TECHNICAL REPORT

Multi-Stage Languages in Hardware Design

Gordon J. Pace
Christian Tabone

Department of Computer Science & A.I.
University of Malta
Msida MSD 06
MALTA

Tel: +356-2340 2519
Fax: +356-2132 0539
reports@cs.um.edu.mt
http://www.cs.um.edu.mt/~reports

Multi-Stage Languages in Hardware Design

Gordon J. Pace
University of Malta, Malta.
`gordon.pace@um.edu.mt`

Christian Tabone
University of Malta, Malta
`christian.tabone@um.edu.mt`

Abstract: *As circuits increase in size and complexity, hardware description techniques have been trying to adopt features already well-established in software languages. In this paper, we investigate how different hardware description languages implement levels of abstraction over the hardware designs, and we examine how improvements have lead to features like parameterised circuits and generic descriptions, that enable users to efficiently model and reason about large regular-shaped structures and connection patterns. Nonetheless, the ability to include non-functional properties of circuits in the same description is still an open issue. Lately, proposed solutions are looking into meta-functional languages and multi-staging techniques. We examine how hardware description languages can benefit from the capabilities of meta-functional languages, which are able to reason about, and transform the circuit generators as data objects, thus providing a means to access both the functional and non-functional aspects of the generated circuits.*

Multi-Stage Languages in Hardware Design

Gordon J. Pace
University of Malta, Malta.
gordon.pace@um.edu.mt

Christian Tabone
University of Malta, Malta
christian.tabone@um.edu.mt

Abstract: *As circuits increase in size and complexity, hardware description techniques have been trying to adopt features already well-established in software languages. In this paper, we investigate how different hardware description languages implement levels of abstraction over the hardware designs, and we examine how improvements have lead to features like parameterised circuits and generic descriptions, that enable users to efficiently model and reason about large regular-shaped structures and connection patterns. Nonetheless, the ability to include non-functional properties of circuits in the same description is still an open issue. Lately, proposed solutions are looking into meta-functional languages and multi-staging techniques. We examine how hardware description languages can benefit from the capabilities of meta-functional languages, which are able to reason about, and transform the circuit generators as data objects, thus providing a means to access both the functional and non-functional aspects of the generated circuits.*

1   Introduction

Over the past years, the number of transistors on a single chip has been growing dramatically, influencing the increase in complexity of integrated circuits. This has lead hardware engineers to turn to high-level abstractions in order to describe and design digital hardware, and handle the vast detailed information of modern micro-electronic circuits. High-level abstractions enable hardware descriptions to be defined in a modular fashion, making these more accessible and manageable. Innovative designs can be tested, debugged and verified at various abstract levels, prior to the actual manufacturing of the hardware. This approach highly facilitates the difficult and tedious task of designing circuits manually through the use of block diagrams or schematic diagrams. Abstraction techniques have been used in a number of areas to handle complex systems. An abstract model of a system is regarded as a simplified

version of the same system showing only the most important components and hiding the irrelevant details. In hardware design, abstraction techniques avoid having to reason about the low-level implementation details, whilst allowing hardware designers to concentrate on alternative high-level designs. This does not only aid the work of hardware engineers, but it also reduces the development time of a system and decreases the overall expenditure costs.

Before considering using high-level abstraction techniques, hardware systems used to be designed by means of block diagrams. A block diagram gives a visual sketch or picture of the system, indicating what components are used, how these are place within a specific area, and how all the connections are laid out. Such diagrams provide all the necessary details for a system, however these are not very modular, and tend to be difficult and tedious to use, especially for larger complex circuits. Nowadays, hardware systems are normally represented by textual descriptions, offering more modularity than block diagrams. Textual descriptions are more manageable and maintainable, and higher-level abstractions can be applied more easily.

Textual descriptions can describe a digital system in different ways or perspectives. In standard hardware description languages such as VHDL or Verilog [LMS86, Ope93], two different description types are usually possible — *behavioural* and *structural*. A behavioural description gives the functional properties of the system, and it considers the whole circuit as a black box, focusing on the relationship between inputs and outputs, and ignoring implementation detail. On the other hand, a structural description specifies precisely what components are used and how these are connected to each other to compose the internal implementation of the circuit. Although the behavioural way of describing hardware is more abstract than the structural way, it does not provide access to the actual implementation details, limiting its use to hardware specification rather than the actual circuit design. In this report we mostly deal with the structural perspective of hardware, and focus on the different software abstractions that can be applied to structural descriptions.

Modular abstractions enable hardware designers to define the most commonly used circuits as reusable modules that could later be instantiated whenever required. Furthermore, it is desirable to have circuit models that can be reviewed, simulated, tested and modified when required. Circuit representations should provide the possibility to reason about the hardware that is being described, and apply appropriate manipulations or transformations, based on the outcome of such reasoning. By means of the software abstractions that are provided to hardware models, these operations are more manageable at higher levels of abstraction. Complex hardware designs can be handled more efficiently, and more innovative designs, development issues and bugs, can be explored without the need to manufacture the real circuit. After simulation, testing and verification, the hardware model can be translated into a low-level repre-

sentation (hardware synthesis). At this stage only the final minor changes are applied, avoiding to work with the unnecessary details of the low abstract levels throughout the development process.

Researchers both in industry and academia have been struggling to develop better hardware description languages (HDLs) and other mechanisms to enable such operations beyond the traditional languages like VHDL or Verilog. Functional languages in particular, have shown to provide an excellent means to develop experimental HDLs [She05], capable to describe how the functional operations of circuits are structured together. Reasoning about the circuits and manipulation is also possible at higher levels of abstraction. Model checking and hardware verification are more recent research areas, where the circuit representations are formally clarified for their correctness. The functional paradigm offers numerous advantages by allowing designers to describe circuit generators rather than the actual circuits, thus providing access to the circuit models. However, one area that remains questioned is the inclusion of non-functional aspects of the hardware, such as floor-planning, power consumption and speed. By using software abstractions to hide away unnecessary information, the resulting circuit descriptions focused mostly on the functional aspects of the circuits, thus ignoring the non-functional aspects. As more improvements are made the handling of non-functional properties of the circuits is becoming more of an unresolved issue. Some techniques regarding this area have been proposed, in particular the use of meta-languages that provides access to the circuit generators themselves and not just the circuit representations. Manipulating and reasoning about the circuit generators should provide an opportunity to induce non-functional properties about the circuits prior or during the generation of the circuits.

In this paper we illustrate the different techniques that have been used to describe hardware, and how different HDLs have developed over the past years, materialising into today's modern research languages. In section 2 we give an overview of the traditional HDLs that are used by the industry. The concept of parameterised descriptions and regular-shaped circuits is introduced in this section, and followed up in section 3, where we illustrate experimental HDLs that focus on this issue. We also start considering how certain non-functional properties like placement can be addressed. In section 4 we outline the more modern technique of embedding that is used to develop *domain-specific languages*. We focus on HDLs that are embedded within the pure functional language Haskell, and discuss the advantages of parameterised functions and higher-order functions. The more modern approach of meta-programming is outlined in section 5, and we illustrate how hardware design can benefit from the features present in meta-languages.

2 VHDL and Verilog

VHDL [LMS86] was standardised by IEEE in 1987, offering a means to describe, model and simulate application-specific integrated circuits. Verilog HDL [Ope93] has very similar functionalities and capabilities to VHDL, and was standardised by IEEE in 1995. VHDL and Verilog have dominated the hardware development market for a considerable number of years, and are still widely in use today. When introduced, VHDL and Verilog addressed a number of problems, and managed to provide hardware designers a means to develop electronic components more rapidly for a wide range of systems. These HDLs provided the necessary software abstractions that improved the capabilities of hardware engineers, and managed to control the exploding development costs of large complex systems, by enabling the industry to cope with the rapidly increasing demand of the market. Although the syntax of the languages is different, the capabilities of VHDL and Verilog are closely related. The most significant similarity we consider lies in the conceptual levels and the available constructs that can be used to describe circuits. The examples in this section are given in VHDL, yet we do not distinguish between the two languages.

Both VHDL and Verilog provide high-level abstractions above the physical implementation details of the circuit. Different levels of abstraction are possible, including the gate-level, the register-transfer-level and the behavioural-level.

The gate-level lies just above the physical details of the hardware. It is a detailed netlist describing the physical connectivity of low-level components, as a network of logic gates and registers. These descriptions depict the structural specification of the circuit using component details, which are usually instanced from a technology library containing specific details such as propagation delay information. This abstract level is closely related to boolean expressions describing the relation between the inputs and the outputs. Designing large complex circuits at this low-level of abstraction can be impractical.

The register-transfer-level (RTL) is a higher conceptual level for structural circuit descriptions. The building blocks at this abstract level are modules constructed from simple logic gates. These modules are considered to be more like functional units. These modules normally include the basic logic functions, adders, multiplexers and storage components. The main feature that distinguishes the register-transfer-level from the gate-level is the use of a common clock that is explicitly included within the logical functions that describe the hardware. This level of abstraction also provides the capability to abstract signal representations, by grouping together wires and interpreting them more as data rather than just signals. Listing 1 illustrates an RTL description of a half-adder component. This textual description clearly defines the

Listing 1: A structural description of a half-adder

```
entity HALFADDER is
    port( A, B : in bit;
          SUM, CARRY : out bit );
end HALFADDER;

architecture RTL of HALFADDER is
begin
    SUM <= A xor B;
    CARRY <= A and B;
end RTL;
```

input wires and the output wires of the component, and how these are related to each other by means of other modules, in this case the basic logic functions **xor** and **and**.

The highest level of abstraction is the behavioural-level, where a hardware system is described in terms of an algorithm, including computational steps, processes and communication. As we have already mentioned, behavioural descriptions are beyond the scope of this paper and we shall not be discussing them any further.

Modular descriptions in VHDL are defined as components which can later be reused to describe larger systems. These modules could be instantiated within other descriptions, thus providing a means to define larger circuits more rapidly. Apart from allowing a form of reusability, the modular descriptions enable hardware designers to describe the hierarchical architecture of a hardware system. This is particularly useful to observe the architecture of the designed circuit and clearly visualise how large circuit blocks are constructed from smaller ones. The module framework is in particular useful to describe regular-shaped circuits, such as an n-bit ripple-carry adder, which is constructed as a chain of full-adders. Listing 2 illustrates the definition of a four-bit-adder given in terms of the full-adder component. The component section of the code acts as a declaration for the full-adder and its interface. The declared component can be either an existing part defined earlier, or a hypothetical part which still needs to be defined or instanced from a technology library. Each of the required full-adder components are instanced separately, and the **port map** construct is used to declare how the interfaces and the wires are connected. Notice how the full-adder component is regarded as a black box, and the internal structural details are not included as part of the adder description. This simplifies the manner in which digital hardware is described. If one had to ignore the concept of components, the equivalent adder circuit would have been described in terms of primitive gates making the task

Listing 2: A four-bit adder defined in terms of full-adder components

```vhdl
architecture STRUCTURE of 4bit_adder is

component full_adder
   port( a, b, cin : in STD_LOGIC;
         sum, cout : out STD_LOGIC );
end component;

signal c1, c2, c3 : STD_LOGIC;

begin
   b0_adder : full_adder
       port map (a => a(0), b => b(0), cin => '0', sum => sum(0), cout => c1);
   b1_adder : full_adder
       port map (a => a(1), b => b(1), cin => c1, sum => sum(1), cout => c2);
   b2_adder : full_adder
       port map (a => a(2), b => b(2), cin => c2, sum => sum(2), cout => c3);
   b3_adder : full_adder
       port map (a => a(3), b => b(3), cin => c3, sum => sum(3), cout => cout);
end STRUCTURE;
```

more difficult. The concept of components or blocks adds a layer of abstraction to the hardware descriptions, making these much more manageable and comprehendible.

Subsequently, consider having to describe an n-bit adder for a larger number of bits, such as 32-bits or 128-bits. The initial version of VHDL (*VHDL'87*) did not provide well suited constructs to describe regular-shaped structures in terms of a natural repetitive style. In the case of an n-bit ripple-carry adder, the designer would have to specify all of the required full-adders and their connections manually, similar to the description given for the four-bit adder. Using this method to define large circuits is a tedious and error-prone process. In *VHDL'93*, generate constructs were introduced to support modular descriptions of such circuits. By means of the **for-generate** statement, an n-bit ripple-carry adder can be described iteratively in a loop-like fashion, during which full-adder components are instanced. An example of a ripple-carry-adder description for an arbitrary bit size is given in Listing 3. The generate construct introduced a two-level style of programming within VHDL, where the first level is describing the actual hardware in terms of the more traditional components, while at a higher level regular-shaped structures are defined iteratively by means of the generate constructs.

Listing 3: Generate constructs used to define an n-bit ripple-carry adder

```
word_adder : for i in 0 to adder_size − 1 generate
  lsb : if i = 0 generate
    lsb_adder : half_adder
      port map (a => addend(0), b => augend(0), s => sum(0), c => carry(0));
  end generate lsb;

  other_bits : if i /= 0 generate
    other_adder : full_adder
      port map (a => addend(i), b => augend(i), c_in => carry(i−1), s => sum(i),
          c_out => carry(i));
  end generate other_bits;
end generate word_adder;
```

One problem with VHDL and Verilog is the syntactic overhead present in the descriptions. This might obstruct hardware designers when defining circuits, and as the complexity of micro-electronic circuits increases, a more abstract way for describing hardware is required. The generate construct helps designers to define regular-shaped circuits in an iterative fashion. Nevertheless, more elaborate structures like tree-shaped circuits, butterfly circuits and others can still be awkward to define by means of the generate construct. Higher abstract levels, are capable to provide parameterised hardware descriptions, thus obtaining more flexible and compact descriptions especially for large regular-shaped circuits.

3   Languages for Parametrised Hardware Design

One option that has been explored is that of having a two-level language approach, similar to what the generate construct provides for VHDL. Having a two-level language gives the possibility of having parameterised hardware descriptions, meaning that circuit structures can be described as generic components for an arbitrary size that can be instantiated by parameterised values. To investigate this approach further we look at two distinct languages; Pollux which is actually the interpretation of the Lustre programming language as circuits, and Pebble which is a completely new language.

3.1 Pollux: A Lustre Environment for Circuits

Pollux [RH91] is a design environment within the programming language Lustre [CPHP87], intended to target the design of high-level hardware. Pollux can be regarded as an extension to the synchronous language Lustre, providing a suite of tools to generate synchronous circuit representations and the corresponding simulation programs. The Lustre syntax is not complicated and hardware specifications can be elegantly described, whilst providing a clear hierarchical arrangement of the definitions of sub-components. The initial idea behind Lustre was that of describing real time applications, therefore these characteristics can be easily applied for hardware design.

A Lustre program is constructed as a network of other Lustre sub-programs. This network is controlled by a global synchronous clock, thus each of the operators execute concurrently and should conceptually take up no time. Another important feature of Lustre is that nodes operate over time on streams of values, hence, even sequential circuits with delay components can be modelled. Small synchronous circuits can be represented and simulated using just basic Lustre operators, however for larger circuits a number of extensions are desirable. Pollux provides such extensions, and are mainly intended to be used for regular circuit descriptions constructed from iterative or recursive structures. Pollux handles the description of such structures by means of parameterised nodes. These parameterised nodes are to be distinguished from parameterised connection descriptions, since these nodes are unable to handle electronic components as arguments to compose generic structures, but rather this parameterised argument (usually of numeric value) is used to set the size of a regular-shaped circuit. Other abstract mechanisms are also available, aimed to handle structured types, recursive types, and even component placement.

The Pollux design environment creates a conceptual framework for a two-level language approach, where the Pollux constructs act as simple meta-constructs over the Lustre sub-programs. This is closely related to how the generate construct works in VHDL. Pollux provides a higher level of abstraction for the interpretation of circuit descriptions. This is done by defining a general node representing a hardware component for an arbitrary width of bits. Subsequently, when this component is required for a larger system, this is instantiated to a fixed size by a second node using a constant value. For instance Listing 4 gives the Pollux definition of a generic n-bit adder. The n-bit adder node is defined to iterate the one-bit adder node (full-adder) over an arbitrary number of bits. In order for this parameterised node to be used correctly it has to be instantiated by a constant n, which can be evaluated at compile time. A second node, given in Listing 5, is required to flatten out the structural description

Listing 4: A Pollux parametrised node defining an n-bit adder

```
node Add (const n:int; A,B:bool^n; ci:bool) returns (S:bool^n; c0:bool);
var C:bool^(n+1);
let
  C[0] = ci;
  (S, C[1..n]) = Add1(A, B, C[0..n−1]);
  co = C[n];
tel;
```

Listing 5: A Pollux node used to instantiate a 32-bit adder

```
node AddN32 (A,B:bool^32; ci:bool) returns (S:bool^32; c0:bool);
let
  (S, co) = Add(32, A, B, ci);
tel;
```

of the n-bit adder to a finite number of bits, in this case 32 bits.

Pollux extends the use of parameterised nodes, and manages to provide a means to define recursive structures, which is impossible to achieve when using only basic Lustre programming features. This is achieved by making use of parameterised nodes together with the static conditional operator "with-then-else". In Lustre, given the expression "$with\ b\ then\ e_1\ else\ e_2$", the boolean expression b is evaluated at compile time and if this is satisfied, the whole expression is replaced by e_1, otherwise it is replaced by e_2. By making use of this construct it is possible to generate networks that are shaped with recursive structures.

Pollux provides a two-level programming style. The conditional compilation operator "with-then-else" acts as the top-level language in constructing iterative or recursive structures, over the lower-level language which describes the individual components. Iterative and recursive structures are extremely common in circuit design, therefore higher-level abstractions such as a two-level language and parameterised descriptions help in providing more flexibility over modular hardware descriptions. Nonetheless, Pollux lacks the ability reason about the descriptions themselves in order to describe generic connection patterns for components. Connection patterns are defined at even higher-levels of abstraction, where the circuit descriptions themselves are be handled as parameters. In Pollux the two-level language approach, enables only a second language to reason about the first language, whereas at higher-levels of abstraction several additional levels can be conceptually introduced by simply using the same

language over the descriptions. This would enable the circuit descriptions to be more accessible, and perform other operations such as the generation of a netlist, circuit modifications and optimisations.

3.2   Pebble

Unlike Pollux, where the circuit descriptions are Lustre programs interpreted as circuits, Pebble [LM98] is a new HDL. Pebble is described as a *parameterised block language*, and is intended to construct hardware systems as a hierarchical structure of parameterised blocks. These blocks are highly modular and reusable, and can be easily parametrised to allow a customisable design size. Pebble is closely related to VHDL, in fact it is regarded as a simplified variant to VHDL.

A Pebble program is considered to be a block, which could either act as an autonomous system, or else it can be used to build more complex block-structured systems. It is worth mentioning that the Pebble architecture can be used to model any block-structured system and not just hardware and circuits. The primitive building blocks of a system are defined as empty blocks, without any implementation details, hence when a Pebble program in compiled or translated into another format, these primitive blocks are mapped to the corresponding components in the targeted format, which are instantiated from some kind of library or repository. A similarity between Pebble blocks and VHDL components can be perceived. The functionality of Pebble is to describe the structural layout of these basic components and how these are connected together to build larger systems.

The Pebble approach is similar to Pollux, since it provides a two-level language approach for the description of circuits. Hardware descriptions in Pebble can be specified by means of parameterised blocks. Compile time constructs similar to the generate statement in VHDL and Verilog are available in the Pebble language. These provide the possibility to reason about the hardware descriptions at compile time. The GENERATE-IF statement allows conditional compilation, where the boolean condition can contain parameterised values, whilst the GENERATE-FOR statement is used to describe regular-shaped circuits which can also make use of a parameterised value. The generate constructs of Pebble are closely related to those present in VHDL. Listing 6 illustrates the use of these constructs to define of an n-bit adder. Other networked structures such as tree-shaped circuits, butterfly-shaped circuits and other regular-shaped circuits, can be easily described by making use of these constructs.

An interesting functionality of Pebble is how the hardware descriptions can be annotated with additional information, such as placement details. This added information

Listing 6: The description of a parametrised n-bit adder in Pebble

```
BLOCK adder (n:GENERIC) [a,b:VECTOR(n−1..0) OF WIRE, c_in:WIRE]
                        [s:VECTOR(n−1..0) OF WIRE, c_out:WIRE]
VAR i
BEGIN
  GENERATE FOR i = 0..(n−1) DO
    GENERATE IF i = 0
      full_adder [a(0), b(0), c_in] [s(0), c_out]
    GENERATE IF i /= 0
      full_adder [a(i), b(i), c_out] [s(i), c_out]
END;
```

is then interpreted by the Pebble tools without interfering with the functional behaviour of the circuit. For instance, placement constructs are available and these could be used independently of the other functional constructs. Placement information can be explicitly included as co-ordinates, or else relational constructs, such as ABOVE or BESIDE can be used to declare how a block should be placed in relation to another, such as on top or to the side of another block. Pebble makes this possible by providing an additional higher abstraction over the blocks themselves. The placement constructs are actually block definitions capable of handling other blocks as parameters. By allowing a two-level language approach and managing to handle parameterised descriptions, some form of reasoning and manipulation about the blocks is achieved.

4   Embedded Languages

An alternative way to build an HDL (or any domain specific language), is to embed the new language into an existing general purpose programming language. Developing any domain specific language from scratch is a time consuming and tedious task. One has to face the burden of designing the language grammar and its syntax, provide mechanisms for typing, variable scoping and module management, as well as the developing of a parser and a compiler. Subsequently, users have to be thought the new language and provided with adequate development tools.

On the other hand, using the embedding approach to construct a new domain specific language, involves only the development of a library containing specific constructs that will handle the new language as a data object within the chosen programming language. In this setting, the embedded language will automatically inherit all of

the underlying features and infrastructure of the host language. A programmer can start using the domain specific language by simply referencing the library constructs, thus making use of the existing syntax, type handling features and development tools. This greatly reduces the development time of a language, which is ideal during experimentation and prototyping stages. However, despite a number of advantages, the downside of this approach is a loss in performance. Programs written by means of the embedded language cannot be highly optimised due to the overhead created by the host language, yet in certain circumstances, it is worthwhile to sacrifice some performance for time. The choice of the host language is extremely important. The embedded language will inherit all of the features of the host language, hence these features must fit the needs of the embedded language as precisely as possible.

In the context we are dealing with we consider our domain specific language to be a hardware description language. Numerous HDLs have been developed using the embedding approach using a wide range of programming paradigms, including object-oriented languages like C++, C# and Java (SharpHDL [Vel], JHDL [BH98]). However, general purpose functional languages have dominated the field of embedded languages, especially for HDLs [She05, CP07] . Functional languages like Haskell, are considered to be ideal host languages for a number of reasons, but mainly because of characteristics like strong typing, pattern matching, lazy evaluation and higher-order programming. Haskell in particular offers features like polymorphism, overloading and type classes, allowing multiple interpretations of the same description.

4.1   A Functional Representation for Circuits

Lava [Cla01] is a well known structural HDL embedded in the functional language Haskell. Hydra [O'D06] and Hawk [LLC99] are other examples of HDLs embedded in Haskell and are very similar to Lava. The usual trend of an HDL embedded in a functional language is to represent circuits as functions. This is how Lava, Hydra and Hawk describe circuits, meaning that a single-directional relationship is defined between the set of inputs and the set of outputs. The idea of using a functional language to describe hardware components is not a recent one, and as Mary Sheeran [She05] explains, the two fields (i.e. functional languages and hardware design), have been closely associated with each other in past research.

A simple HDL would represent basic circuit components as functions, while other larger circuits would be described in terms of these simple functions. For instance, Listing 7 gives the first-order functions `and2` and `xor2`, as implemented in Haskell, representing a two-bit and-gate and a two-bit exclusive-or-gate respectively. In this

Listing 7: Shallow embedding of primitive gates

```haskell
and2 :: (Bool, Bool) -> Bool
and2 (a,b) = a && b

xor2 :: (Bool, Bool) -> Bool
xor2 (a,b) = a /= b
```

Listing 8: A half-adder component described in terms of primitive functions

```haskell
halfAdd :: (Bool, Bool) -> (Bool, Bool)
halfAdd (a, b) = (sum, carry)
  where
    sum   = xor2 (a,b)
    carry = and2 (a,b)
```

case the boolean type represents a signal value of either low (False) or high (True). Following these kind of descriptions, larger circuits can be functionally described using the previously defined functions. Listing 8 illustrates how this can be achieved for the description of a half-adder component.

At a higher level of abstraction, parameterised functions are used to describe regular circuits for a specific component by means of recursion and pattern matching. A bit-adder is a circuit which adds one bit to a binary number, and is constructed as a chain of half-adder components connected together in a regular pattern. The function illustrated in Listing 9 is a typical example where recursion is applied over a previously defined function (`halfAdd`) to generate the required pattern of connections. Unlike Pebble or Pollux, the size of the circuit is not parameterised, but rather this is implicitly included as the length of the input list of booleans.

Listing 9: A Haskell function representing a bit-adder component

```haskell
bitAdder :: (Bool, [Bool]) -> ([Bool], Bool)
bitAdder (carryIn, []) = ([], carryIn)
bitAdder (carryIn, a:as) = (b:bs, carryOut)
  where
    (b, carry) = halfAdd (carryIn, a)
    (bs, carryOut) = bitAdder (carry, as)
```

Listing 10: Deep embedding of primitive gates by means of data types

```
data Signal = Bool Bool
            | Var String
            | Inv Signal
            | And (Signal, Signal)
            | ...

inv :: Signal -> Signal
inv a = Inv a

and2 :: (Signal, Signal) -> Signal
and2 (a, b) = And (a, b)
```

Embedding an HDL in a functional language is quite straightforward, and the nature of function definitions fit perfectly the requirements of circuit descriptions. The functional paradigm offers numerous features that correspond to the needs of HDLs. However, using only functions to represent circuits is not very practical, since the only operation that can be performed on these hardware models is simulation. This method of embedding a language is known as *shallow embedding*. In such an embedding, the circuit descriptions represent the semantics, and not the actual syntax, thus the hardware representation is not accessible for manipulation or interpretation, limiting the possibility to perform any other operation apart from evaluating the resulting output. In order to maintain the structure of the defined hardware, languages like Lava, Hydra and Hawk, make use of data types to represent the circuits' structure, while functions are used to construct the data type. These functions would not represent the circuit descriptions directly, but rather these act as the circuit generators. This approach is known as *deep embedding*. In this kind of environment, the hardware descriptions are first class data objects that have a full Turing-complete language (the host language) sitting above, thus the host language is capable to inspect and manipulate these objects at higher levels of abstraction. Listing 10 illustrates a simplified version of how such an embedding can be achieved. The functions `inv` and `and2` are used to construct the recursive data type representing the basic components of an *inverter* and an *and-gate* respectively.

Lava, Hydra and Hawk implement this approach in the functional language Haskell. All of the languages have unique implementation details, and each embedding provides different building blocks. However, these details are beyond the scope of this paper. We classify these implementations together and focus on the main characteristics. Accessibility to the circuits' structure enables previously impossible operations,

14

such reasoning about the representation, transformations and information gathering. For example in Lava, type classes are used to provide different interpretations of the generated circuits. Simulation is achieved by applying the actual values as inputs, where as if symbolic values (or variables) are used, a finite symbolic representation of the circuit is generated. This symbolic representation could be optionally processed and translated into other formats, such as a netlist description of the circuit, or a standard structural description in VHDL or Verilog. A layer of abstraction is achieved where instead of having netlist descriptors, we now have netlist generators. This two-staged process is separating the language used to describe the circuits from the actual semantics of the circuits, because the HDL is not describing the circuits directly but describing how the circuits are to be composed.

One of the major advantages of having an HDL deeply embedded in a functional programming language like Haskell, is the ability to inspect and reason about the circuits as data objects. Earlier we have examined the textual descriptions of circuits in VHDL or Verilog, and noted that these are not easily accessible. In Pebble and Pollux the use of parameterised descriptions (blocks or nodes), facilitate the designing of regular circuits for an arbitrary size. An n-bit-adder can be defined parametrically for any size, however a generic description of an array-like structure cannot be defined in Pollux, while Pebble offers only limited capabilities for these higher abstract descriptions. Lava, Hydra and Hawk handle these abstractions by means of higher-order functions. Higher-order functions are functions that handle other functions as arguments. This feature is a key benefit in having an HDL embedded in a pure functional language. Higher-order functions enable the manipulation and reasoning on the previously defined first-order functions. In Lava, higher-order functions are used to define connection patterns for generic sub-components (first-order functions) having the same function type. For example, the regular pattern that is used to compose a bit-adder is a common component arrangement know as a **row** that is widely used in hardware design (see Figure 1). A possible description for the row connection pattern is given in Listing 11, defined in terms of a higher-order function which can accept a range of circuit descriptions as a parameter. This definition is not bound to the specific types of the inputs, but rather it is defined for generic types. This connection pattern can be applied to any component having the function type (`c, a) -> (b, c)`, such as a half-adder or a full-adder. The given example shows how the higher-order function `row` is used to construct a bit-adder by passing the whole half-adder definition as the parameter.

Higher-order functions can be used in various kinds of connection patterns and structural arrangements of circuits. Other commonly used patterns are the column-shaped arrangement, which is similar to row but it is portrayed in a vertical position, and
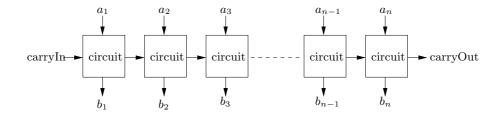
Figure 1: The row connection pattern

Listing 11: Definition of the row connection pattern and its application

```
row :: ((c, a) -> (b, c)) -> (c, [a]) -> ([b],c)
row circuit (carryIn, []) = ([], carryIn)
row circuit (carryIn, a:as) = (b:bs, carryOut)
  where
    (b, carry) = circuit (carryIn, a)
    (bs, carryOut) = row circuit (carry, as)

bitAdder :: (Bool, [Bool]) -> ([Bool], Bool)
bitAdder = row halfAdd
```

the grid-shaped structure which is a combination of the row and column connection patterns. Tree-shaped structures and butterfly circuits can also be defined for generic components by means for higher-order functions [BCSS98]. Higher-order functions adds credit to the software abstraction that is used to describe hardware, by providing more flexible constructs that enable hardware designers to describe large and complex circuits much more efficiently.

We have seen how HDLs benefit from functional languages to address the functional aspects of the circuits. However, challenges still exist, especially to handle the non-functional properties of circuits, such as placement or power consumption. Researchers have been questioned about how to introduce non-functional aspects of circuits without weakening the levels of abstraction that have already been established, thus maintaining the advantages brought by functional languages especially the ability to write abstract functions for generic descriptions. In Lava and other similar HDLs, higher-order functions are used to describe how circuits are connected to each other in a regular-shaped pattern. However, despite the fact that this approach enables the definition of correct structural descriptions, these software abstractions are actually discarding some level of detail rather than just hiding away information which can be retrieved later. The result of a higher-order function describing a con-

16

nection pattern is yet another function, which generates the data object representing the whole circuit. This means that we still end up with a single function describing a one-way relation between the inputs and the outputs, and no information is being stored to guide the inspection of individual components that constructed these patterns. The resulting data type representing the circuit describes how the primitive components are connected to each other to form the whole circuit. As an example, consider again the definition for the n-bit-adder. The resulting representation would provide information about how the circuit is constructed, but the boundary of each of the half-adders that are used is lost since each of these instances are defined in terms of primitive gates. By representing circuits as functions, hardware systems can be defined in a hierarchical manner, however the details of the hierarchy of components cannot be maintained. Non-functional information like the nesting of the components and how these are connected to each at different abstract levels is extremely important to perform operations such as floor-planning.

4.2   Combinators

An alternative approach that has been proposed for the description of structural hardware is to use a combinator calculus. Initially the idea was introduced in Ruby [SJ90]. A combinator calculus defines a number of mathematical constructs and properties about how functional elements or components can be combined together to construct larger elements, by placing circuit elements next to each other. The functional behaviour of the hardware is never really described, but rather this is defined indirectly as a result of how the circuit components are structurally placed in relation to one another.

The most trivial combinator presented by Ruby is *composition* (Figure 2(i)), where an element is placed next to another element, as long as the connecting sides have a common interface. *Repeated composition* is also allowed, enabling the same component to be placed alongside a copy of itself iteratively for a number of times. Another basic combinator is *parallel composition* (Figure 2(ii)), which describes how two components are placed besides each other such that these would operate independently, yet concurrently with each other. The idea of combinators as presented in Ruby is to build the functionality of a hardware system by describing the structure in terms of these basic combinators, subsequently, it is possible to analyse the placement details of sub-components over a two-dimensional plane. These kind of combinators are to be distinguished from the functional combinators (connection patterns) that where mentioned earlier. Functional combinators are defined by means of higher-order functions, which offer a different kind of abstraction for the structural descriptions. These kind of connection patterns join together the functionality of the given functions, re-
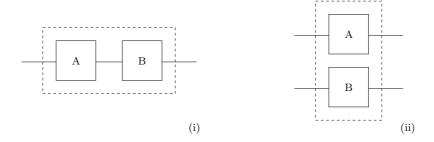
Figure 2: (i) Composition, and (ii) Parallel composition

sulting in a single function, having a one directional relation from the input to the outputs. Functional compositions do not offer any geometrical or topological information about the hardware components. On the other hand, Ruby combinators offer the geometrical relation of the components and how these are combined, which can be interpreted as a bi-directional relation.

In Lava it is possible to achieve similar capabilities to the combinators found in Ruby. We have already emphasised that functional composition is different from combinators, since the two operate differently. However, in a certain way this is not completely true, because functional composition can be optionally interpreted as combinators given some restrictions. Only a single directional relation can be described by means of functional composition, and all of the hardware representation has to be described in terms of functional composition from the lowest-levels of abstraction. If at some stage the structure of the circuit is not given in terms of these functional compositions, but instead it is defined as the functional description, then the final representation would have insufficient and inconsistent information about how the components are combined together. If this restriction is met then it is possible to achieve a hardware model which implicitly includes the relation between the different components, enabling to perform operations such as reasoning about placement, area management and power consumption. Listing 12 gives the definition of the serial connection pattern in Lava. This function composition is similar to the composition combinator of Ruby, but a single direction relation is defined. The output of the first function is directed as the input of the second function.

Wired [ACS05] is a more recent HDL implementation that follows the combinator approach. The ideas presented in Wired are based directly on the approach proposed in Ruby. Wired is embedded in the functional language Haskell, thus offering numerous advantages over the initial implementations of Ruby. In Wired, hardware descriptions define how components are tiled over a two-dimensional area. Functions are not

18

Listing 12: Serial functional composition in Lava

```
(->-) :: (a -> b) -> (b -> c) -> a -> c
(circ1 -> circ2) inp = out
  where
    mid = circ1 inp
    out = circ2 mid
```

used to represent the functionality of the hardware, therefore unlike Lava, Hydra and Hawk, the Haskell interpreter is not used to interpret the hardware representations directly. Instead, additional interpreting functions are needed, which are also implemented in Haskell, enabling to have both functional and geometrical interpretations of the hardware design.

Components in Wired can be either primitive functional elements (which can be instanced from Lava descriptions), or a group of components combined together by a combinator. A primitive element is described in terms of a surface and surface relation, where these provide information regarding the size (length and width) of the component, and the interface of each of the sides. Based on this kind of data, different components are either allowed or not allowed to be placed besides each other, and if two or more components are combined, then these are grouped together and regarded as a single component. The data type that is used by Wired to represent the combination of components is given in Listing 13. Wires are also treated as first class objects like components, and a number of wire configurations are available. Combinator constructs are similar to Ruby. The *besides* combinator ($*||*$), places the one component to the side of another, while the *below* combinator ($*=*$), places one component below the other. Other forms and variants of these combinators are available to the user. Typical hardware descriptions in Wired are defined at low levels of abstraction, however at higher levels, larger designs are managed by parameterised connection patterns. For example, Listing 14 gives the definition of the row connection pattern in terms of the Wired combinator $*||\tilde{}$ which is a variant to the besides combinator. This is a simple version of the row connection pattern, which takes the length parameter n explicitly. The features of Wired enable hardware designers to describe and analyse circuits, both in terms of the structural aspect and the functional relation of components, since combinators enable hardware descriptions to mix functional aspects with non-functional properties.

Listing 13: The data type used by Wired to represent hardware descriptions

```
data Description = Primitive Surface Relation
                 | Combined Combinator Description Description
                 | ...
```

Listing 14: A simple row connection pattern in Wired

```
rowN 0 _ = thinEmptyY
rowN n d = d *||˜ rowN (n−1) d
```

4.3   Representing Components

We have seen how the functional paradigm is particularly useful to define abstract descriptions for circuits, enabling hardware designers to rapidly develop larger and more complex systems. In research the most commonly used approach is to represent circuits as functions, similar to the approach taken by Lava, Hydra and Hawk. When using this approach the relations of the function composition are not clearly defined, and information about how sub-circuits are connected together to form larger circuit is not accessible, thus the hierarchical view of the components is lost. On the other hand, the relational language Wired makes use of combinators to enable an alternative way to compose circuit elements, giving a mathematical perspective about how different sub-components are connected to each other in order to compose larger components. This approach is quite successful in providing information for floor-planning, as well as to calculate non-functional properties such as area and wire length. A less popular alternative is to represent the circuits as autonomous components. This means that the components are described such that each of these can be clearly distinguished from each another, and optionally used when required explicitly. Such components can be instantiated to different formats for a wide range of operations. For example, if a functional simulation is needed then the components would be instantiated for functional analysis, where as if another operation is required, such as relational analysis, hierarchical analysis, or transformation procedures, then the components would be instantiated for the appropriate need. Dual-Eval [BWAH97] and HeDLa [Pac07] are such HDL examples that follow the component-based approach.

Dual-Eval is embedded within the programming language Lisp. The Lisp interpreter can only recognise list structures, and each list is interpreted by capturing the first element as the operator while the succeeding elements are regarded as arguments to this operator. Using this framework Dual-Eval represents a component as a list

Listing 15: A half-adder component as defined in Dual-Eval

```
(HALF−ADDER∗) =
   '( HALF−ADDER (A B)
                 (SUM CARRY)
                 (
                    ( G0(SUM)    B−XOR (A B) )
                    ( G1(CARRY) B−AND (A B) )
                 )
                 NIL )
```

of sub-circuits, and the component boundary is maintained by means of the Lisp quotation. The quotation construct instructs the Lisp interpreter not to evaluate the upcoming list, but rather treat it as a literal structure. Circuit components are defined by a preceding quote, hence, these are not evaluated but are captured as components. Subsequently, when larger systems are composed from other components the hierarchical view of the system is well defined, and is made available for interpretation or reasoning. Listing 15 gives the definition of a half-adder component in Dual-Eval.

HeDLa is another HDL, embedded in the functional language Haskell. In HeDLa the structural description of a circuit is not given directly as a Haskell function, but rather the structural description is wrapped within a record. This kind of representation cannot be directly evaluated, but has to be instantiated before it can be processed any further. Additional information can also be annotated to the record, such as string values for the component name, the input wires and the output wires. The connections between components are not regarded as arguments to Haskell functions, but these are handled by the HeDLa implementation which maps together the string representation of each connection. To make use of sub-components the special *use* construct is required to explicitly indicate how the components are to be connected. The *use* construct also maintains structural information about how the component has been composed. A typical definition in HeDLa of a half-adder component is given in Listing 16.

Clearly, the major disadvantage of these HDLs is the syntactical overhead that is present. The syntax that is required to describe circuits as components is not as clean as when the same circuits are described as functions. However, it is hard to abstract away details about the structure of the components, whilst indirectly maintain the equivalent necessary information. A similarity to VHDL can be perceived in these kind of circuit descriptions, especially HeDLa. The circuit details are given implicitly

Listing 16: A component definition of a half-adder as given in HeDLa

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

Listing 17: A simple combinator description in HeDLa

```
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 left2
                      & use c2 (inputs c2) (outputs c2)
      }
```

and clearly indicating the type of description as structural, and which connections are the inputs and the outputs. The advantage of having a component perspective as offered by HeDLa is that it allows access over the hierarchical architecture of a hardware system. One advantage in having a component-based HDL is to analyse and manage effectively the non-functional aspects of the circuits. For instance, HeDLa targets the placement of the defined components by means of constructs that combine circuits together. One such constructor is a sequential composition combinator, which combines two circuit components by placing one after the other. Listing 17 gives the implementation details of this operator. As can be seen, the input and output details of the two components are being integrated together to form a new set of input and output tuples, that would describe the interface of the new component. This new description is specifying how the first component is to be connected to the second component.

22

5 Meta-Languages

In the previous sections we have seen how the functional aspects of circuits can be represented by means of an HDL embedded in a functional language such as Haskell. Non-functional properties of circuits however, present a particular challenge. It is true that in languages like Wired, Dual-Eval and HeDLa, some non-functional aspects are handled, in particular component placement, but the approach ignores the perfect embedding that can be achieved in a functional language, since these HDLs do not represent the circuit descriptions as functions. On the other hand, HDLs like Lava, Hydra and Hawk, use functions to represent circuits, and are in general easier to use and understand. The problem with these languages is that certain non-functional aspects of hardware design cannot be handled through the modular abstraction that is present. Researchers have been looking into meta-programming techniques for a possible solution. Meta-programming will also serve as an alternative way, in which hardware can be described.

Meta-programming can be defined as the ability to write programs that are able to create and modify other programs as data-objects. New embedding solutions in meta-languages are being proposed, where both the functional and non-functional aspects of the hardware can be managed. Hardware representations can be regarded as an object-program within a meta-language, hence, the hardware model can be inspected, reasoned about, or translated into other formats by means of the meta-language itself. Additionally meta-languages are capable to create other object-programs, which could represent components, and later used to produce other complex programs representing larger circuits. In such a setting, both functional and non-functional properties can be handled, since the meta-language does not only provide access to the circuit representations, but it even provides control over the circuit generators. Furthermore, in meta-programming a circuit generator represents a whole family of circuits, hence, all the enhancements and optimisations performed on the circuit generators will be reflected within the resulting circuits.

5.1 Template Haskell

With the introduction of C++ templates, programmers were allowed to set-up a macro-like process that operates on a C++ program that is executed at compile time. Such a feature allows the development of a program that can be reconfigured and optimised during the compilation process. This results in the generation of a program that is dynamically modified to fit a number of criteria or constraints. More recently, similar extensions have been introduced into functional languages. Template Haskell [SP02] is a library that provides support for compile-time meta-programming

Listing 18: The toggle circuit definition

```
toggle = inv (delay toggle)

toggle = inv (delay (inv (delay toggle)))
toggle = inv (delay (inv (delay (inv (delay toggle)))))
toggle = ...
```

in the function language Haskell. Template Haskell uses a quasi-quotation mechanism for the Haskell language, and it represents the quoted code as an abstract data type. A programmer is therefore able to compute programs, or parts of a program, rather than actually write them. It is possible to write programs that would process and generate Haskell code, and hence producing other programs as output. Template Haskell provides access to the structure of the quoted code by means of the built-in case analysis of Haskell, and is capable to perform transformations and modifications to the code of other programs.

A variant of the Hydra [O'D04] HDL has been embedded in Template Haskell. This implementation of Hydra is based on program manipulation and it proposes a solution for the handling of non-functional properties of circuits. The main problem that is targeted in Hydra is that of sharing. This problem is clearly visible when feedback loops are present within sequential circuits, such as a toggle circuit. The first line of code in Listing 18, gives a definition example of the toggle circuit. Due to the referential transparency of a pure functional language, the left hand side of an equation can be replaced by the right hand side. The interpreter of the functional language tries to resolve all the function calls, and since a feedback loop is usually defined in terms of a function that refers to itself, this would supposedly result in an infinite number of calls. In Haskell, such infinite function calls would be handled by means of lazy evaluation, however this does not provide the correct non-functional properties of the circuits' structure. As Listing 18 illustrates, the resulting circuit is not the same as the one that has been described, since the number of inverter and delay components is not clear for the interpreter.

In previous versions of Hydra and other HDLs, sharing is solved by the use of monads, or a manual implementation for labelling, where the inputs and outputs are either implicitly or explicitly tagged, indicating exactly how the connections within the representation of the circuit are to be structured. In latest version of Lava this problem is solved by a technique known as observable sharing. The Hydra implementation in Template Haskell, solves this problem by applying compile-time meta-programming techniques to automatically transform circuit descriptions into a manageable for-

Listing 19: The toggle circuit definition in the Template Haskell

```
circ_defs_toggle = [d|
  toggle = inv (delay toggle)
  |]

$(transform_module circ_defs_toggle)
```

mat where sharing can be identified. By quoting the original definition, Template Haskell stores the syntax tree of the defined function as a data type. Hence, a special transformation module is used to insert labelling information within the circuit description automatically. This module handles the problem of referential transparency and sharing, while at the same time it reduces the syntactical overhead that would have otherwise been present, if the user had to insert labelling information explicitly. In Listing 19, the toggle circuit is described as a quoted function. The transformation module is a function which inspects and transforms the given quoted function, while the meta-construct '$' evaluates the returned function which is annotated with labelling information.

Template Haskell is an extension to Haskell, and it does provide access to all of the other features of Haskell. An HDL such as this version of Hydra, is still able to provide features like parameterised descriptions and higher-order functions. Template Haskell adds an additional or alternative abstract perspective, where the quotations and other meta-constructs provide a two-level language. This means that a second language (the meta-language) is used to inspect and reason about the original language, in this case, the Haskell language itself.

5.2 Meta-ML

The work accomplished on meta-programming and hardware design by means of Meta-ML and MetaOCam [Tah06, KT04, KST04], spreads over a wide range of methodologies and techniques. When designing hardware systems, the resources can be fixed, in particular the space or area in which the system will have to be implemented. This is noted especially when considering reconfigurable hardware, such as Field-Programmable-Gate-Arrays (FPGAs). For this reason, the development of HDLs should aim at the limited resource usage and performance issues, but unfortunately abstractions tend to cover up these details. This is complementary to the inclusion of non-functional properties of the circuit in the hardware descriptions.

Listing 20: The power funcion defined in terms of staging constructs

```
let rec power n x = if n = 0
                    then .<1>.
                    else .< .~x * .~(power (n-1) x)>.

// This function generates .<fun x -> x * x * x * 1>.
let power3 = .<fun x -> .~(power 3 .<x>.).>.
```

Apart from solving these kind of problems directly, the work done on Meta-ML offers an alternative approach to hardware design. The proposed solution is Program Generation, where a two-level language setting is used to generate descriptions for hardware systems. In a two-level language approach, the object-program that is being generated can be control be the meta-language, hence, optimisation issues can be handled during the generation stages. Furthermore, these program generators would represent a whole family of circuits and not just a single specific design. This kind of abstraction presents a high level of flexibility and reusability, which can easily be applied for the description of large circuits having regular-shaped structures. Program generation would also enhance the current verification techniques and model checking, since analysis and reasoning can be applied on the generator (a whole family of circuits) rather than on individual circuits. A generator that is formally verified for correctness should supposedly generate correct circuits.

The meta-constructs that are used are quotations (<>) and splicing (~) constructs. Quotations stops the interpreter from evaluating the contents, thus maintaining the code as an object-program, while a splice joins object-programs together. Listing 20 gives an example of how these constructs can be used. The illustration gives the recursive definition of the power function with added meta-constructs, which actually acts as a parameterised generator for the power function. This kind of descriptions can be used to describe hardware systems, thus describing generators for a whole family of circuits. Abstract interpretation is another technique that is presented in [KST04], where certain aspects of the object-program are interpreted during its generation, thus performing a number of optimisations. Unlike Hydra in Template Haskell, the required transformations and modifications are handled during the generation process. This eliminates completely the presence of intermediate object-programs, that require further processing, and therefore improves the overall efficiency.

The recent development of this work is a prototype language called Uccello [ET07]. Uccello formalises a graphical description language that includes staging constructs,

which can also be used to describe hardware graphically. In the introduction we argued that graphical diagrams are difficult to handle large complex circuits, and that is why we use abstractions over textual descriptions. The Uccello language tries to overcome this situation by formalising transformation techniques from textual descriptions to graphical descriptions and vice versa. Hence, the advancements that have been made on software languages can be conceptually imported to graphical descriptions, whilst the clarity of graphical diagrams can aid the textual descriptions. For instance, sharing is clearer in a graphical representation of a circuit, than in a textual description. This kind of interpretation between textual and graphical representations should provide an in-site to placement and wire routing problems.

5.3 *reFL^{ect}*

reFL^{ect} [GMO06] is a strongly typed functional language with built-in meta-programming features. The language was developed by Intel, and it is based on the functional language *FL*, but with extended reflection features. *reFL^{ect}* is the main programming language used with the Forte tool [SJO$^+$05]; a hardware verification system used by Intel. Both *reFL^{ect}* and Forte were purposely developed for the development of applications in hardware design and verification, and are mostly applied for model checking, decision making algorithms and theorem provers for hardware analysis.

The *reFL^{ect}* language uses quotation and anti-quotation constructs to compose or decompose expressions (object-programs) written in the *reFL^{ect}* language itself. This provides a form of reflection within a typed functional paradigm setting. The metalanguage features in *reFL^{ect}* enable the programmer to access the underlying data structure that is normally used to represent the abstract syntax tree of the language. This gives the possibility to create, manipulate and inspect programs written in *reFL^{ect}*, whilst using the *reFL^{ect}* language itself. A pattern matching mechanism is available, and this can even be applied directly on the object-programs, allowing unevaluated expressions to be inspected and analysed. This provides the possibility for the developer to modify or transform object-programs at runtime.

reFL^{ect} has been used to developed a simple HDL similar to Lava, which we refer to as *reFL^{ect}* HDL [MO06]. Listing 21 gives an extract of how a simple basic component can be defined. The built-in AND operator is being used to represent the and-gate directly, and boolean values would represent signal values. A separate function (ANDT) is defined to handle the quoted expression for the AND operator. This function and the AND operator are hence overloaded, such that, if boolean values are supplied as inputs, then the evaluation would result in a simulation of the component, while if

Listing 21: The and-gate and multiplexer definitions in *reFL*^{ect} HDL

```
// Forward declare a netlist−building AND gate
forward_declare {ANDT :: term −> term −> term};
// Overloaded combinational, and netlist−building AND gate
overload AND AND ANDT;
// Define the term−building AND gate in terms of the overloaded function
let ANDT t1 t2 = {| {AND :: *a −> *a −> *a} 't1 't2 |};


// Multiplexer defined in terms of the basic gates
let mux (s, (a, b)) = OR (AND s b) (AND (NOT s) a);
```

object-programs are given, then the evaluation would splice these sub-components together with the AND operator, resulting in the object-program of the component. A typical circuit description of a one-bit multiplexer is also given to illustrate that hardware descriptions at higher levels of abstraction would follow the functional style of representation. The difference between *reFL*^{ect} HDL and languages like Lava, is the way the deep embedding is handled, where instead of having data types to represent the circuits, in *reFL*^{ect} HDL hardware descriptions are represented by functions similar to the shallow embedding approach. The meta-constructs are what provides access to these functions as abstract data types.

A functional meta-language like *reFL*^{ect} offers numerous benefits. Research and experimental work indicate that the meta-programming characteristics of *reFL*^{ect} could provide enhancements for the way hardware is currently being designed. Simple circuits could be easily embedded within *reFL*^{ect} similar to how Lava is embedded in Haskell. More complex circuits, in particular regular-shaped circuits, could be described by applying transformation techniques and manipulations over the object-programs that represent the smaller sub-components. Connection patterns can be applied over the object-programs rather than over the functions. This provides a setting in which object-programs can be annotated with non-functional information. Sharing and the referential transparency problem can also be handled by means of meta-constructs, similar to the Hydra implementation in Template Haskell.

6   Conclusions

The primary objective behind hardware description languages is to provide features that allow hardware designers to describe and compose circuit descriptions. Circuit

descriptions in languages like, VHDL and Verilog [LMS86, Ope93], can then be reasoned about through the use of external tools. Conditional and iterative generation constructs, added a higher form of abstract description enabling circuit descriptions to be composed during compilation time following a set of parameters. By adopting the embedding approach, the hardware descriptions are regarded as circuit generators rather than circuit descriptions. Hardware description languages such as Lava, Hydra and Hawk [Cla01, O'D06, LLC99], enable designers to reason about the circuits, since these are generated as data objects within a full-blown Turing language. As we have discussed throughout the previous sections, various levels of abstraction have been explored over hardware description languages; from modular circuit descriptions, to parameterised circuit descriptions (two-level languages), to circuit generators (embedded languages). Each abstract level provides hardware designers more flexibility and efficiency in describing circuits.

The introduction of meta-constructs within the functional setting exposed other areas, including abstraction techniques, which are currently being researched and explored for further enhancements in hardware design. The primary aim in having a multi-staged hardware description language is to provide access not only to the generated circuits but also to the generators themselves. Having access to the circuit generators as data objects within a programming language, should provide features and capabilities that researchers can take advantage of. The starting issue when adopting meta-functional programming is to enable the analysis, reasoning and manipulation of the circuits' structure. The use of a meta-functional language provides the possibility to perform these operations, while maintaining the capability to describe hardware designs in terms of different abstractions, like parameterised descriptions and higher-order functions. Problems like sharing can also be handled by means of the two-level language approach that is achieved through the use of the meta-constructs. Additionally, by accessing the circuit generators and reasoning about the generators themselves, certain non-functional properties could be induced before the actual generation of the circuit. For instance, by examining the generator, one should be able to extract the generation procedure that is used to create the circuit, thus obtaining information about, which parts of the circuit have been generated by the generator, and in what order. This kind of information can be interpreted and used to aid during the physical placement or floorplanning of the circuit, especially if language constructs such as combinators are added.

Another area which could benefit from multi-staging techniques is that of hardware compilation. Embedded HDLs have been used to describe hardware compilers [CP02], however, the lack of information has lead to non-optimised circuit descriptions. By adopting a meta-functional language within such areas, should provide the possibility to introduce post-compilation optimisations or abstract interpretations that would

reflect changes in the generated circuits. Hence a meta-language should provide a means to maintain information that would have otherwise been unhandled, and make use of it to optimise the hardware compilation process.

References

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

[BCSS98]   Per Bjesse, Koen Linström Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 1998.

[BH98]   Peter Bellows and Brad Hutchings. JHDL - an HDL for reconfigurable systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[BWAH97]   Bishop C. Brock and Jr. Warren A. Hunt. The dual-eval hardware description language and its use in the formal specification and verification of the fm9001 microprocessor. *Form. Methods Syst. Des.*, 11(1):71–104, 1997.

[Cla01]   Koen Linström Claessen. *Embedded Languages for Describing and Verifying Hardware.* PhD thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology, April 2001. Dept. of Computer Science and Engineering, Chalmers University of Technology. Ph.D. thesis.

[CP02]   Koen Claessen and Gordon J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, April 2002.

[CP07]   Koen Linström Claessen and Gordon J. Pace. Embedded hardware description languages: Exploring the design space. In *Hardware Design and Functional Languages (HFL'07), Braga, Portugal*, March 2007.

[CPHP87]   P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[ET07]   Stephan Ellner and Walid Taha. The semantics of graphical languages. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 122–133, New York, NY, USA, 2007. ACM.

[GMO06]   Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

[KST04]   Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 249–258, New York, NY, USA, 2004. ACM Press.

[KT04]   Oleg Kiselyov and Walid Taha. Relating fftw and split-radix. In *Embedded Software and Systems, First International Conference, ICESS 2004, Hangzhou, China, December 9-10, 2004, Revised Selected Papers*, volume 3605 of *Lecture Notes in Computer Science*, pages 488–493. Springer, 2004.

[LLC99]   John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within haskell. *SIGPLAN Not.*, 34(9):60–69, 1999.

[LM98]   Wayne Luk and Steve McKeever. Pebble: A language for parametrised and reconfigurable hardware design. In *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, pages 9–18, London, UK, 1998. Springer-Verlag.

[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 in reFLect. In Mary Sheeran and Tom Melham, editors, *Sixth International Workshop on Designing Correct Circuits: Vienna, 25–26 March 2006: Participants' Proceedings*. ETAPS 2006, March 2006. A Satellite Event of the ETAPS 2006 group of conferences.

[O'D04]   John O'Donnell. *Embedding a Hardware Description Language in Template Haskell*, chapter Embedding a Hardware Description Language in Template Haskell, pages 143–164. Springer Verlag, 2004.

[O'D06]   John O'Donnell. Overview of hydra: a concurrent language for synchronous digital circuit design. *International Journal of Information*, pages 249–264, 2006.

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

[Pac07]   Gordon J. Pace. Hedla: A strongly typed, component-based embedded hardware description language. 2007.

[RH91]   F. Rocheteau and N. Halbwachs. Pollux, a lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, June 1991.

[She05]   Mary Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.

[SJ90]   Mary Sheeran and Geraint Jones. Circuit design in ruby. In Jrgen Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.

[SJO$^+$05] Carl-Johan H. Seger, Robert B. Jones, John O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.

[SP02]   Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[Tah06]   Walid Taha. Two-level languages and circuit design and synthesis. In *Designing Correct Circuits*, 2006.

[Vel]   Christine Vella. Sharphdl: A hardware description language embedded in c#. University of Malta.