Meta-Functional Embedded Languages for Verification of Parametrised Hardware Descriptions

Gordon J. Pace, Christian Tabone
University of Malta

Abstract

In the literature, one finds various uses of the use of functional programming languages, combined with embedded language techniques for the design and description of circuits. In this paper we explore the use of a meta-programming language to extend this approach to have access to information about 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. This enables automatic verification of whole *families* of circuits, an approach which we apply to verify control path sanity of an Esterel hardware compiler.

1 Introduction

The functional 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 [BCSS98], Hydra [O'D06] and Hawk [LLC99] 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 [She05]. These embeddings enable various abstraction techniques which enable the description of generators for generic circuits, such as parametrised circuits and connection patterns. However, 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 from performing certain operations within the embedded language itself. For instance, whenever a circuit generator is invoked, the hierarchical structure of the circuit descriptions is lost completely, and unless explicitly annotated, the resulting circuits lack information about their generators

which can be used for placement and other non-functional analysis. Furthermore, not having access to the generators restricts us from applying formal reasoning on the circuit generators. For example, the verification of a hardware compiler as advocated in [PC05], may involve the description of a separate translation function to perform all the necessary logical changes to produce an appropriate observer circuit for the compiler implementation itself, and despite the close relation between the translation function and the hardware compiler, the respective generators are subject to possible inconsistencies, simply because the applied reasoning and manipulations are done manually by the hardware designer.

In earlier work, we have argued that a functional language with meta-programming constructs such as *reFLect* [GMO06], can be used to embed a hardware description language capable of maintaining hierarchical block markings of subcomponents within structural circuit descriptions [PT08]. In this paper, we build on these results by enabling placement combinators, similar to those found in Wired [ACS05] and DualEval [BW97], to be added to the block markings, without disrupting the functional style of the circuit descriptions. Furthermore, we propose the use of the meta-programming features of *reFLect* to automatically generate and extract verification models from the circuit generators of hardware compilers. Quoting the hardware compilers provides direct access to the circuit generators, enabling us to apply reasoning on the actual circuit generators. Hence, by means of the meta-constructs in conjunction with pattern matching, transformations are applied over the circuit generator in order to produce observer circuits suitable for verifying hardware compiler properties by structural induction. 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. To further illustrate the proposed approach, a number of invariant properties are presented for a subset of the Esterel hardware compiler [Ber99].

2 Embedding a HDL in *reFLect*

Typically, when embedding a DSL, a deep embedding is required since one would want not only to describe programs, but allow the possibility to give them different interpretations as may be necessary, 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 [GMO06] is a strongly-typed functional language with meta-programming capabilities. *reFLect* was developed as part of the Forte tool [SJO$^+$05]; a hardware verification system used by Intel. *reFLect* provides quotation and antiquotation constructs, allowing the composition and decomposition of uneval-

uated 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 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 are available which 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.

2.1 Shade

Shade is a HDL we have developed, as an embedding in *reFLect*. Circuits are strongly typed, but are internally stored as quoted *reFLect* terms. In a language without reflection a deep embedding of a language one usually transforms descriptions into data objects. Through the use of the reflection features in *reFLect*, 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 also 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 structure of signals. Phantom types are used to keep track of the types of the quoted expression, thus still achieving strong typing, and enabling type checking over quoted expressions.

```
lettype *a signal = Signal term;
```

The 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.

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


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

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) produces a stream of values identical to the input except that it is delayed by one clock cycle. The boolean parameter is used as the initial value of the output stream.

```
high, low :: bool sig
delay :: bool -> bool sig -> bool sig
```

Note that 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 perspective of the end user, who sees only the signature of these functions, all use of meta-programming features is hidden away.

When defining larger circuits, these primitive functions are used in a functional approach, with no reference to the meta-programming features. 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 [LLC99], 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 inn
  val (a, b)  = unzipp ab inn
  or2 (zipp (and2 (zipp (inv s, a)),
             and2 (zipp (s, b))));
```

To create loops in a circuit, Shade provides a fix-point operator illustrated in the following circuit:

```
let setRegister s_n =
  val (set, new) = unzipp s_n inn
  loop now . let old = delay low now inn
    mux (zipp (set, zipp (old, new)));
```

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:

[1] For simplicity, most of the examples given in this paper do not include the zipping and unzipping functions

```
let sometimes x = loop ok . or2  (zipp(x, delay F ok))
let always    x = loop ok . and2 (zipp(x, delay T 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.

2.2 Circuit Interpretations

One particular aspect of having a deep embedding of a HDL is the ability to provide multiple interpretations to the same circuit description [She05]. Shade provides various interpretations for the described circuits. Internally a circuit description is simply a quoted function denoting both the structural details and the actual functionality, therefore simulation can be achieved by unquoting this function and applying appropriate input values. Other interpretations involve traversing the circuit structure using the meta-programming characteristics found in *reFLect* 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 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 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 outputs 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 [McM92].

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)
```

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

3 Hardware Compilers

The characteristics of embedded languages provides 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 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 [CP02] 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 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 practise 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 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 [PC05]. However, using a functional language such as Haskell, 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. Thus, the major 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, 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 *reFLect*, to automatically transform the compiler into an appropriate generator capable to construct observer models that can be interpreted by a finite state model checker.

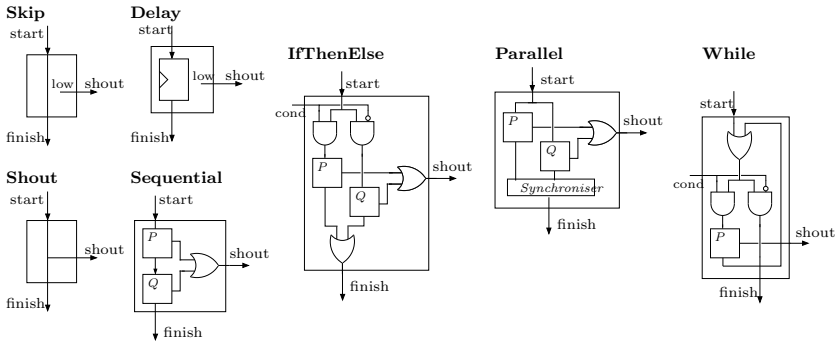

Figure 1: Hardware designs of Flash

3.1 Compiling Flash

We illustrate the process by looking at the embedding of a hardware compiler in Shade — using the Flash language from [CP02], 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
  | 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 [CP02]. The hardware compilation schemes for Flash are given in figure 1. In Shade, the constructs designs can be implemented directly using pattern matching over the datatype, and calling the compile function recursively over the subprograms. 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

The development process of a hardware compiler, just like any other hardware or software system, would typically begin from a set of specifications or requirements. Hence, once the design is completed and the compiler is implemented, one would want to conduct a series of tests to ensure that the system is performing the correct functionalities as defined in the specifications. For example, a possible requirement for the Flash compiler would be that a program should never terminate unless this has been started some time before. Another could state 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 [PC05]. 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. For instance, for Flash one can prove an invariant π over a program using structural induction as shown below:

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

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:

```
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.

3.3 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 weaker 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 flashInv02 (start, (shout, finish)) = finish ==> sometimes start;
```

Using the naive version of structural induction shown above, the model checker identifies a counter example for the sequential case, as shown in the table below, in which we show 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	0	0
finish	0	1
inv	1	0
start1	0	0
finish1	1	0
inv1	0	0
start2	1	0
finish2	0	1
inv2	1	1

The solution to such a problem is either to strengthen the invariant to ensure that once broken, it remains broken forever, or by adding a form of temporal induction in the verification methodology by assuming that the subcircuits have always satisfied the given property up to the current time. We prefer to go for the latter solution to get the verification rule given below:

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

3.4 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:

```
let flashCompiler =
  let metaFlash  = {|
    letrec compile_flash (Skip, start) =
      let shout  = low in
      let finish = start in
      (shout, finish)
   /\ compile_flash (Sequential f1 f2, start) =
      val (f1Shout, f1Finish) = compile_flash (f1, start) in
      val (f2Shout, f2Finish) = compile_flash (f2, f1Finish) in
```

```
    let shout               = or2 (f1Shout, f2Shout) in
    (shout, f2Finish)
  /\ ...
  in compile_flash |}
in metaFlash;
```

Using the quoted compiler, we are able to access the structure of the actual generator — by performing structural induction, the compiler description is destructed into the individual compiler alternative cases. For each of the cases the respective circuit is transformed using the type as the observer — by applying pattern matching over the circuit structure whilst modifying the circuit such that the inputs to a recursive call are quantified, and these together with the respective outputs are transformed and redirected, such that the observer circuit is applied. Finally, the list of terms are composed together as alternate cases, and the quoted resulting function is typecasted (using phantom types which are omitted in the example), and an appropriate function is composed as the result. Meta-programming is essential in order to automate the necessary transformations, since this enables pattern matching over the function describing the hardware compilers.

This approach has been used to prove several compiler invariants. For instance, consider a property for Flash which states that if a program has terminated, then this implies that the program must have been started some time before. Below is the observer circuit for this invariant:

```
let flashInv01 (start, (shout, finish)) = finish ==> sometimeInThePast start;
```

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.

4 Correct Compilation of Esterel Into Hardware

Esterel [Ber99] 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 so called *schizophrenia* [Ber99], which

arises when a restarted loop terminates immediately. The compilation process is similar to that of Flash, but adds an additional finish wire.

```
compile :: Esterel -> bool sig -> (bool sig, (bool sig, bool sig))
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 (Parallel (IfThenElse w Delay Skip, Delay));
```

Consider the situation when the program is started when `w` is high. The loop starts, trigerring 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 now 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 in-depth study of the schizophrenia problem is given in the circuit translations of the constructive semantics of Esterel [Ber99]. Although the solution is well known, ensuring correctness of the compilation is not straightforward due to the intricate compilation. For instance, unless one works out the details, it is not all that clear that two finish wires suffice.

Using structural induction with automatically induced cases, we have proved various invariants of the compilation of Esterel.

The finish wires work correctly: This property ensures that the finish wire encoding works correctly. Note that the use of the finish wires is assumed to produce `(low, low)` (in the case of no finishes), `(high, low)` (in the case of one finish) or `(high, high)` (in the case of two finishes). We ensure that the combination `(low, high)` can never occur:

```
let esterelInvariant1 (go, (e, (f1, f2))) = 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 (go, (e, fs)) = 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 (go, (e, (f1, f2))) =
  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`). The observer uses some code to calculate (on the basis of the circuit interface) whether it is was running one clock cycle ago.

```
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, there will never appear 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 is unclear why a third (and a fourth, fifth, ...) 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 (go, (e, (f1, f2, f3))) =
  always (usedWell (go, (e, (f1, f2, f3)))) ==> inv f3
```

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

5 Conclusions

In this paper we have shown how the use of a meta-language as a host language for an embedded hardware description language 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 automatically extract structural induction cases for a hardware compiler, to enable the model checking of invariants. The approach has been applied to a subset of the Esterel language, for which we have shown that the control path satisfies a number of expected 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

[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.

[Ber99] Gérard Berry. The constructive semantics of Pure Esterel. Unfinised draft, 1999.

[BW97] 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.

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

[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.

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

[McM92] Kenneth L. McMillan. *Symbolic Model Checking: An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1992.

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

[PC05] Gordon J. Pace and Koen Linström Claessen. Verifying hardware compilers. In *Computer Science Annual Workshop 2005 (CSAW'05).* University of Malta, September 2005.

[PT08] Gordon J. Pace and Christian Tabone. Accessing circuit generators in embedded hdls. In *Designing Correct Circuits '08, Budapest, Hungary*, March 2008.

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

[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.