Meta-Functional Languages for Hardware Design and Verification

Gordon J. Pace
Department of Computer Science
University of Malta
gordon.pace@um.edu.mt

Christian Tabone
Department of Computer Science
University of Malta
christian.tabone@um.edu.mt

Abstract

The approach of embedding hardware description languages in general-purpose languages has been widely explored in the literature and has been shown to aid hardware design and verification. In this paper we explore the use of a meta-functional language reFL^{ect} as a host language for a hardware description language. We show how this approach aids the development, analysis and manipulation of embedded objects, whilst at the same time we keep meta-programming features largely invisible to the hardware designer. We illustrate the use of these techniques in supporting circuit placement techniques and automatic model checking of hardware compiler invariants.

Index Terms

Embedded Languages, Meta-Functional Languages, Hardware Compilers, Hardware Verification

1. Introduction

The embedding of a domain specific language (DSL) within an existing general purpose programming language has been widely researched, and the approach is well understood and documented. The embedded language is usually developed as a library within the host language, providing a number of key benefits. For instance, there is no need to define a new syntax, since the embedded language will directly inherit the syntax of the host language. Actually, the embedded language inherits all of the features and characteristics of the host language, such as the type system and the module system, as well as all of the underlying tools like compilers, interpreters and debuggers. When following the embedding approach the constructs that make up the DSL are defined as first class objects within the host language, thus creating a two-stage language approach, with a fully developed programming language sitting above the DSL. This enables the analysis, transformation, and general manipulation of the language constructs just like any other data object. Functional languages, such as Haskell, have proved to be a popular choice for embedding languages. Features such as strong typing, lazy evaluation, pattern matching and higher-order functions, make them ideal host languages offering a variety of abstraction techniques. One particular domain in which the embedding approach has been extensively applied to, is hardware design. Various languages, such as Lava [1] and Hawk [5] have been embedded within the functional language Haskell. These embedded hardware description languages (HDLs) are able to access the hardware descriptions as first class objects, in fact the definitions are actually regarded as circuit generators rather then just simply circuit descriptions. This abstraction enables the hardware designer, not only to perform circuit generation, but also to analyse, transform and interpret the circuit. These features enable the possibility to gather static information about the circuit, simulate, test and verify the circuit, as well as apply re-timing translations and generate netlists. Circuits can be described at even higher levels of abstraction by means of higher-order functions and parameterised circuits. For example, connection patterns are defined as higher-order functions, where whole circuit descriptions are passed as parameters, whilst the size of the circuit is determined also from the inputs. It has been shown how this high abstract level can be pushed even further by having a behavioural description for the hardware [3]. A behavioural description is more closely related to the specification of the circuit than the actual structure, hence this high level description needs to be synthesised into the structural equivalent. In [3], Pace and Claessen manage this by parameterising circuit descriptions with high level language constructs.

Furthermore, such hardware compilers can be analysed and verified to satisfy certain properties by means of finite state model checkers [8].
More recently, the use of meta-programming techniques for the embedding of HDLs has started to be explored [6], [11], [7]. A meta-programming language enables the development of programs that are able to compose and manipulate other programs or even themselves. This is commonly known as reflection. Meta-programming provides an opportunity not only to access the circuits being generated, but even the generators themselves that created these circuits. Therefore, reasoning about the circuit generators is made possible, providing a means to inspect and analyse the composition of circuits in terms of nested blocks, thus achieving a level of abstraction higher than when embedding an HDL in a normal functional language.
In this paper we present Shade, a HDL embedded within the meta-functional language *reFLect*, built upon our recent results presented in [9]. We show how the nesting of circuit blocks can provide placement information for the circuits, that can be combined with other user defined placement annotations. We also present the verification approach used on hardware compilers, and discuss how meta-programming enables us to access the circuit generators, thus manipulating and transforming them in such way that the verification observers can be applied autonomously. This approach avoids user intervention thus reducing the possibility of errors.

2. Embedding a HDL in *reFLect*

Typically, when embedding a DSL, a deep-embedding is required since one would want not only to generate programs, but allow the possibility to give them different interpretations as may be required, thus complex data objects are defined to provide access to the underlying syntax of the DSL. On the other hand, in a meta-programming language a shallow representation is sufficient since the language constructs can be quoted, resulting in having access to the described programs as data objects.
reFLect [4] is a strongly-typed functional language with meta-programming capabilities. *reFLect* was developed as part of the Forte tool [10]; a hardware verification system used by Intel. *reFLect* provides quotation and antiquotation constructs, allowing the composition and decomposition of unevaluated expressions, defined in terms of the *reFLect* language itself. These meta-programming constructs allow a form of reflection within a typed functional paradigm setting, enabling direct access to the structure of programs as data objects. This is made possible by giving access to the internal representation of the abstact syntax tree of the quoted expressions. Traditional pattern matching can even be used on this representation, allowing the structure of unevaluated expressions to be inspected and interpreted according to the developer's requirements. Furthermore, by combining the pattern matching mechanism with the quotation features, the developer is able to modify or transform the quoted expression at runtime before evaluation.

2.1. Shade

Shade is a HDL we have developed, embedded in *reFLect*. Circuits are strongly typed, but are internally stored as untyped quoted *reFLect* terms — simply maintaining an unevaluated expression of a circuit definition, effectively providing the actual structural description which can still be interpreted directly to obtain its output.

```
lettype *a sig = ...

high, low :: bool sig
inv :: bool sig -> bool sig
and2 :: (bool, bool) sig -> bool sig
delay :: bool -> bool sig -> bool sig
```

Internally, the primitive gates are simply quoted versions of their boolean operator counterparts, using antiquotations to deal with quotations in their parameters. These primitive functions can be used by the hardware designer to define larger circuit descriptions by simply following the functional paradigm. Note that, the `and2` gate takes one input stream made up of pairs of boolean values. This approach of having wires of structures as used, for instance, in Hawk [5], requires explicit use of functions, to convert the signal structure back and forth to the structure values using the polymorphic functions `zipp` and `unzipp` functions[1]. For instance a two-bit multiplexer circuit would be defined as follows:

```
let mux s_ab =
  val (s, ab) = unzipp s_ab in
  val (a, b)  = unzipp ab in
  or2 (zipp (and2 (zipp (inv s, a)),
             and2 (zipp (s, b))));
```

Also note that the `delay` latch component takes a boolean value which sets the initial output. To create loops in a circuit, Shade provides a fix-point operator illustrated in the following circuit which outputs whether its input wire has always been high (in the past, up to and including now):

```
let alwaysTillNow x =
```

1. Most of the examples given in this paper will omit the zipping and unzipping functions

```
  loop ok . and2 (zipp(x, delay T ok))
```

Finally note that reuse of user defined circuit components is identical to the use of the primitive components:

```
let alwaysInThePast x =
  delay T (alwaysTillNow x)
```

2.2. Circuit Interpretations

Shade provides various interpretations for the circuits described. Since internally, a circuit is simply a quoted function denoting the result of evaluating a circuit, simulation simply involves unquoting the circuit. Other interpretations involve traversing the circuit using the meta-programming characteristics found in *reFLect* and applying an appropriate interpretation. The possibility to apply pattern matching over quoted expressions, enables us to inspect, analyse and translate the structure into other formats, such as netlist generation from *reFLect* circuit descriptions.

Apart from simulation and netlist generation, Shade also supports circuit verification though the use of external model checkers. Instead of embedding a property language, we take an observer-based approach, in which properties are described as other circuits which take the inputs and outputs of the circuit to be verified and outputs a single boolean output. Although limiting the verifiable class of properties to safety properties, this approach avoids the need of an additional embedded language. Currently, Shade is connected to the NuSMV model checker [2].

For instance, to verify that no matter what the value of the selector wire is, if both inputs of a multiplexor are equal, then the output is equal to this value. This property can be expressed as follows. Note that equality (`===`) and implication (`==>`) operators are built using the primitive gates.

```
let obs_mux ((s, (a, b)), o) =
            (a === b) ==> (o === a)
```

Passing this observer as an input to the NuSMV interpretation function `writeToSMV` generates an NuSMV model which can be verified.

2.3. Marking Blocks in Circuits

In *reFLect*, as in most other embedded HDLs such as Lava and Hydra, one views and defines circuits as functions. As a circuit description is unfolded, all the internal structure (implicit in the way the generators are invoked) is lost, and all that remains is a netlist of interconnected gates. Structure is thus lost in the process, which may store information which could be useful in circuit analysis and processing.

Block tagging can be challenging when the HDL is embedded in a purely functional language. In our case, the meta-programming features enable a straightforward approach to block definition — since a circuit is simply an unapplied function, we simply delay evaluation of the function by quoting it, leaving the reuse of the circuit unaffected, and simply involves unshelling additional quotations for simulation or analysis. The blocks are then however used in various interpretations. In particular, when producing netlist descriptions for placement, all marked blocks are tagged so as to enable shared placement of multiple copies of the same block. Similarly, VHDL output of circuits produced in Shade can share the same block (entity) structure, maintaining an isomorphism between the Shade and VHDL description, thus simplifying the use of VHDL tools on Shade descriptions.

2.4. Placement Annotations

Another area in which we are interested, is how the marking of blocks can be used to help in the placement and layout of the final circuit. We approach this by providing the possibility to add user defined placement annotations to the circuit description. Having a meta-language at hand, we can add pieces of information which can be accessed through the use of meta-programming constructs, thus have a significant value when looking the structure. However these annotations do not interfere with the actual functionality of the circuit, and are therefore discarded when simulated. Similar to the marking of blocks, we use the meta-programming capabilities to enhance the syntax of the circuit structure without interfering with the semantics. By doing so, we are able to define non-function properties and the function of the circuit within the same description, whilst maintaining a clear distinction between the two.

In Shade, we provide the means of adding placement annotations to blocks — currently tagging blocks to lie beside or below each other. Semantically these functions are of no significant value, however, these are maintained within the circuit structure and can therefore be accessed and interpreted.

This relative placement information describing a circuit's layout, are translated into Relative Location (RLOC) Constraints [12], which are added to the VHDL description as attributes. These user constraints are interpreted by the Xilinx development tools and can be used with most of the Xilinx FPGA series.

3. Hardware Compilers

The use of embedded languages provides a way of pushing up the level of abstraction used for circuit descriptions. In the case of regular circuits, concise descriptions in the host language can be used to describe large, complex circuits, using modularity and abstraction techniques from the host language. However, in most such approaches the abstraction layers still leads to a structural description of the circuit. An alternative approach which is nowadays increasingly used, is that of automatic hardware synthesis or compilation from a high level algorithmic description directly into a structural description.

In [3], Pace and Claessen present a framework in which such algorithmic, or behavioural descriptions can be merged within the structural descriptions, by following the embedding approach. The idea is to develop another layer on top of the already existing embedded HDL. The behavioural description language is embedded by specifying the syntax in terms of a datatype, and the structural description for each of the language constructs are described. The compilation procedure corresponds to a circuit parameterised by the data object representing the language constructs.

One issue with hardware compilation is that the compilation procedure should ideally be verified correct. In practice this can be a long and tedious process. As the trend to develop various, but highly domain specific languages to solve a problem is adopted, the more frequently this has to be done, sometimes by the same hardware designer. In [8], Pace and Claessen showed how certain hardware compiler invariants can be model checked automatically through the use of the compiler description and structural induction over the program type. However, when using a functional language such as Haskell (as was used in [8]), with no meta-programming capabilities, transforming the compiler description into the verification framework had to be performed by hand, even if it follows a uniform pattern. Nonetheless, the disadvantage with this approach is that the transformation function might not match exactly the structure of the hardware compiler, due to user induced errors since the descriptions are defined separately by the hardware designer. Despite the relation between the two circuit generators, when using a language like Lava, there is no possible way to maintain a programmable connection between the two. In Shade, by using the meta-programming features of *reFL^ect*, we can automatically allow the designer writing a domain specific hardware compiler to verify such properties using a finite state model checker.

3.1. Compiling Flash

We will illustrate the process by looking at the embedding of a hardware compiler in Shade — using the Flash language from [3], which is a basic language with imperative programming constructs. Programs in Flash are simply instances of a datatype in *reFL^ect*:

```
lettype Flash
  = Skip
  | Shout
  | Delay
  | Sequential Flash Flash
  | IfThenElse (bool sig) Flash Flash
  | Parallel Flash Flash
  | While (bool sig) Flash;
```

Note that Flash has the standard imperative language features, such as sequential composition and conditional, but also supports a fork-join construct. For simplicity, programs in Flash have a single output wire low by default, but which can be pushed up to high (for one clock cycle using the `Shout` instruction. The basic instructions `Shout` and `Skip` terminate immediately (in the same clock cycle), whereas `Delay` takes one clock cycle to terminate. Flash programs will be compiled into circuits with one input wire `start` (which will be high for one clock cycle to start the program), and two output wires `shout` and `finish` (the first is the output of the program, while the latter will be high for one clock cycle when the program has terminated). For more details about Flash and its compilation refer to [3]. The hardware compilation schemes for Flash are given in figure 1. In Shade, the constructs can be implemented directly using pattern matching over the datatype, and calling the compile function recursively over the sub-programs. Consider two of the syntactic cases:

```
letrec compile Shout start =
  let shout  = start in
  let finish = start in
  (shout, finish)
/\     compile (Sequential p q) start =
  val (pShout, pFinish) = compile p start in
  val (qShout, qFinish) = compile q pFinish in
  let shout = or2 (pShout, qShout) in
  (shout, qFinish);
```

3.2. Compiler Invariants

More interesting language properties can be proved by using structural induction over the language constructs. By proving that the circuits produced by the compiler always satisfy the invariant property, as long as the property is satisfied by the subprograms, we can prove that any compiled program satisfies the invariant property. For instance, for Flash one can prove an invariant

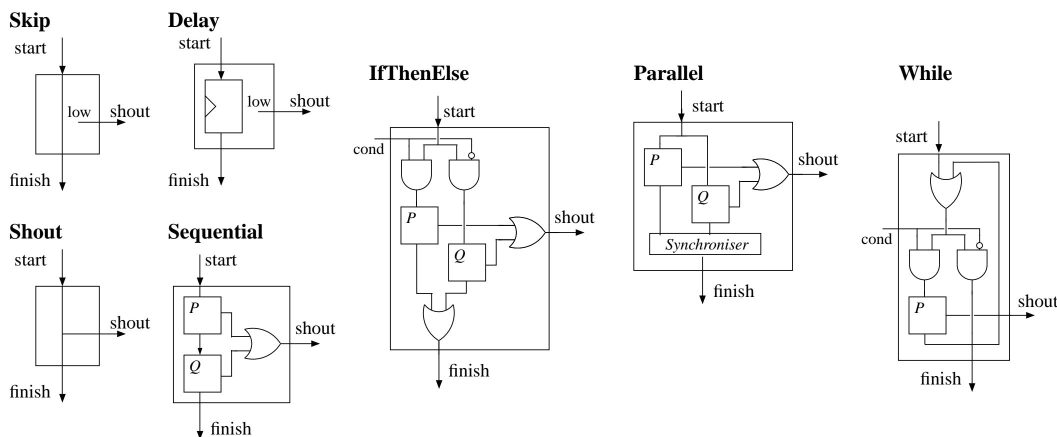

Figure 1. Hardware designs of Flash

π over a program using structural induction as shown below:

$$\frac{\begin{array}{l}\vdash \pi(\mathtt{Skip}) \\ \vdash \pi(\mathtt{Shout}) \\ \vdash \pi(\mathtt{Delay}) \\ \forall \mathtt{c}, \mathtt{P}, \mathtt{Q} \cdot \pi(\mathtt{P}) \wedge \pi(\mathtt{Q}) \vdash \pi(\mathtt{IfThenElse\ c\ P\ Q}) \\ \forall \mathtt{P}, \mathtt{Q} \cdot \pi(\mathtt{P}) \wedge \pi(\mathtt{Q}) \vdash \pi(\mathtt{Sequential\ P\ Q}) \\ \forall \mathtt{P}, \mathtt{Q} \cdot \pi(\mathtt{P}) \wedge \pi(\mathtt{Q}) \vdash \pi(\mathtt{Parallel\ P\ Q}) \\ \forall \mathtt{c}, \mathtt{P} \cdot \pi(\mathtt{P}) \vdash \pi(\mathtt{While\ c\ P})\end{array}}{\forall \mathtt{P} \cdot \pi(\mathtt{P})}$$

Since properties are specified as observers, for each language construct, one has to (i) compile the construct with empty subprograms; (ii) connect the input and output wires of each empty subcomponent to an observer circuit; (iii) connect the input and output wires of the outer block to an observer; (iv) universally quantifying over the outer circuit inputs, and the inner block outputs; and (v) prove that the conjunction of the inner observers implies the outer observer. Consider the case of sequential composition hand coded below:

```
let seqCase obs (s, (pSh, pF), (qSh, qF)) =
  let qS  = pF in
  let f   = qF in
  let sh  = or2(pSh, qSh) in
  let pOk = obs(pS, (pSh, pF)) in
  let qOk = obs(qS, (qSh, qF)) in
  let ok  = obs(s,  (sh,  f)) in
  and2(pOk, qOk) ==> ok
```

Similar cases would be written for each syntactic case, and verifying a compiler invariant then corresponds to model checking each of the cases. In practice we strengthen structural induction with temporal induction, assuming that the subcomponents have always worked up to now; and that the outer component always worked in the the past, modifying the last line of the cases to:

```
alwaysTillNow(and2(pOk, qOk))
    ==> alwaysInThePast ok
```

3.3. Automating Hardware Verification

Note that the design of these cases can become quite complex and error prone. Furthermore, when developing a hardware compiler, changes to the compiler code will have to be reflected faithfully in the syntactic cases. It is thus very desirable to be able to extract this information automatically from the hardware compiler code. Through the use of the host meta-language, it is actually possible to extract it, reducing user intervention, thus ensuring that the structural induction cases are automatically and accurately generated.

By quoting the hardware compiler we provide functions to access the structure of the actual compiler code for the different patterns. For instance, below is the code used to extract the observer for the sequential composition inductive case. Note that the function `indCase` takes the (quoted) hardware compiler, the invariant and the language construct as parameters — `Any` is a construct used as a place-holder for the constructor's parameters:

```
let verifySequentialCase obs =
  indCase compile obs (Sequential Any Any)
```

Any changes to the `compile` function are immediately, and automatically reflected in the inductive cases with no further need for modification.

3.4. Flash Invariants

To illustrate the use of this approach, we will show how Shade has been used to prove invariants of the Flash compiler relating the `start` and `finish` wires.

Invariant 1: No finish before start

The first and most basic invariant shown, states that if a Flash program terminates (it produces a high signal

over the `finish` wire), then the program must have been started at some point in time:

```
let flashInv01 (start, (sht, finish)) =
  finish ==> sometimeTillNow start;
```

Invariant 2: One start, one finish
The second invariant strengthens the previous relation to state that, if a program is started only once, then it cannot terminate more than once.

```
let flashInv02 (s, (shout, f)) =
  and2
  ( neverTillNow s ==> neverTillNow f
  , onceTillNow s ==>
      or2 (neverTillNow f, onceTillNow f)
  );
```

Invariant 3: One finish for each start
The third invariant shows that for each finish, the program must have a corresponding start. To encode this property, we define a circuit which returns whether a circuit (produced by the Flash compiler) is running:

```
let active (s, f) =
  loop active .
    let pre  = delay F active in
    let act1 = and2 (or2 (s, pre), inv f) in
    let act2 = and3 (prev, s, f) in
    or2 (act1, act2);
let running (s, f) =
  or2 (s, delay F (active (s, f)))
```

It is now possible to define the property:

```
let flashInv03Test (start, (shout, finish)) =
  finish ==> running (start, finish)
```

Model checking gives a counter example in which the circuit is started again while active. The compilation scheme of Flash assumes that this does not arise. To resolve this, we add an antecedant to the property as an environment constraint which states that no start signals will be forthcoming before termination. The following modified invariant model checks correctly:

```
let flashInv03 (start, (shout, finish)) =
  environment (start, finish) ==>
    finish ==> running (start, finish)
```

4. Conclusions

In this paper we have shown how the use of a meta-language as a host language for an embedded HDL can aid manipulation and analysis of embedded programs. Clearly, one has to ensure that the increased complexity of using a meta-language is counter-balanced by the gain in expressivity. In our approach, the hardware designer using Shade need not be aware of, or use the meta-programming features of *reFL*^{ect}, which are hidden inside Shade. The only exception to this design principle is the need to quote a hardware compiler before analysis.

The primary gains in the use of meta-programming within Shade are marking and manipulation of circuit blocks, and the analysis of circuit generators. In this paper we have explored the use of Shade to prove invariant properties of a simple Flash compiler, which we are currently extending to prove the correctness of a simplified Esterel compiler, and to use the block marking mechanism to generate automatic placement schemas for compiled Esterel programs.

References

[1] P. Bjesse, K. L. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 1998.

[2] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Computer-Aided Verification 2002*, LNCS 2404, 2002.

[3] K. Claessen and G. J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02*, 2002.

[4] J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

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

[6] T. Melham and J. O'Leary. A functional HDL in reFLect. In *Designing Correct Circuits '06*, 2006.

[7] J. O'Donnell. Embedding a hardware description language in template haskell. In *Domain-Specific Program Generation*, LNCS 3016, 2004.

[8] G. J. Pace and K. L. Claessen. Verifying hardware compilers. In *Computer Science Annual Workshop 2005 (CSAW'05)*. University of Malta, Sept. 2005.

[9] G. J. Pace and C. Tabone. Accessing circuit generators in embedded HDLs. In *Designing Correct Circuits '08*, 2008.

[10] C.-J. H. Seger, R. B. Jones, J. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. 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.

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

[12] Xilinx. *Xilinx Constraints Guide*, 9.1i edition.

View publication stats