);
8   mult_add: process (data) begin
9     variable tmp: integer;
10    begin
11       tmp := 0;
12       for i in 0 to (coefs ’length − 1) loop
13          tmp := tmp + to_integer (signed (data (sig_width *
14            (i+1) downto sig_width
15            * 1)) * coefs (i));
16       end loop;
17       out1 <= std_logic_vector (to_unsigned (tmp, sig_width ));
18    end process;
19    end Behavioral;

Listing 2. The behaviour of the FIFO-queue
1 architecture Behavioral of fifo is
2 signal index: integer;
3 signal data: std_logic_vector (dszie *
4   slots downto 0) := (others => ’0’);
5 begin
6   process (clk) begin
7     if activate = ’1’ and enq_or_deq = ’1’
8        and index + 1 < slots then
9        index <= index + 1;
10       empty <= ’0’;
11      else
12        full <= ’1’;
13    end if;
14    data(dszie−1 downto 0) <= in1;
15    data(dszie * slots downto dszie) <=
16    data(dszie * (slots − 1) downto 0);
17    out1 <= data(dszie * (index+1) downto
18            dszie * index);
19    elsif activate = ’1’ and enq_or_deq = ’0’
20           and index > 0 then
21       index <= index − 1;
22      else
23        empty <= ’0’;
24    end if;
25    out1 <= data(dszie * index downto
26            dszie * (index − 1));
27    end if index = 0 then

Listing 3. FIR-filter in CaaSH tabsize
1 #ent = fir testData testData
2 1 fir xs ys = fir ’ ’ ’ ’ (xs, ys)
3 4 firi : (State (xs, hs)) x = (State (shiftInto x xs hs), vfoldl (+) 0 (vzipWith (+) (shiftInto x xs hs)))

4.2 CaaSH

Working with CaaSH was significantly harder than we originally anticipated. While many Functional Programming features are supported, CaaSH cannot use standard Haskell lists. As an alternative one can use Vector, which is a fixed-size array. However, the length of a Vector is part of its type, which can lead to complicated situations when converting to Vectors of a different length. Taking some number of elements from a Vector, or concatenating two Vectors can confuse Haskell’s type-system, and the solution is not always obvious. Solving these problems proved difficult and required familiarity with Haskell’s (complicated) type-system. We can imagine most users would prefer not to spend their time struggling with the language and learning advanced functional programming concepts.

We found the way CaaSH handles State to be very pleasant and intuitive to work with, especially in comparison to the other languages. One can usually tell by one look at a function what it does and does not keep track of, and change in state is very explicit and noticeable to the reader.

The FIR-filter in CaaSH is shown below. It is immediately obvious that this description is much shorter than the VHDL. All the behavior is described in the single line that defines fir’. The component is initialized with two arrays, one with the initial values of the registers and one with the coefficients. The resulting component will accept an input signal and generate output. This component should be able to handle any number.

The CaaSH version of the FIFO-queue is slightly longer. We regret that we were unable to include bounds checking, it turned out that it is hard: it requires comparing some numbers of incompatible types. As can be seen on line 1 of the code below, even the type signatures of very simple
code can quickly become quite complex when Vectors are involved. When too many enqueue or dequeue actions are performed the statement \(xs!count\) will result in an exception as \(count\) goes out of bounds. We are unsure what the effect of such an event would be in actual hardware. As can be seen in the code below, we have used pattern matching to simplify the code.

Listing 4. FIR-filter in C\(\lambda\)Sh tabsize


\[
\begin{align*}
1 & \text{fifo} = \text{Fifo} \quad \text{State} \quad (xs, count) \\
2 & \text{fifo} = \text{Fifo} \quad \text{State} \quad (xs, a) \\
3 & \text{fifo} = \text{Fifo} \quad \text{State} \quad (xs, count + 1) \\
4 & \text{fifo} = \text{Fifo} \quad \text{State} \quad (xs, count - 1) \\
\end{align*}
\]

4.3 Kansas Lava

We found Kansas Lava to be better than C\(\lambda\)SH in some aspects, and worse in others. The way Kansas Lava handles state is more advanced than C\(\lambda\)SH it uses registers that are linked to a clock, and registers can all use different clocks. However, this has also added much complexity which can be confusing. This is made worse by the fact that the presence of state is slightly more easily overlooked because it can be located anywhere in a function, whereas in C\(\lambda\)SH it is usually obvious by looking at the first line of a function.

Kansas Lava does not use Vector, but uses a type called Matrix for its arrays. As the name suggests, matrices can have rows and columns, and has several functions to facilitate performing operations on a 2D-matrix. In our experience Vector was easier to use, but would like to add that 2D-matrices were not required in the FIR-filter and the FIFO-queue, so we have not had a chance to make a proper judgement on this subject.

Kansas Lava was able to do some things that the current version of C\(\lambda\)SH could not, for instance chaining together a list of functions that was used in an early implementation of the FIFO-queue. Overall the language was noticeably more mature. As with C\(\lambda\)SH knowledge about Haskell, especially concerning types, is a requirement, especially when working with matrices.

Kansas Lava was by far the most complex language and we did not manage to produce a working FIR-filter. The FIFO-queue that we did manage to create works every clock tick, as opposed to the FIFO-queues in the other languages, which can be activated and offer a choice between enqueue and dequeue. The code can be seen below. Line 9 is especially interesting as it shows that Kansas Lava is able to use ordinary Haskell lists in the hardware descriptions, something C\(\lambda\)SH could not do.

Listing 5. Incomplete FIFO-queue in Kansas Lava

```
1 shiftInto :: (PosT s, Natural s, n \(
2 & \text{Pred } s, \text{ Succ } n) \Rightarrow a \rightarrow \text{Vector } s a
3 shiftInto x xs = x \mapsto \text{vinit} \space xs
4 data EnqOrDeq = Enqueue | Dequeue
5
6 {-# ANN ent TopEntity #-} ents
7 ent = \text{fif testData}
8
9 \text{if } xs = \text{fifo } \mapsto (xs \mapsto 00) \mapsto \text{xs } 0
10
11 \text{fifo} (\text{State} \quad (xs, count)) (\text{False}, \text{False}) = (\text{State} \quad (xs, count), \text{x } (count))
12 \text{fifo} (\text{State} \quad (xs, count)) (x, \text{True}, \text{Enqueue}) = (\text{State} \quad (\text{shiftInto} \space x \space \text{x} \space \text{count } + 1),
13 \text{xs } (\text{count } 1))
14 \text{fifo} (\text{State} \quad (xs, count)) (x, \text{True}, \text{Dequeue}) = (\text{State} \quad (xs, count - 1), \text{x } (\text{count } 1))
15
16 {-# ANN testData TestInput #-} testData = 0 \rightarrow (0 \rightarrow (0 \rightarrow \text{empty}))
17
```

4.4 MyHDL

The MyDHL implementations bear a strong resemblance to the VHDL implementations. While the syntax is different from VHDL, the mapping from MyDHL to VHDL is quite obvious when the MyDHL code and generated VHDL are put side by side. When it comes to abstraction, MyHDL has no significant advantage over VHDL. The syntax will be easier for programmers who are familiar with Python, but this comes at a cost: most of the time when MyHDL encounters an error while converting to VHDL it simply crashes and leaves a trace-back which leads to somewhere in its Python-to-VHDL conversion code, which offers no hint as to what went wrong, where it went wrong, or how it can be fixed.

One advantage of MyHDL is that it is relatively easy to test components without converting to VHDL. Because code written in MyHDL is also valid Python code, testing components only requires a Python interpreter, which is already present because MyHDL requires it. No complicated extra tools are required, and during testing all features of Python can be used.

A potential disadvantage would be Python’s duck-typing.[4] It is not immediately clear what kind of signals a component accepts by looking at its declaration. In this aspect MyDHL is quite different compared to VHDL, which is statically typed.

The FIR-filter as implemented in MyDHL is shown below. We were unable to get this code to compile; MyHDL produces the following unhelpful message: "AttributeError: 'NoneType' object has no attribute 'size'." This error occurs somewhere deep in the toVHDL function, and we were unable to discover its cause. We are forced to conclude that the error messages in the current version of MyDHL are inadequate and hope they will be improved in the future. Even if translation to VHDL were to succeed, the component below would only be able to handle integers: MyDHL does not appear to have support for floating point and handles all numbers with the intbv class, which
is treated as an integer.

**Listing 6. FIR-filter in MyHDL**

```python
1 def firfilt(sig_in, sig_out, coefs, clk):
2     buffer = Signal(intbv(0)[len(sig_in) * len(coefs)])
3     inlen = len(sig_in)
4     saved_coefs = coefs
5     colen = len(saved_coefs)
6     @always(clk.posedge)
7         def logic():
8             buffer.next = sig_in
9             buffer.next[0] = buffer[0] * (colen - 1):
10            data.next = data[0]
11            if (inlen == 1) and (dindex == maxindex):
12                empty.next = 1
13                full.next = 0
14            else:
15                empty.next = 0
16                full.next = 1
17            data.next = data[datalength*index]
18        return logic
19    @always(clk.negedge)
20        def dequeue():
21            sig_out = sig_in
22            dindex = dindex - 1
23            if (dindex == 1):
24                full.next = 0
25                empty.next = 1
26            else:
27                empty.next = 0
28                full.next = 1
29            sig_out.next = sig_in
30        return dequeue
31
32 toVHDL(firfilt, sin, sout, coefs, clk)
33 convert()
```

The FIFO-queue we created in MyHDL closely resembles the one we wrote in VHDL. The signals we have to define for a VHDL entity, and indeed that is where the definitions end up when the code below has been translated to VHDL. The complete description is 56 lines long, which is only 2 lines shorter than the FIFO-queue we have written in VHDL.

**Listing 7. FIFO-queue in MyHDL**

```python
1 def FIFO(data_in, data_out, enq_or_deq, active, empty, full, clk, slots=10, datalength=8):
2     maxindex = slots - 1
3     data = Signal(intbv(0)[datalength * slots:])
4     dindex = Signal(intbv(0, min=0, max=slots))
5     @always(clk.posedge)
6         def logic():
7             if (active == 1 and enq_or_deq == 1 and dindex < maxindex):
8                 enqueue
9                 dindex.next = dindex + 1
10                empty.next = 0
11            else:
12                full.next = 0
13            data_out.next = data[datalength*(dindex+1):datalength*dindex]
14            data.next[0] = data_in
15
16 toVHDL(FIFO, sin, sout, action, active, empt, full, clk,1)
17 convert()
```

### 4.5 Comparisons

#### 4.5.1 Conciseness

It is quite clear that the hardware descriptions in CλaSH and Kansas Lava have significantly fewer lines compared to the VHDL and the MyHDL descriptions. In the following table the number of words and lines for each source file are shown. The counting was performed with 'wc', a standard Unix program that counts the number of characters, words, and lines in a file. To prevent encouraging 1-character variable names we have ignored the number of characters and will focus exclusively on the number of words and lines.

Using this table we can observe that VHDL and MyHDL have a similar number of lines when we compare the FIFO-queues, however VHDL uses significantly more words. CλaSH manages to use very few lines but many operators. A significant contribution to CλaSH's word-
count appears to be the type signature of shiftInto, which uses 24 words. This matches our earlier observation that CλaSH functions have complex type signatures. It should be noted that the Kansas Lava implementation of the FIFO-queue has fewer features than the other FIFO-queues. If a version supporting the same features as the other languages was written it would very likely be longer than the current implementation, which would make CλaSH the most concise language of the four.

4.5.2 Comprehensibility
We found Kansas Lava to be the most difficult language to work with by far, to the point where we were unable to create a working version of the FIR-filter, and only a very simple version of a FIFO-queue. We have no doubt these components could be created in a few lines by someone who has more experience with the language, but we had no such experience and failed. Most of the difficulty in this language comes from Signal, Matrix, and their types. Having been unable to complete the components, we did not have a good chance to compare the resulting source code with the other languages. We are open to the possibility that well-written Kansas Lava code is very easy to read and understand, but all we know for sure is that it is very hard to write.

Our other functional language, CλaSH, was second hardest to write, but we consider it the easiest of all languages to understand. In the FIR-filter most of the behaviour is expressed by a single line, which uses the functions fold and zipWith. These basic functions are very well known within the functional programming community. In the FIFO-queue the behaviour is spread over three lines which use pattern matching. This approach is definitely more readable than the large conditional blocks that had to be used in MyHDL and VHDL.

That leaves MyHDL and VHDL. Between these two, it’s hard to pick a winner. We prefer VHDL because it is more explicit about types, and because we think the \((x \downarrow y)\) notation is easier to read than MyHDL \([x : y]\) slices. A person who has gotten used to Python’s syntax may well come to the opposite conclusion. In the end the differences between these two languages are so small that it is primarily a matter of personal preference.

4.5.3 Reusability
When comparing reusability of the FIR-filters it appears there are no real differences. It is possible in all of the four languages to create some generic component that acts as a FIR-filter for some type of numbers. We did not actually manage to do this with Kansas Lava, but are quite sure such a thing is possible in the language.

When comparing the FIFO-queue it seems CλaSH is the most reusable language. In every language it is possible to create a FIFO-queue that stores some number of bits, therefore every type that can be represented as some string of bits could be stored in a FIFO-queue in every language. In this case CλaSH distinguishes itself by being able to automatically convert some user-defined types into binary presentations, which leads us to conclude that it is the most convenient language with respect to reuse of functions and components.

5. CONCLUSION
There are several observations this research has allowed us to make, the first and most important of which being that functional languages are significantly more concise compared to more traditional languages such as VHDL and MyHDL. However, writing in these functional languages is considerably harder. For small projects using an easier language such as VHDL is probably the best idea, once you get to more complex projects the more difficult but also more powerful functional languages become a better choice.

Our second observation is that even among two functional languages there can be many differences. CλaSH and Kansas Lava have very different approaches for State, and this is noticeable during programming. And while the Kansas Lava approach of using Signals allows for multiple clocks, it also makes programming significantly more complex.

We were surprised to discover how much VHDL and MyHDL were alike. Especially in the FIFO-queue, the code looked almost exactly the same; large blocks of conditional statements. Perhaps MyHDL’s advantages only become apparent in larger projects, but in small projects such as the components in this research it has no real advantage over VHDL.

6. REFERENCES


