Verifying Hardware Compilers

Gordon J. Pace[1] and Koen Claessen[2]

[1] University of Malta
[2] Chalmers University, Gothenburg, Sweden

Abstract. The use of hardware compilers to generate complex circuits from a high-level description is becoming more and more prevalent in a variety of application areas. However, this introduces further risks as the compilation process may introduce errors in otherwise correct high-level descriptions of circuits. In this paper, we present techniques to enable the automatic verification of hardware compilers through the use of finite-state model checkers. We illustrate the use of these techniques on a simple regular expression hardware compiler and discuss how these techniques can be further developed and used on more complex hardware-description languages.

1 Introduction

The size, and level of complexity of hardware has increased dramatically over these past years. This has led to the acceptance of high-level hardware synthesis — allowing the compilation of program-like descriptions into hardware circuits [10]. As in the case of software compilers, correctness of synthesis tools is crucial.

Hardware description languages embedded in general purpose languages have been given proposed and used with various languages and areas of application [4, 1, 2, 7]. In [3], we proposed a framework in which different hardware synthesis languages can be combined and compiled together within the framework of a single embedded hardware description language, Lava [4]. The high-level synthesis languages provide a number of complex composition operators, the interaction of which can be difficult to understand and ensure the correctness of. The verification of the synthesis procedures proved to be tedious, and in some cases very difficult to demonstrate. When combining languages, this proved to be even more difficult since each language has its own underlying compilation invariants which need to be satisfied for the compilation to be correct.

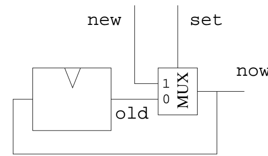
Lava is linked to a number of model-checking tools, which one can use to verify properties via the use of synchronous observers. We propose a technique using which we can verify the correctness of our compilers using finite-state model checkers to verify the compilation techniques. Compositional compilation techniques are usually verified using structural induction over the language constructs, the individual cases of which usually turn out to be of a finite nature.

2 Circuit Descriptions in Lava

Circuit descriptions in Lava correspond to function definitions in Haskell. The Lava library provides primitive hardware components such as gates, multiplexors and delay components. We give a short introduction to Lava by example.

Here is an example of a description of a register. It contains a multiplexer, `mux`, and a delay component, `delay`. The delay component holds the state of the register and is initialised to `low`.

```
setRegister (set, new) = now
  where
    old = delay low now
    now = mux (set, (old, new))
```



Note that setRegister is declared as a circuit with two inputs and one output. Note also that definitions of outputs (now) and possible local wires (old) are given in the where-part of the declaration.

After we have made a circuit description, we can simulate the circuit in Lava as a normal Haskell function. We can also generate VHDL or EDIF describing the circuit. It is possible to apply circuit transformations such as retiming, and to perform circuit analyses such as performance and timing analysis. Lava is connected to a number of formal verification tools, so we can also automatically prove properties about the circuits.

Should we want to decompose the multiplexor into more basic logic gates, one could define in terms of negation (inv), disjunction (<|>) and conjunction (<&>) gates:

```
mux (set, (case0, case1)) =
    (case0 <&> inv set) <|> (case1 <&> set)
```

2.1 Verification of Circuit Properties

Lava is connected through a number of model-checking tools which allow the verification of properties of circuits. To avoid introducing yet another formalism for property specification, An observer based approach as advocated in [5] is used to specify safety properties.

Given a circuit C, the property is specified using a separate circuit, called the *observer* reading the inputs and outputs of C, and outputting a single bit.

The circuit is then passed onto the model-checker to ensure that it outputs a constant high value.

For example, to check that the value in a register does not change if set is low, we can use the following observer:

```
checkRegister (set, new) = ok
  where
    current = setRegister (set, new)
    ok  = inv set ==> (current <==> new)
```

Note that ==> and <==> denote boolean implication and equivalence respectively. To check that it always holds using external model-checkers from within the Lava environment:

```
Lava> verify checkRegister
Proving: ... Valid.
```

Which allows us to conclude that $\forall \texttt{set}, \texttt{new} \cdot \texttt{checkRegister (set, new)}$.

2.2 Generic and Parametrized Circuit Definitions

We can use the one bit register to create an n-bit register array, by putting n registers together. In Lava, inputs which can be arbitrarily wide are represented by means of lists. A generic circuit, working for any number of inputs, can then be defined by recursion over the structure of this list.

```
setRegisterArray (set, [])       = []
setRegisterArray (set, new:news) = val:vals
  where
    val  = setRegister (set, new)
    vals = setRegisterArray (set, news)
```

Note how we use pattern matching to distinguish the cases when the list is empty (`[]`) and non-empty (`x:xs`, where `x` is the first element in the list, and `xs` the rest).

Circuit descriptions can also be parametrized. For example, to create a circuit with n delay components in series, we introduce n as a parameter to the description.

```
delayN 0 inp = inp
delayN n inp = out
  where
    inp' = delay low inp
    out  = delayN (n-1) inp'
```

Again, we use pattern matching and recursion to define the circuit. Note that the parameter `n` is *static*, meaning that it has to be known when we want to synthesise the circuit.

A parameter to a circuit does not have to be a number. For example, we can express circuit descriptions which take other circuits as parameters. We call these parametrized circuits *connection patterns*. Other examples of parameters include truth tables, decision trees and state machine descriptions. In this paper, we will talk about circuit descriptions which take behavioural hardware descriptions, or *programs*, as parameters.

2.3 Behavioural Descriptions as Objects

In order to parametrize the circuit definitions with behavioural descriptions, we have to embed a behavioural description language in Lava. We do this by declaring a Haskell datatype representing the syntax of the behavioural language. To illustrate the concepts with a small language, we will use simplified regular expressions without empty strings[3]. The syntax of regular expressions is expressed as a Haskell datatype:

```
data RegExp = Input Sig
            | Plus RegExp
            | RegExp :+: RegExp
            | RegExp :>: RegExp
```

The data objects belonging to this type are interpreted as regular expressions with, for example, $a(b+c)^+$ being expressed as:

[3] This constraint can be relaxed, but it allows us to illustrate the concepts presented in this paper more effectively.

```
Input a :>: Plus (Input a :+: Input c)
```

Note that the variables a, b and c are of type Sig — they are *signals* provided by the programmer of the regular expression. They can either be outputs from another existing circuit, or be taken as extra parameters to the definition of a particular regular expression. We interpret the signal a being high as the character 'a' being present in the input.

Since regular expressions are now simply data objects, we can generate these expressions using Haskell programs. Thus, for example, we can define a power function for regular expressions:

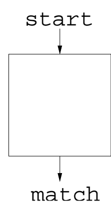
```
power 1 e = e
power n e = e :>: power (n-1) e
```

Similarly, regular expressions can be manipulated and modified. For example, a simple rewriting simplification can be defined as follows:

```
simplify (Plus e :>: Plus e) =
  let e' = simplify e
  in  e' :>: Plus e'
simplify (Plus (Plus e))     =
  simplify (Plus e)
...
```

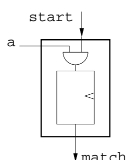
3 Compiling Regular Expressions into Hardware Circuits

Following the approach presented in [8], it is quite easy to generate a circuit which accept only input strings which are admitted by a given regular expression. The circuits we generate will have one input start and one output match: when start is set to high, the circuit will start sampling the signals and set match to high when the sequence of signals from a received start signal until just before the current time is included in the language represented by the regular expression.



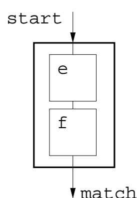
The type of the circuit is thus:
`type Circuit = Signal Bool -> Signal Bool`

The compilation of a regular expression is a function from regular expressions to circuits:
`compile :: RE -> Circuit`



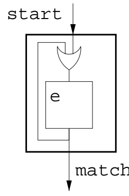
Signal input
The regular expression `Input a` is matched if, and only if the signal a is high when the circuit is started.

```
compile (Input a) s = delay low (s <&> a)
```



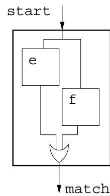
Sequential composition
The regular expression `e :>: f` must start accepting expression e, and upon matching it, start trying to match expression f.

```
compile (e :>: f) start = match_f
  where
    match_e = compile e start
    match_f = compile f match_e
```



Loops
The circuit accepting regular expression `Plus e` is very similar to that accepting `e`, but it is restarted every time the inputs match `e`.

```
compile (Plus e) start = match
  where
    start' = start <|> match
    match = compile e start'
```



Non-deterministic choice
The inputs match regular expression `e :+: f` exactly when they match expression `e` or `f`.

```
compile (e :+: f) start = match
  where
    match_e = compile e start
    match_f = compile f start
    match   = match_e <|> match_f
```

4 Model Checking Compilers

As we have seen in the regular expression example, we will be using embedded language techniques to represent programs as instances of a datatype:

```
data Program =
      Variable := Expression
    | Program :> Program       -- Sequential composition
    | ...
```

In general, a synthesis procedure is nothing but a function from a program to a circuit:

```
compile :: Program -> (Circuit_Ins -> Circuit_Outs)
```

To reason about individual programs is no different from reasoning about circuits. For example the following function generates an observer to verify whether a given program satisfies a given property (with some constraints on the environment):

```
observer (environment, property) program =
  \ ins -> let outs = compile ins program
           in  environment (ins, outs) ==> property (ins, outs)
```

Similarly, we can compare programs by generating an appropriate observer:

```
p === q =
  \ ins -> let outs_p = compile ins p
               outs_q = compile ins q
           in  outs_p <==> outs_q
```

However, we can do more than this. We have identified a number of levels in which we can use model checking to reason about the synthesis procedure itself.

– We would like to be able to quantify over programs, to be able to write properties like:

```
forAllPrograms $ \ e ->
  forAllPrograms $ \ f ->
    forAllPrograms $ \ g ->
      e :> (f :> g) === (e :> f) :> g
```

  To do this, we add a new component to the `Program` datatype, which represents a circuit:

```
data Program =
     ...
   | Circuit (Circuit_Ins -> Circuit_Outs)
```

  We can now quantify over the outputs of the circuit to obtain an observer for the quantification:

```
forAllPrograms fprogram =
  \ (outs_c, ins) -> fprogram (Circuit (\ _ -> outs_c)) ins
```

  Using this approach, we can verify various properties of our regular expression compilation function:

```
plusCommutative ins =
  forAllPrograms $ \e ->
    forAllPrograms $ \f ->
      e :+: f ===  f :+: e

Lava> verify plusCommutative
Proving: ... Valid.
```

  Using this technique, we managed to prove that the compilation procedure satisfies standard axioms of regular expressions, hence effectively verifying the compiler.
– However, most interesting language properties can only be proved using structural induction. It is usually impossible to prove properties of a program unless one assumes that the subcircuits satisfy these properties. This can be encoded inside the synthesis procedure, by adding an extra output which confirms whether the sub-components of the compiled circuit satisfy the invariant:

```
compile2 (p :+: q) invariant ins = (ok, outs)
  where
    (ok_p, outs_p) = compile2 p invariant ins
    (ok_q, outs_q) = compile2 q invariant ins

    outs = combine (outs_p, outs_q)
    inv  = invariant (ins, outs)

    ok   = ok_p <&> ok_q ==> inv

compile2 (Circuit c) invariant ins = (invariant (ins, outs), outs)
  where
    outs = c ins
```

To prove the `invariant` inductive case for an operator, it would suffice to prove the following observer holds:

```
proveAlt invariant (outs_c1, outs_c2, ins) = ok
  where
    c1 = Circuit (\_ -> outs_c1)
    c2 = Circuit (\_ -> outs_c2)
    (ok, _) = compile2 (c1 :+: c2) invariant ins
```

Similar inductive step observers can be constructed for the other language operators:

```
proveStructuralInduction invariant (o1, o2, ins) = ok
  where
    ok  = ok1 <&> ok2 <&> ok3 <&> ok4

    ok1 = proveSeq   invariant (o1, o2, ins) -- :>:
    ok2 = proveAlt   invariant (o1, o2, ins) -- :+:
    ok3 = provePlus  invariant (o1, ins)     -- Plus
    ok4 = proveInput invariant ins          -- Input
```

One such property we can prove only through the use of structural induction is:

```
noEmptyString (start, match) = start ==> inv match

Lava> verify (proveStructuralInduction  noEmptyString)
Proving: ... Valid.
```

This confirms all the cases of the structural induction, allowing us to confirm its truth for all regular expressions.

- One weakness with the above proof, is that if the sub-circuits ‘break’ for some time but then resume to work correctly, the top-level circuit is expected to resume correctly. This is a strong property which compilation procedures which encode some form of state usually fail to satisfy. To strengthen induction to deal with this adequately we need to add temporal induction – assuming that the sub-components always worked correctly, the top-level component works as expected:

```
compile3 (p :+: q) invariant ins = (inv, outs)
  where
    (ok_p, outs_p) = compile3 p invariant ins
    (ok_q, outs_q) = compile3 q invariant ins

    outs = combine (outs_p, outs_q)
    inv  = invariant (ins, outs)

    ok = always (ok_p <&> ok_q) ==> inv
```

Where:

```
always s = ok
  where
    ok = delay high (s <&> ok)
```

- Most compiler invariants assume that the environment satisfies certain conditions. At first sight, this could be expressed as `environment ==> property`. However, we run into the same problem that we had with sub-circuits breaking and then starting to work once again. The solution is to add an environment condition to the condition verified:

```
compile4 (p :+: q) ins = (ok, outs)
  where
    (ok_p, outs_p) = compile4 p ins
    (ok_q, outs_q) = compile4 q ins

    outs  = combine (outs_p, outs_q)
    inv   = invariant (ins, outs)
    env   = environment (ins, outs)
    env_p = environment (ins, outs_p)
    env_q = environment (ins, outs_q)

    ok = always (env <&> ok_p <&> ok_q)
            ==> (inv <&> env_p <&> env_q)
```

5 Conclusions

In this paper, we have outlined how finite state model-checkers can be used to verify properties of hardware compilers, and we have illustrated the use of these techniques on a simple regular expression compiler. Through the (external) use of structural induction, we decompose the general property into a number of finite state cases. This works on hardware compilers thanks to the fact that the size of the data path can be determined at compile time. In languages where this is not possible, our techniques clearly fail to work as presented. All related work we have identified use pen-and-pencil proofs (eg [6]) or theorem provers (eg [9]) to verify the correctness of the synthesis procedure. Our approach, although narrower in scope, has the distinct advantage of being a (relatively speaking) 'push-button' approach to compiler verification.

We plan to apply our techniques on more complex languages. In particular, we would like to investigate the correctness of Esterel compilation and standard Verilog and VHDL synthesis techniques. Furthermore, the use of these techniques on other generic and parametrised circuits can also prove to be effective.

References

1. P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. 2003.
2. Paul Caspi and Marc Pouzet. Synchronous Kahn networks. ACM Sigplan International Conference on Functional Programming, 1996.
3. Koen Claessen and Gordon J. Pace. An embedded language framework for hardware compilation. In Designing Correct Circuits '02, Grenoble, France, 2002.
4. Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verication of a sorter core. In CHARME. Springer, 2001.
5. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. Proceedings of the IEEE, 79(9):1305-1320, September 1991
6. Jonathan P. Bowen Jifeng He, Ian Page. Towards a provably correct hardware implementation of occam. CHARME 1993, 1993.
7. John Launchbury, Jerey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within haskell. ACM SIGPLAN international conference on Functional programming, 1999.
8. Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In 23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96), number 1099 in LNCS. Springer, 1996.
9. Klaus Schneider. A veried hardware synthesis of esterel programs. In DIPES '00: Proceedings of the IFIP WG10.3/WG10.4/WG10.5 International Workshop on Distributed and Parallel Embedded Systems, pages 205-214, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.
10. Niklaus Wirth. Hardware compilation: Translating programs into circuits. Computer, 31(6):25-31, 1998.