Meta-Functional Languages for Hardware Design and Verification

Gordon J. Pace and Christian Tabone
University of Malta
gordon.pace@um.edu.mt, christian.tabone@um.edu.mt

***Abstract*—General purpose functional languages have been widely used as host languages for the embedding of domain specific languages, especially for hardware description languages. The embedding approach provides various abstraction techniques, enabling the description of generators for whole families of circuits, in particular parameterised regular circuits. The two-stage language setting that is achieved by means of embedding, provides a means to reason about the generated circuits as data objects within the host language. Nonetheless, these circuit objects lack information about their generators, and about the manner in which these where generated. In this paper, we explore the use of a meta-programming language to extend the embedding approach thus enabling us to access the underlying circuit generators, and not just the circuits themselves. We show the applicability of this approach by using circuit generator analysis techniques to extract information from a hardware compiler to enable verification, through the use of model-checking, of compiler invariants. The main contribution of this paper is to show how automatic verification of whole *families* of circuits can be used in an embedded language setting to verify hardware compiler invariants.**

***Keywords*-hardware synthesis; correct compilation; embedded languages.**

I. Introduction

As circuit size and complexity grows, the need for higher abstraction in hardware description languages is increasing. One approach is to have a high level language which is used to describe circuit generators — running programs in such a language results in concrete instances of the circuit (or families of circuits). However, designing and developing a new language for a specific domain presents various challenges. Not only does one need to identify the basic underlying domain-specific constructs, but if the language will be used for writing substantial programs, one has to enhance the language with other programming features, such as module definition and structures to handle loops, conditionals and composition. Furthermore, one has to define a syntax, and write a suite of tools for the language — parsers, compilers, interpreters, etc — before it can be used. One alternative technique that has been explored in the literature is that of embedding the domain-specific language inside a general purpose language, borrowing its syntax, tools and most of the programming operators. The embedded language is usually developed simply as a library in the host language, thus effectively inheriting all its features and infrastructure.

The use of embedded languages to build hardware description languages has long been explored [13]. It has been argued, and shown, that for the exploration of circuit design (especially at the gate level), and such an approach gives access to very effective abstraction techniques. Programs are written in the host language, which when executed, generate the circuit object itself. Despite the advantages gained from this two-stage language environment, the approach still does not give direct access to the circuit generators themselves — restricting hardware designers to performing analysis and apply transformation only at the gate level. Unless one can inspect the code generating the circuits at the programming level itself, a higher level analysis is impossible. In this paper, we address this issue by exploring the design of an embedded hardware description language, Shade, within a *reflective* language *reFL^{ect}* — enabling us to view and process the circuit generators directly. We show that by following this approach we are able to prove invariants of whole families of circuits, and we illustrate this by proving a number of properties of a hardware compiler for the Esterel language.

The paper is organised as follows — Section II discusses the current state of the art in the area, followed by Section III which explains the underlying embedding of Shade in *reFL^{ect}*, Section IV shows how hardware compilers can be described in this framework, Section V shows how an Esterel hardware compiler has been proved to be correct, and finally we conclude in Section VI.

II. State-of-the-Art

The functional programming paradigm has long been paired with the design and verification of circuits, typically by developing a hardware description language embedded in functional languages like Haskell. Lava [3], Hydra [9] and Hawk [6] are such hardware description languages, and are evidence of the fact that functional languages are not only ideal host languages for the embedding of any domain specific language, but accommodate the semantics of circuit descriptions in conjunction with the required abstractions with striking similarities to functional language features [13]. These embeddings enable various abstraction techniques which enable the description of generators for generic circuits, such as parametrised circuits and connection patterns. By delaying the evaluation of circuit descriptions and by having access to the abstract syntax tree of the expression, one is able to traverse this structure and output additional semantic interpretations. The advantage is that the different semantic interpretations operate on the same instance of the quoted expression. However, this needs to be done in two separate stages, first to compose the structure, and then to interpret the structure.

To enable analysis of circuits at higher levels of abstraction, block definitions are used in the language Wired [1] — which, however, pays the price of having to use combinators for the composition of circuits.

The main weakness of these approaches is that the information about generators is lost at the programming level. One solution to this dilemma is to use a reflective language to host a hardware description language. A number of such approaches have been explored. Of particular note is Melham *et al.*'s work in *reFL^{ect}* [8], in which they explore the use of reflection to mark subcircuits but at the cost of forcing hardware designers to interact with the meta-programming features directly. Even paying this price, their approach does not enable direct reasoning about circuit generators, which arguably is a desirable goal. Our approach is closely related to theirs, except that we structure the language in such a manner to enable processing the generators directly — thus enabling us to prove properties of whole families of circuits.

III. Embedding a HDL in *reFL^{ect}*

reFL^{ect} [5] is a strongly-typed functional language with meta-programming capabilities. *reFL^{ect}* was developed as part of the Forte tool [12]; a hardware verification system used by Intel. *reFL^{ect}* provides quotation and antiquotation constructs, allowing the composition and decomposition of unevaluated expressions, defined in terms of the *reFL^{ect}* language itself. These meta-programming constructs allow a form of reflection within a typed functional paradigm setting, enabling direct access to the structure of object programs. This is made possible by giving access to the internal representation of the abstract 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. Antiquotation constructs can be used in conjunction with the pattern matching mechanism to compose or decompose object programs, permitting the developer to modify or transform the quoted expression at runtime before evaluation.

Shade is a HDL we have developed, as an embedding in *reFL^{ect}*. Circuits are strongly typed, but are internally stored as quoted *reFL^{ect}* terms. In a language without reflection one usually transforms descriptions into data objects by means of a deep embedding. Through the use of the meta-programming features in *reFL^{ect}*, we use quoted shallow embedded descriptions which still allow us access to the circuit structure. The conservation of an unevaluated expression of a circuit definition provides the actual structural description that is required, which can still be interpreted directly to obtain an output, thus achieving circuit simulation. Unevaluated terms thus become the primary type of embedded programs which, in our case, contain circuit descriptions with the potential to evaluate to any signal representation.

Primitive gates ensure that the signals are of the correct structure and type, whilst decomposing the structure within the type term into the appropriate input signals. These signals or sub-expressions are hence used to compose the required expression. Note that the signals are unquoted using the ` command, and then re-quoted using the {| |} parentheses.

```
inv :: bool sig -> bool sig
let inv  (Signal {| `a |}) = Signal {| NOT `a |};

and2 :: (bool, bool) sig -> bool sig
let and2 (Signal {| (`a, `b) |}) = ...
```

Internally, the primitive gates are composed by quoted versions of their boolean operator counterparts, using antiquotations to deal with quotations in their parameters. However, from the end user perspective, who sees only the signature of these functions, all use of meta-programming features is hidden away. When defining larger circuits no reference to the meta-programming features is made.

Other primitive gates are defined using functions similar to the above, which can be presented to the end user to be used for other circuit descriptions. The constant expressions *high* and *low* represent the constantly high, and constantly low signals respectively, and the *delay* gate (parametrised by a boolean value giving the initial value of the output stream) produces a stream of values identical to the input except that it is delayed by one clock cycle.

To create loops in a circuit, Shade provides a fix-point operator, illustrated in the following example defining a register based on a multiplexer:

```
let mux (s, (a,b)) =
  or2 (and2 (inv s, a), and2 (s, b));

let setRegister (set, new_value) =
  loop now . let old = delay low now in
                  mux (set, (old, new_value));
```

Note that the reuse of user defined circuit components is identical to the use of the primitive components. Another two examples, which will be used later on in this paper, are the circuits `sometimes` (and `always`), which given an input, output a high signal if the input was true sometime (always) in the past up to, and including, the current point in time:

```
let sometimes x = loop ok . or2  (x, delay low ok)
let always    x = loop ok . and2 (x, delay high ok)
```

Other similar circuits, such as `never` (the input has never been true up to and including now) and `once` (the input has been true exactly once in the past up to and including now) can be similarly defined.

The embedding of a HDL gives the ability to provide multiple interpretations to the same circuit description [13]. Shade provides various interpretations for the described circuits, such as simulation by traversing the circuit structure using the meta-programming characteristics found in *reFL^{ect}* and performing 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 netlists.

Shade also supports circuit verification though the use of external model checkers. Instead of embedding a property language, we follow an observer-based approach, in which properties are also described as circuits which take the inputs and outputs of the circuit to be verified and return the result as a single boolean output. Hence, for a circuit to satisfy a property, the observer circuit has to output a constant high value. Although this limits the verifiable class of properties to safety properties, this approach avoids the need of an additional embedded language. Currently Shade is connected to the SMV model checker [7].

For instance, consider a property to verify that if both inputs of a multiplexer are equal, then no matter what the value of the sector wire is, the output is equal to the common input values. 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)
```

Combining this with the multiplexer circuit, and passing the output of this observer to create a model using a circuit interpretation which generates the input of a model checker, enables the property to be verified automatically.

As an illustrative example we show the use of Shade to define parallel prefix circuits, in particular Sklasky networks. The implementation in Shade of more complex prefix circuits can be found in [11].

Parallel prefix networks are circuits which given (i) a number of n input wires $i_0, i_1, \dots i_{n-1}$; and (ii) an associative binary operator $\oplus$, output n wires carrying the values i_0, $i_0 \oplus i_1$, $i_0 \oplus (i_1 \oplus i_2)$, etc. A well-known prefix circuit generator is the Sklansky network — performing the parallel prefix operation by dividing the input bus into two recursively. The binary operator is applied to the last bit of the first half over the entire second half of the bus. In implementing the Sklansky prefix network circuit (see Figure 1), we focus on how the marking of blocks is handled within such descriptions. The recursive definition for the Sklansky network is given as the auxiliary function `skl'` marking blocks as the recursive description unfolds. Note the use of `makeBlock` to delimit the recursive calls in the circuit generator.

```
letrec skl n op inp =
  let skl' 1 op inps = inps
  /\  skl' n op inps =
         val (lst,rst) = splitSignalBus n inps in
         let ls2       = skl (busLength lst) op lst in
         let rs2       = skl (busLength rst) op rst in
         let carry     = lastSignal ls2 in
         let apply r   = op (carry, r) in
         ls2  @  map apply rs2 in
  makeBlock (skl' n op) inp;
```

The circuit generator corresponds to an infinite family of circuits — one for each width of the bus. Although we can use Shade to access model checkers to verify particular instances of Sklansky networks, it is not straightforward to verify properties of the whole family of circuits. In the rest of the paper we discuss how we can provide access to the circuit generators to be able to reason about such families of circuits, rather than just individual instances.

IV. Hardware Compilers

The characteristics of embedded languages provide ways to advance to higher levels 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

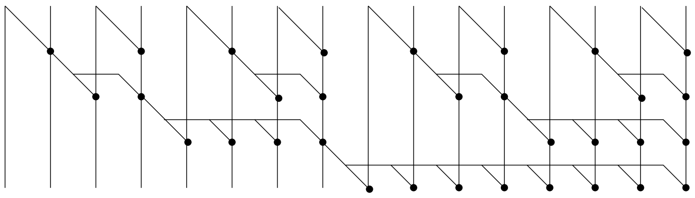
Fig. 1. A Sklansky parallel prefix circuit

host language. However, in such approaches the abstraction layers that are achieved still lead to a structural description of the circuit. An alternative approach which is increasingly being used is that of automatic hardware synthesis, or compilation; from a high level algorithmic description directly into a structural description.

Pace and Claessen [4] 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 is described. The compilation procedure corresponds to a circuit parametrised by the data object representing the language constructs.

An ongoing issue with hardware compilation is that the compilation procedure should ideally be verified to be correct. In practice this can be a long and tedious process. Developing various high level DSLs to solve a problem is increasingly becoming a common trend, which mean that hardware verification has to be done more frequently, sometimes by the same hardware designer. Pace and Claessen [10] 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, using a functional language such as Haskell, with no meta-programming capabilities, transforming the compiler description into the verification framework has to be performed by hand, even if it follows a uniform pattern. Thus, the major disadvantage with this approach is that the transformed description might not match exactly with 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, we can allow the designer to write a domain specific hardware compiler, and verify properties without the need to rewrite a transformation function. We achieve this by using the meta-programming features of *reFL*^{ect}, to automatically transform the compiler into an appropriate generator capable to construct observer models that can be interpreted by a finite state model checker.

A. Compiling Flash

We illustrate the process by looking at the embedding of a hardware compiler in Shade — using the Flash language from [4], which is a basic language with imperative programming constructs. Programs in Flash are simply instances of a datatype in *reFLect*:

```
lettype Flash = Skip | Shout  | Delay
  | IfThenElse (bool sig) Flash Flash
  | Seq Flash Flash  | Par Flash Flash
  | While (bool sig) Flash;
```

Note that Flash has the standard imperative language features, such as sequential composition and conditional, but it 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. The instruction `Shout` is the only instruction used to set the output wire to high. 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 [4]. The hardware compilation schemes for Flash are given in Figure 2. In Shade, the constructs designs can be implemented directly using pattern matching over the datatype, and calling the compile function recursively over the subprograms. The following is the code to handle two of the syntactic cases:

```
letrec compile Shout start =
  let shout  = start in
  let finish = start in
  (shout, finish)
/\     compile (Seq 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);
```

B. Compiler Invariants

A compiler should conform to its specification — for example, circuits produced by the Flash compiler should should never terminate unless started some time before. Another property is that if a program is started once, then this should only terminate once. Consequently, we can specify that a program generates a termination signal for each time it is started.

One way to verify whether such properties, or invariants, are satisfied by the hardware compiler is to use formal model checking [10]. To prove the correctness of an invariant, structural induction is applied over the language constructs, where each construct is proved to satisfy the given property by assuming that this is also satisfied by the subprograms, thus proving that any compiled program satisfies the property. When considering Flash, one can prove an invariant π over a program using structural induction:

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

To perform the above formal reasoning on circuits, the properties need to be encoded as observer circuits and appropriately attached to the compiler circuit. Therefore, 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, which says that for a given observer circuit `obs`, the circuit generated by the sequential composition of two circuits satisfying the observer will itself satisfy the observer:

```
let seq 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.

C. Temporal Induction

In practice, for most properties the approach does not work. The structural induction cases we are attempting to prove state that if the inner compiled blocks are working well now, then so is the outer block. If the inner blocks break the invariant for a period of time, but then satisfy it again later on, in this approach we expect that the outer block starts satisfying the invariant again. In practice we need a stronger notion — once the inner blocks have stopped working sometime in the past, the outer block is relieved of its obligation to satisfy the invariant. For instance, consider the following invariant which 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, (shout, finish)) =
  finish ==> sometimesInThePast start;
```

Using the naïve version of structural induction shown above, the model checker identifies a counter example for the sequential case in which the values of the start, finish wires and the output of the invariant observer for the outer block, and the two inner blocks of the sequential composition of two programs. Note that the first block finished without starting in the first time unit, but then proceeded to work correctly in the second time unit. This induced the second block to produce a finish signal in the second time unit, thus finishing the outer block in the second time unit (when both inner blocks satisfy the invariant) without ever having received a start signal:

start	fin	inv	start1	fin2	inv1	start2	fin2	inv2
0	0	1	0	1	0	1	0	1
0	1	0	0	0	0	0	1	1

The solution to such a problem is either to strengthen the invariant or by adding temporal induction into the verification methodology as given below:

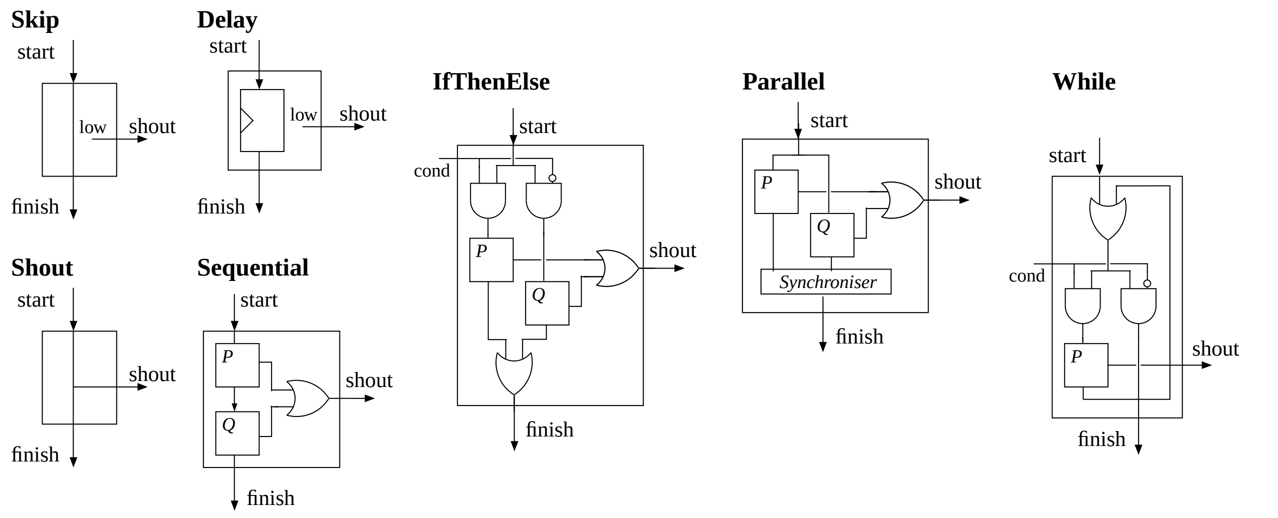

Fig. 2. Hardware designs of Flash

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

D. 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 one can access the structure of the actual compiler code to identify the different cases — by performing structural induction, the compiler description is broken into the individual alternative cases. For each of the cases the circuit is transformed using the type as the observer, and an appropriate function is composed as the result. Meta-programming is essential in order to automate the necessary transformations, since this enables inspection of the function describing the hardware compiler. This approach has been used to prove several compiler invariants. For instance, the invariant `flashInv01` can now be verified automatically.

The strength of this approach is that when changing the compiler code, the inductive cases need not be recoded to match the new compiler code, ensuring that the inductive cases and the compiler code match and thus that we are really verifying properties of the actual compiler we have written.

V. Correct Compilation of Esterel Into Hardware

Esterel [2] is a synchronous programming language with characteristics that enables the programming of concurrent systems. Concurrent constructs enable different sections of the same program to function in parallel, yet in synchrony with each other. This is achieved by enabling communication through the broadcasting of signals. Esterel is used to program reactive systems, such as real-time controllers, communication protocols and system drivers. Apart from simulation tools, Esterel compilers can translate programs into C code and hardware description languages, such as VHDL or Verilog.

The Esterel language is similar to the Flash language we presented earlier, but with a more intricate semantics to handle *schizophrenia* [2], which arises when a restarted loop terminates immediately. The compilation process is similar to that of Flash, but adds an additional finish wire.

```
compile program start =
  let ...
  in  (emit, (finish1, finish2))
```

To appreciate the intricacies of the language, consider the following Esterel program:

```
let prog w = While high
  (Par (IfThenElse w Delay Skip, Delay));
```

Consider the situation when the program is started with a high signal on `w`. Once the loop starts it triggers the fork-join construct, terminating one clock cycle later. In particular consider the finish wire on the output of the conditional case, which is high in the second cycle. Upon termination of the fork-join block, the loop is started again. Note that now if `w` is low, the conditional terminates immediately, overlapping the finish signal at the same time unit as the previously produced one. Since outputting high on the same wire over the same clock cycle has no noticeable effect, the second finish signal is lost to the synchroniser of the fork-join construct, which proceeds to wait indefinitely till the first branch produces another high signal. To avoid the overflow on the finish wire, the second finish appears on the second finish wire.

The constructive semantics for Esterel solve this problem by duplicating the logic related to such termination wires, thus each circuit would contain multiple termination wires depending on the number of possible occurrences. An indepth study of the schizophrenia problem is given in the circuit translations of the constructive semantics of Esterel [2]. Although the solution is well known, ensuring correctness of the compilation is not straightforward due to the intricate compilation. Using structural induction with automatically induced cases, we proved invariants of Esterel compilation.

- The finish wires work correctly: This property ensures that the finish wire encoding works correctly in that the use of the finish wires can never produce the combination low, high:
```
let esterelInvariant1 ... = f2 ==> f1;
```
- No start, no finish: Another sanity check for the compiler is that an Esterel program may never terminate unless explicitly started:
```
let esterelInvariant2 ... =
  never go ==> inv (or2 fs)
```
- Single start, single finish: If only a single start is ever given, the circuit may not output on the second finish wire, and may at most, output only once on the first finish wire. The following observer uses the `once` circuit which outputs high as long as the input has been high at most once in the past:
```
let esterelInvariant3 ... = once go ==>
  and2 (never f2, or2 (never f1, once f1))
```
- One finish for each start: Each finish must have a corresponding start, as long as the environment disallows a program to be started unless it has previously finished (encoded in the observer `usedWell`).
```
let esterelInvariant4 (go, (e, (f1, f2))) =
  let wasRunning = ... in
  always (usedWell (go, (e, (f1, f2)))) ==>
    and2(f1 ==> or2  (go, wasRunning)
        ,f2 ==> and2 (go, wasRunning))
```
- The second finish wire is never high twice in succession: As long as the environment disallows a program to be started unless it has previously finished, it will never be the case of having two successive high signals on the second finish wire.
```
let esterelInvariant5 (go, (e, (f1, f2))) =
  always (usedWell (go, (e, (f1, f2)))) ==>
     (f2 ==> delay T (inv f2))
```
- A third finish wire is redundant: Although adding a second wire seems a reasonable solution to the problem, it may be unclear why a third wire is not necessary. One way of showing that such a wire would be redundant is by extending the Esterel hardware compiler to have three finish wires, and proving that the third finish wire is constantly low:
```
let esterelInvariant6 ... = always (
  usedWell (go, (e, (f1, f2, f3))))   ==> inv f3)
```

In this manner, by means of model-checking techniques, we have proved that the control path of compiled Esterel programs maintains certain compiler invariants, thus increasing our confidence in the compilation process.

VI. Concluding Discussion

In this paper, we have built on these results by enabling placement combinators to be added to the block markings, without disrupting the functional style of the circuit descriptions. Furthermore, we have shown how meta-programming features can be used to automatically generate and extract verification models from the circuit generators of hardware compilers. Reflection enables us to provide a framework in which user intervention is minimised, thus ensuring that changes to the hardware compiler reflect faithfully the observer circuits required for the structural induction reasoning.

An advantage of our approach to other related ones, is that the hardware designer using Shade need not be aware of meta-programming features, which are kept hidden inside Shade. The only exception to this design principle is the need to quote a hardware compiler before analysis.

The use of model checking for structural reasoning about families of systems has been used under various guises in different domains. Our application of the technique, automatically extracting the inductive cases from the descriptions enhances the use embedded languages for hardware design support. 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 automatically extract structural induction cases for a hardware compiler, to enable the model checking of control path invariants. We are currently working on extending these results for the analysis of the data path in such languages, which poses new challenges, since the size of the output may grow as the output wires increase.

References

[1] Emil Axelsson, Koen Linström Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *Proc. of Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *LNCS*, 2005.

[2] G. Berry. The constructive semantics of Pure Esterel. Available from `http://www-sop.inria.fr/esterel.org/filesv5_92/`, 1999.

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

[4] K. Claessen and G. J. Pace. An embedded language framework for hardware compilation. In *the proceedings of Designing Correct Circuits 2002*, 2002.

[5] 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.

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

[7] Kenneth L. McMillan. *Symbolic Model Checking: An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1992.

[8] Tom Melham and John O'Leary. A functional HDL in reFLect. In *the proceedings of Designing Correct Circuits 2006*, 2006.

[9] J. O'Donnell. Overview of hydra: a concurrent language for synchronous digital circuit design. *Int. J. of Information*, pages 249–264, 2006.

[10] G.J. Pace and K. Linström Claessen. Verifying hardware compilers. In *Computer Science Annual Workshop 2005*. University of Malta, 2005.

[11] Gordon J. Pace and Christian Tabone. Access to circuit generators in embedded hdls. In *the proceedings of Designing Correct Circuits 2008*, 2008.

[12] C.-J. 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), 2005.

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