# HeDLa: A Strongly Typed, Component-Based Embedded Hardware Description Language

Gordon J. Pace

Department of Computer Science, University of Malta
gordon.pace@um.edu.mt

**Abstract.** Over the past years, various techniques for the embedding of hardware description languages within general purpose languages have been developed and explored. In particular, numerous HDLs embedded in strongly typed functional languages have been developed and used for different applications. A common trait of most of these languages is that they treat hardware components as functions or relations between the inputs and outputs of the circuit. The alternative view, of viewing the circuits as components which can be instantiated, composed and transformed has been a relatively less well explored area in this context. In this paper we present HeDLa, a component-based hardware description language embedded in Haskell, and show how features such as strong-typing and higher-order functions enable us to design and compose circuits in a safer and more abstract fashion. Furthermore, the component-based approach allows access to circuit structure directly, enabling us to reason about non-functional aspects of the component, such as placement, area and power consumption more easily. Finally, we discuss some initial experiments in multi-level simulation of circuits which enable testing and more effective simulation of large circuits.

## 1 Introduction

The utility of domain-specific languages has been widely accepted by the programming language design community and has increased in popularity over the past few decades. General purpose languages are excellent media to express algorithms that work in a variety of contexts, but the need for a context-dependent, domain-specific core language is extremely useful when designing solutions for a constrained problem-domain. One area which saw the rise of various (text-based) domain-specific languages in the seventies and eighties was that of hardware description and design. Languages such as Verilog [Ope93] and VHDL [LMS86] enabled modular design of hardware, allowing reuse of designed components, and abstraction in hardware design. Such languages recognised the need for different interpretations of hardware descriptions, allowing both structural descriptions (in which components are described in terms of their constituent parts, decomposing them into subcomponents until the gate or transistor level is reached) and behavioural descriptions (in which the behaviour of a component is given algorithmically). The former was necessary to allow actual circuit instantiation

based on the descriptions given in the language, while the latter enabled easier testing, more efficient simulation, and better structured design approaches. Due to the finite nature of hardware, algorithmic constructs, especially ones for loops were not allowed in structural descriptions, hence ensuring that every structural description is a finite one (that is, terminates) simply by ensuring that a structure does not (transitively) refer to a copy of itself — hence resulting in modular descriptions which are in the shape of a tree (or a directed acyclic graph if one considers reuse of modules). This restriction, however reasonable it may seem, did not allow algorithmic descriptions of regular shaped circuits in a concise manner through the use of loops, recursion and conditional selection based on values of static (compile-time) parameters. Certain such circuits could be described through the use of arrays of data shifted in smart ways to ensure that a repetative structure is produced. A classic textbook example would be that the array of carry-outs of the full-adders in a serial-carry adder can be shifted to the right and passed on as the carry-ins of the full-adders. However, more complex regular circuit structures, such as trees, and non-constant shifts of the arrays are simply not possible, or possible only in a convoluted unintelligible manner.

This restriction was addressed in various tools, providing non-standard extensions to VHDL and Verilog, allowing certain general-purpose constructs to be used in the circuit structural descriptions. Different tools provided different languages, some simply allowing regular repetition or structure-composition formats, others providing an essentially full-blown meta-language sitting above the structural descriptions. This two-tier meta-language approach worked well in the description of large regular circuits, and as long as the meta-program terminated, it produced a finite circuit that could be analysed, tested, verified and fabricated.

This two-tier approach is, however, useful in most other domain-specific languages, and was independently adopted in other areas. This approach of building a Turing-complete meta-language above the domain-specific language has various drawbacks. Domain-specific language design is clearly more challenging with this approach, since the language designer must also look into the design and implementation of the general-purpose meta-language. Furthermore, the end-users are expected to learn different syntax, different languages each with their different quirks, and in some cases different programming paradigms, for different domain-specific languages they are working in.

Embedded languages, a technique developed in the programming language community, was found to be an excellent approach to alleviate a number of these problems. In embedded languages, one builds a domain-specific library using a general-purpose language, called the host language, enabling a programmer to describe 'programs' in the domain-specific language to appear as though they were part of the host language. The programmer may then generate, analyse and manipulate programs in the domain-specific programs as though they were part of the host language itself. The host language thus automatically becomes a meta-language for the embedded language. Clearly, the more flexible and high-level the host language and its syntax are, the more difficult it becomes to

distinguish where the domain-specific program ends and where the rest of the code starts. From the language designer's point of view, the main advantage of designing a domain-specific language and embedding it within a host language is that he or she needs not reinvent the wheel and create a new general-purpose language, while from the end-user's perspective, the main advantage is that the meta-language is a standard language with which he or she may already be familiar and knowledge of which goes beyond the use of the domain-specific language. Some argue that embedded languages are nothing more than domain-specific libraries, and certainly, the dividing line between a domain-specific library, and a domain-specific language is blurred. However, one may look at embedded languages as a structuring principle for domain-specific libraries, to provide structures allowing first-order programs in the domain-specific language to be written without resorting to the host language.

Over the past years, various embedded hardware description languages have been designed and used (see, for instance Lava [CSS03], Hydra [O'D96], Hawk [DLC99], *reFLect* [MO06], Wired [ACS05] and Dual-Eval [WAHR04]). In particular, functional languages have proved to be particularly suited for this purpose. A common trait of most of these languages is that they treat hardware components as functions or relations between the inputs and outputs of the circuit. The alternative view, of viewing the circuits as components which can be instantiated, composed and transformed has been a relatively less well explored area in this context [WAHR04,ACS05]. Circuits are produced as a result of function calls in the host language, and a common challenge in most embedded hardware description languages is that of having access to the structure induced from the description of the circuit in the host-language. For example, a circuit induced through a linear chain of recursive calls could be viewed as a nested chain of similar components, which information can be used to aid reasoning about the circuit or for inducing hints for placement tools. However, without additional machinery, this information cannot be created and accessed directly. Being able to refer to such blocks, would also enable a hardware designer to reason about non-functional aspects such as power consumption and wire lengths of such compound components, and use other standard development techniques such as refinement in the design process, all within the same language.

In this paper we present initial experiments in building a component-based hardware description language embedded in Haskell [Jon03], and show how features such as strong-typing and higher-order functions enable us to design and compose circuits in a safer and more abstract fashion. Furthermore, we discuss the use of refinement and multi-level description and simulation of circuits which enable testing and more effective simulation of large circuits. Essentially, this enriches the approach the embedded hardware description language with a design scripting language. Finally, we discuss how the component-based approach allows more direct access to circuit structure, enabling us to reason about non-functional aspects of the component, such as placement, area and power consumption more easily.

## 2 HeDLa: Embedding Yet Another Structural HDL in Haskell

### 2.1 Breaking Away from Circuits as Functions

When embedding a language, one would want the embedded programs to look similar in style to the rest of the code. It is thus natural to emulate the host language paradigm in the embedded language. Most probably for this reason, one finds that in most hardware description languages embedded in Haskell, circuits are described as functions from inputs to outputs. Circuit reuse simply becomes function application, as for example can be seen in the following example in Lava [CSS03]:

```
halfAdder (a, b) = (s, c)
  where
    c = and2(a, b)
    s = xor2(a, b)

fullAdder (cin, (a,b)) = (s, cout)
  where
    (c1, s1) = halfAdder(a, b)
    (c2, s)  = halfAdder(s1, cin)
    cout     = or2(c1, c2)
```

In HeDLa, we take a different approach, in which circuits are objects which can be tagged, manipulated and structurally modified. Structurally describing a a half-adder in terms of basic gates, and a full-adder in terms of half-adders is done using the following code:

```
halfAdder = Circuit
  { name        = "halfAdder"
  , inputs      = ("a", "b")
  , outputs     = ("s", "c")
  , description = use xor2 ("a", "b") "s"
                & use and2 ("a", "b") "c"
  }

fullAdder = Circuit
  { name        = "fullAdder"
  , inputs      = ("cin", ("a", "b"))
  , outputs     = ("s", "cout")
  , description = use halfAdder ("a", "b") ("s1", "c1")
                & use halfAdder ("s1", "cin") ("s", "c2")
                & use or2 ("c1", "c2") "c"
  }
```

Each circuit is defined as a Haskell record, with not only a Haskell function name, but also a string as a name. The wires are not viewed as Haskell parameters to the actual circuit objects, but are labelled and cross-referenced

using strings. The inputs and outputs themselves of a circuit are expressed as structures (tuples and lists) of variable names, and are referred to string names. Finally, the description of the circuit behaviour is given (in this case) as a structural description — a combination of instances of basic components and compound components defined elsewhere in the code. The `use` keyword enables the creation of an instance of a circuit component (with the input and output wires given as additional parameters), and these can be combined together using the `&` operator.

From this example, it is clear that there is syntactic overhead in using HeDLa to describe the structure of a circuit as opposed to Lava functional-style descriptions. The primary syntactic distractions are the two names given to each component (the name of the Haskell object and the string used in the record) and the clunkiness of quoted wire names. Despite the additional syntax, the use of both the component and the wire names is particularly useful when the descriptions are exported to VHDL or other external formats, since they give a way of relating the simulation and verification with the original HeDLa descriptions. In fact, the component name is only used for this purpose.

### 2.2 Strongly-Typed Circuits

In HeDLa, inputs and outputs of a circuit are not directly equivalent to inputs and outputs in the Haskell sense. Instances of a circuit, take both the inputs and outputs of the circuit component being instantiated as inputs in the functional sense. Although it creates a dichotomy between the Haskell code and the embedded language code, the resulting structure is closer to VHDL and Verilog descriptions of circuit instances. To enable structuring the input as tuples and lists, Haskell type classes are used to allow different circuits to take different types of structures as inputs and outputs — in the above example, the basic gates, the half-adder and the full-adder all have different structures of inputs and outputs.

Furthermore, Haskell types are used to ensure that circuit instantiation is done in a type-safe and correct manner. The types of the circuits given and the basic gates are:

```
xor2, and2, or2 :: (Bit, Bit) |=> Bit
halfAdder :: (Bit, Bit) |=> (Bit, Bit)
fullAdder :: (Bit, (Bit, Bit)) |=> (Bit, Bit)
```

The infix parametrised type `|=>` is used to describe circuits taking inputs with the structure appearing as the first type parameter, returning a structure given as the second type parameter. This enables us to use Haskell type checking to ensure that all circuits are instantiated in type-safe manner.

Currently HeDLa supports only wires carrying boolean streams, but it is planned to be extended to support other streams carrying integers and other

data types. Streams can be structured not only in tuples, as we have seen in the examples so far, but also using lists as shown in the example below[1]:

```
nBitAdder :: Int -> (Bit, [Bit], [Bit]) |=> [Bit]
nBitAdder n = Circuit
  { name = "nBitAdder"
  , inputs = ("c"!0, bus "a" n, bus "b" n)
  , outputs = bus "s" n ++ ["c"!n]
  , description
      = combine
           [ use fullAdder ("c"!m, ("a"!m, "b"!m)) ("c"!(m+1), "s"!m)
           | m <- [0..n-1]
           ]
  }
  where
    bus v i = map (v!) [0..i-1]
    v!i = v ++ show i
```

### 2.3 Circuit Interpretations

Starting with these descriptions, one can interpret them in different ways. The most straight-forward interpretation is that of simulation, where the given circuit is (recursively) interpreted in terms of its underlying components, until the basic gates are encountered, and interpreted using a default interpretation in Haskell:

```
> simulate fullAdder (high, (low, high))
(low, high)
```

Based on the description, one can also produce VHDL output of a circuit. However, unlike embedded HDLs such as Lava and Hawk, we produce a modular description following the structure of the circuit as defined, rather than one flattened netlist with no structure. Apart from outputting to VHDL, HeDLa also allows exporting a circuit to model checkers[2] to verify safety properties of circuits.

## 3 Behavioural Descriptions, Specifications and Refinement

### 3.1 Behavioural Descriptions and Simulation

The underlying gate components in HeDLa are used no differently than the compound components the user may define. The real difference is the interpretation of the components during simulation — whereas compound circuits are decomposed into their subcomponents and simulated separately, the basic gates

[1] The function `combine` takes a list of circuit instances and combines them together using `&`

[2] Currently we support only SMV as a model-checker.

have a behavioural semantics associated to them, which enables direct simulation. HeDLa allows the description of new components with a behavioural, as opposed to structural description. In languages such as VHDL, this necessitated the definition of a sub-language for the description of behavioural code. In embedding HeDLa in Haskell, we reuse the host language to describe behaviour of circuits. For instance, the behaviour of a two-input xor gate can be defined in the following manner:

```
xor2 = Circuit
  { name        = "xor2"
  , inputs      = ("a", "b")
  , outputs     = "z"
  , description = behaviour (\(a,b) -> a /= b)
  }
```

The simulation of circuits simply uses the behavioural description to interpret the gate when it is encountered in a circuit. Behavioural descriptions of higher level circuits are frequently used in standard HDLs to test the structural description, by simulating the two side-by-side for different inputs. Rather than keeping different descriptions for the different functionalities of a circuit separately, we enable dual descriptions of circuits, with can be simulated in either structural or behavioural modes:

```
fullAdder = Circuit
  { name = "fullAdder"
  , inputs = ("cin", ("a", "b"))
  , outputs = ("cout", "s")
  , description
      = use halfAdder ("a", "b") ("c1", "s1")
      & use halfAdder ("s1", "cin") ("c2", "s")
      & use or2 ("c1", "c2") "cout"
  } `setBehaviour` (\(cin, (a, b)) ->
     let (cin', (a', b')) = (bit2int cin, (bit2int a, bit2int b))
     in  (cin' + a' + b' >= 2, isOdd (cin' + a' + b'))
  )
```

By considering the behavioural description to be the specification, and the structural to be the implementation, we enable testing of circuits through the use of QuickCheck [CH00]. Furthermore, in larger circuits, the designer can switch between modes of the constituent subcircuits to enable more efficient simulation through the use of the specification, rather than the structural description of large, but trusted subcomponents.

### 3.2 Observer-Based Testing and Verification

One strength of behavioural descriptions is that they can not only be used as specifications against which to run tests, but also used to simulate the actual circuit directly (as opposed through the interpretation of its subcomponents).

This means that the specification has to be a deterministic one, to enable the calculation of the correct outputs based on the inputs. Furthermore, sometimes it is easier to write a specification checking that the inputs and outputs are correct. HeDLa supports the use of observers, which given the input and output of a circuit, return a single boolean value stating whether the circuit is working correctly. Both structural and behavioural observers can be defined, with behavioural observers used in testing, while structural observers can be used both in testing and when producing output to model-checkers.

The following example shows a behavioural observer for a full-adder, asserting that the interpretation of the output as a two-bit number gives the same value as the addition of the three input bits:

```
fullAdder = Circuit
  { name = "fullAdder"
  , inputs = ("cin", ("a", "b"))
  , outputs = ("cout", "s")
  , description
      = use halfAdder ("a", "b") ("c1", "s1")
      & use halfAdder ("s1", "cin") ("c2", "s")
      & use or2 ("c1", "c2") "cout"
  } `setBehaviouralObserver` (\((cin, (a, b)), (cout, sum)) ->
     let (cin', (a', b')) = (bit2int cin, (bit2int a, bit2int b))
         (cout', sum') = (bit2int cout, bit2int sum)
     in  cin' + a' + b' == 2 * cout' + sum'
  )
```

### 3.3 Multi-Level Refinement and Data Refinement

Frequently, behavioural descriptions and observers use different data representations than the structural description, requiring translation to and from the different representations. For instance, in the case of an $n$-bit adder, the concrete inputs (one carry-in bit, and two bitstrings of length n) can be translated into a more abstract interpretation — as a list of three numbers. The outputs can be translated back from a number into $n+1$ bits. Once this data refinement is specified, the behaviour is defined to work on the abstract interpretation of the inputs, and producing abstract outputs. In this case, the specification simply becomes the Haskell function `sum`:

```
nBitAdder n = Circuit
   { ...
   } `usingDataRefinement`
   ( \(c, (as, bs)) -> [bit2int c, bits2int as, bits2int bs]
   , int2bits (n+1)
   ) `setBehaviour` sum
```

Rather than constrain descriptions to just a specification and an implementation, we are currently experimenting with extending HeDLa with multi-level refinement, to enable stepwise refinement of a component for design, testing and

verification. In this context, the use of explicit data-refinement specification is much more useful, since it allows the designer to go up and down the refinement layers in multiple steps.

### 3.4 Non-Functional Circuit Properties

The major advantage of using a structure, rather than a functional view of circuits, is that we maintain a hierarchical view of the circuits and their components. Furthermore, one can add circuit information as the circuit is constructed. We are currently looking into the use of these features for the description of non-functional properties of circuits. One area of application is adding placement information, or hints through the use of combinators for combining circuits. Consider the operator `->-`, which connects two circuits next to each other as shown in figure 1. An implementation, of such an operator can be written as follows[3]:

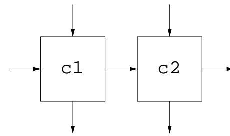


**Fig. 1.** Placing two circuits next to each other: `c1 ->- c2`

```
c1 ->- c2 =
  let (left1,  up1)   = inputs c1
      (left2,  up2)   = inputs c2
      (right1, down1) = outputs c1
      (right2, down2) = outputs c2
   in Circuit
      { name        = name c1 ++ "->-" ++ name c2
      , inputs      = (left1, (up1, up2))
      , outputs     = (right2, (down1, down2))
      , description = use c1 (inputs c1) (outputs c1)
                    & use wire right1 left1
                    & use c2 (inputs c2) (outputs c2)
      }
```

As it stands, the only information we maintain is that of how wires are connected. However, one can easily tag the component `c1` to lie to the right of `c2`. In this manner, we can actually produce concrete placement and wire length information from such a description.

[3] This implementation allows for any pair of structure of wires as input and as output, but assumes that there are no name clashes between wire names in the subcircuits. Note that the `wire` circuit simply connects the input to the output.

## 4 Future Work and Conclusions

In this paper, we have presented the basic functionality of an experimental structural hardware description language embedded in Haskell. Various HDLs embedded in Haskell and other functional languages have been developed and used over the past years. Languages such as Lava [CSS03], Hawk [DLC99] and Hydra [O'D96] have followed a strictly functional view of circuits — circuits appear as functions in the host language, and their inputs and outputs are identical to the inputs and outputs of the functions. The descriptions tend to be cleaner in these languages, but lose information about the structure due to the functional view of the circuits. The challenge to incorporate information about the structure of the circuit description has spawned a number of other embedded languages. Wired [ACS05] uses a component-based approach to circuits, with combinators used to describe circuits, keeping structural and layout information. As opposed to our approach, the combinators lie at the core of the language making it much better at describing information about layout, though less oriented towards actual circuit behavioural description. Just like Wired, Dual-Eval [WAHR04] is also similar to HeDLa in that it takes a component-based approach to circuit description, with explicit wire names and connections. *reFL*$^{ect}$ uses a functional meta-language to embed a HDL, with the use of language reflection features used to have access to the circuit generators themselves. HeDLa loses some of this information — for instance, when function calls are used to generate families of circuit instances (in the `description` field). On the other hand, we have access to the nested structure of circuits defined explicitly.

In HeDLa, we have tried to find a balance between using a component description of circuits, but still keeping as close as possible to the functional view. There are still various issues we are working on resolving in HeDLa. On one hand, we are looking into extending the functionality of the language, by introducing multi-level refinement, and looking into techniques to enable the user to add general non-functional features. In particular, we plan to look at the use of HeDLa to reason about placement, wire length and area analysis. In this paper, we have only used small examples using combinational circuits. Although HeDLa can also handle sequential circuits, the interaction between the behavioural descriptions and observers with sequential circuits still needs to be refined, to enable us to test run the language on a number of large case studies thus assessing how effective it is in the design of large hardware systems.

## References


[ACS05] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*. Springer-Verlag, October 2005.

[CH00] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming 2000*. ACM SIGPLAN, 2000.

[CSS03] Koen Claessen, Mary Sheeran, and Satnam Singh. Using Lava to design and verify recursive and periodic sorters. *International Journal on Software Tools for Technology Transfer*, 4(3):349–358, 2003.

[DLC99] Nancy A. Day, Jeffrey R. Lewis, and Byron Cook. Symbolic simulation of microprocessor models using type classes in Haskell. In *CHARME'99 Poster Session*, 1999.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[LMS86] Roger Lipsett, Erich Marchner, and Moe Shahdad. VHDL — the language. *IEEE Design and Test*, 3(2):28–41, April 1986.

[MO06] Tom Melham and John O'Leary. A functional HDL for ReFLect. In *Designing Correct Circuits*, 2006.

[O'D96] John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languages in Education*, volume 1125 of *Lecture Notes In Computer Science*, pages 221–234. Springer Verlag, 1996.

[Ope93] Open Verilog International. *Verilog Hardware Description Language Reference Manual (Version 2.0)*. Open Verilog, 1993.

[WAHR04] Jr. Warren A. Hunt and Erik Reeber. A hierarchical modeling system. In *Designing Correct Circuits*, 2004.