Extracting formal models from synthesised Verilog *(Category B submission)*

Mike Gordon, David Greaves, Konrad Slind

University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
U.K.
mjcg@cl.cam.ac.uk
http://www.cl.cam.ac.uk/~mjcg

Abstract. The verification of implemented algorithms can be decomposed into (i) showing properties of an abstract representation of the algorithm and (ii) showing the abstract representation is correctly implemented. In this paper we discuss an approach to (ii) based on automatically extracting formal logical models from the output of the CV3 Verilog compiler. CV3 output is processed by inference using HOL, but the use of HOL is completely hidden from the user. The model extraction is packaged as a Unix tool that reads a Verilog source file and creates a HOL theory file. Command line options are used to select the kind of model produced and the form in which it is represented. This work illustrates the approach being explored by the Prosper project [9] in which user level tools make use of an internal proof engine.

1 Introduction

Formal verification is often applied to algorithms represented abstractly. For example, Harrison [5] verifies the correctness of floating point algorithms expressed in a simple imperative programming language semantically embedded in HOL, and Paulson [8] verifies properties of cryptographic protocols modelled directly in Isabelle/HOL[1].

There are several approaches to ensuring that abstract representations of algorithms are correctly realised by concrete implementations, for example:

- formally refine the algorithm to implementable code or hardware;
- use a verified compiler;
- use an unverified compiler, but check for each run that the output is equivalent to the input.

The approach described here is similar to the last of these: we automatically extract logical models from the output of an unverified (and rapidly evolving) Verilog compiler. These models are in higher order logic and are suitable for further analysis by theorem proving or model checking. Application scenarios include post synthesis checking of properties of the implementation and showing that the synthesised implemention matches previously formulated abstract models.

Industrial tools exists to check the equivalence between HDL input and the results of synthesis, but as far as we know these use ad hoc semantics of the input HDL and are mainly intended to ensure that if synthesisable HDL is verified by simulation, then this verification will also apply to the results of synthesis.

The rest of the paper proceeds as follows: first an overview of the relevant aspects of CV3 is given, next the modelling in HOL of the result of compiling Verilog is discussed, then the various options for processing by HOL of the synthesis output are described.

[1] Isabelle/HOL is Isabelle's instantiation to simply typed higher order logic [7].

2 CV3

CV3 is a tool written by David Greaves that reads a Verilog source file and generates output in a variety of formats. It is implemented in a version of Standard ML[2] and "supports compilation of nearly the whole Verilog language but has some bugs".[3]

The output of CV3 is determined by a technology library. The library used here is called `cv2.100` and consists of various combinational components and a positive edge-triggered Dtype. The example in this paper only uses inverters (`INV`), 2-input exclusive-or gates (`XOR2`) and Dtype flip-flops (`DFF`). These components have simulation models written in Verilog. The models of the combinational components all have small delays[4] to avoid asynchronous (zero-delay) loops.

```
module INV(o, i);
  output o;
  input i;
  assign #2 o = ~i;
endmodule

module XOR2(o, i1,i2);
  input i1, i2;
  output o;
  assign #3 o = i1 ^ i2;
endmodule
```

The Dtype has a more complex model that includes additional variables (`last_d`, `last_clk`) and tasks for generating and displaying simulation output.

```
module DFF(q, d, clk, ce, ar, spare);
  output q;
  reg q;
  initial q = 0;
  input clk;          // Clock (positive edge triggered)
  input d;            // Data input
  input ce;           // Clock enable
  input ar;           // Asynchronous reset
  input spare;

  integer last_d, last_clk;

  always @(posedge clk or posedge ar)
        if (ar) q <= #10 0;
        else if (ce)
                begin
                if ($time - last_d < 5)
                  $display("Time %t,DFF %m violated set-up time",$time);
                last_clk = $time;
                q <= #10 (d & 1);
                end

  always @(d)
        begin
        last_d = $time;
        if ($time - last_clk < 4)
          $display("Time %t,DFF %m violated hold time",$time);
        end
endmodule
```

[2] CV3 is written in a version of ML implemented by David Greaves.

[3] `http://www.cl.cam.ac.uk/users/djg/localtools/oldindex.html`

[4] #n specifies a delay of n units of simulation time

One possible output from CV3 is an unflattened (i.e. module hiererarchy preserving) Verilog netlist (`vnl`), another output form is an ML datatype representing the unflattened netlist (`mlout`). The former is more readable, but the latter is used in the interface to HOL.

To illustrate CV3, suppose the file `COUNT2.cv` contains the following Verilog module definitions:

```
module CLK_DIV(clk,ce);
  input clk;
  output ce;
  reg ce;
  always @(posedge clk) ce = !ce;
endmodule

module COUNT2(clk,out);
  input clk;
  output [1:0] out;
  reg [1:0] out;
  wire ce;
  CLK_DIV M1(clk,ce);
  always @(posedge clk) if (ce) out = out+1;
endmodule
```

Executing the command

```
 cv3core cv2.100 -root CLK_DIV -vnl $PWD/COUNT2.cv -o CLK_DIV.vnl
```

will write the file `CLK_DIV.vnl` with the following Verilog netlist:[5]

```
module CLK_DIV (clk, ce);
  supply0 LGND; supply1 LVCC;
  input clk;
  output ce;
  wire I100;
  DFF  ce(ce, I100, clk, LVCC, LGND, LGND);
  INV  I100(I100, ce);
endmodule
```

Executing the command

```
 cv3core cv2.100 -root COUNT2 -vnl $PWD/COUNT2.cv -o COUNT2.vnl
```

will write the file `COUNT2.vnl` with the following Verilog netlist:

```
module COUNT2 (clk, out);
  supply0 LGND; supply1 LVCC;
  input clk;
  output [1:0] out;
  wire g102,I100;
  wire ce;
  CLK_DIV  M1(clk, ce);
  DFF  i1out103(out[1], g102, clk, ce, LGND, LGND);
  XOR2  g102(g102, out[1], out[0]);
  DFF  i0out101(out[0], I100, clk, ce, LGND, LGND);
  INV  I100(I100, out[0]);
endmodule
```

[5] Automatically generated comments have been removed and the format of the output Verilog has been made more compact.

3 Importing CV3 output into HOL

It is very easy to import the netlists output by CV3 into HOL, because CV3 can generate its output as an ML datatype. The standard representation of structure in HOL [3] associates predicates with components, variables with wires and then expresses the structure using conjunction and existential quantification. For example, when COUNT2 is imported into HOL the Verilog module declaration becomes the following definition.

```
|- COUNT2 CLK_DIV (clk,out_1,out_0) =
    ∃I100 g102 ce.
      CLK_DIV (clk,ce) ∧
      DFF (out_1,g102,clk,ce,LGND,LGND) ∧
      XOR2 (g102,out_1,out_0) ∧
      DFF (out_0,I100,clk,ce,LGND,LGND) ∧ INV (I100,out_0)
```

The components in the cv2.100 technology library (e.g. XOR2, DFF) are predefined with a user-selected semantics that is discussed below. Any module that has not been predefined (e.g. CLK_DIV) is made a paramenter.

The translation to a HOL-netlist is achieved by the command[6]

```
cv2hol COUNT2.cv COUNT2 COUNT2
```

The first argument is the Verilog source file (COUNT2.cv). The second argument (COUNT2) is the name of the Verilog module in the source file that is to be imported into HOL, and the final argument (COUNT2) is the name of the theory to be created. This call to cv2hol creates a theory COUNT2Theory (which is represented by two files: COUNT2Theory.sml and COUNT2Theory.sig) containing the definition above.

Often one wants to read in a sequence of modules and create a theory containing them all. If a module M1 is defined and then used in a subsequently defined module M2, then M1 will not be a parameter to M2. However, if M2 is defined first then M1 will be a parameter. The order in which modules are defined is specified by the order they are listed when cv2hol is invoked. For example

```
cv2hol COUNT2.cv CLK_DIV COUNT2 COUNT2
```

first reads CLK_DIV and then COUNT2 from the file COUNT2.cv and creates a theory COUNT2Theory containing

```
|- CLK_DIV (clk,ce) =
    ∃I100. DFF (ce,I100,clk,LVCC,LGND,LGND) ∧ INV (I100,ce)

|- COUNT2 (clk,out_1,out_0) =
    ∃I100 g102 ce.
      CLK_DIV (clk,ce) ∧
      DFF (out_1,g102,clk,ce,LGND,LGND) ∧
      XOR2 (g102,out_1,out_0) ∧
      DFF (out_0,I100,clk,ce,LGND,LGND) ∧ INV (I100,out_0)
```

Since CLK_DIV was defined when COUNT2 was processed, it is treated as a predefined constant and not made a parameter. To get CLK_DIV as a parameter to COUNT2 use

```
cv2hol COUNT2.cv COUNT2 CLK_DIV COUNT2
```

The general form of a command to create a netlist theory is

cv2hol <*source*> <*module*> ··· <*module*> <*theory*>

where <*source*> is a Verilog source file, each <*module*> is the name of a module declared in the source file and <*theory*> is the name of the theory to be created.

[6] See Section 8 for current status of implementation.

4 Extracting models from netlists

The meaning of a term like COUNT2(clk,out) defined by cv2hol depends on the meaning of the components INV, XOR2, DFF etc. This is determined by the parent theory that predefines these constants. Currently four theories are provided, each giving a different model of the components.

simTheory approximates the "golden" simulation semantics, complete with combinational delays;
edgeTheory gives all combinational components zero delay and DFF a unit delay on a rising edge;
tickTheory gives all combinational components zero delay, shrinks clock cycles to a single 'tick' and models DFF as a unit delay on a tick;
cycleTheory cycle-based semantics: clock lines are ignored and DFF is modelled as a pure unit delay.

The default theory is cycleTheory. The other three theories are selected by giving cv2hol the argument -sim, -edge or -tick before the Verilog source file.

4.1 The theory simTheory

Example definitions of the combinational components in simTheory are shown below. Note that the delays correspond to the Verilog simulation models of the components.

```
LVCC(t)           = T
LGND(t)           = F
INV(o,i)          = ∀t. o(t+2) = ¬(i t)
XOR2(o,i1,i2)     = ∀t. o(t+3) = i1 t xor i2 t
```

The HOL model of DFF in theory simTheory approximates the simulation model, though the waveform monitoring is ignored. A rising edge is defined by

```
rise clk t = ¬(clk t) ∧ clk(t + 1)
```

and then DFF is defined by

```
DFF(q, d, clk, ce, ar, spare) =
 (∀t. t<10 ⇒ (q t = F))
 ∧
 ∀t. if ((rise clk t) ∨ (rise ar t))
       then (if ar(t+1)
              then q(t+10) = F
              else if ce(t+1) then q(t+10) = d t
                              else q(t+10) = q t)
       else (q(t+10) = q t)
```

Although the HOL models in theory simTheory of the cv2.100 components use the same delay values as the Verilog simulation models, it is far from clear how the representation of behaviour in HOL corresponds to that generated by the Verilog simulation cycle. This is an important question, since one would like verification by simulation and formal verification to produce consistent results [4]. Attempts so far at reconciling simulation and formal verification semantics are at best rather preliminary (e.g. [2]).

The combinational delays in the Verilog models of the cv2.100 components are in practice mainly to ensure well behaved simulation, rather than to support timing analysis. The model supports the implementation of storage via combinational loops and the implementation of simulators is such that the delays do not get in the way of efficient modelling.

The HOL theory simTheory leads to rather messy formal models that are hard to analyse. In particular, additional state variables are needed to define transition relations (see Section 6). Thus the model simTheory is really only of academic interest.

A more tractable model is edgeTheory in which there are no combinational delays, but clock edges are still explicit, and transparent latches and gated and derived clocks can be represented.

4.2 The theory edgeTheory

The theory edgeTheory is obtained from simTheory by setting all combinational delays to zero, and giving DFF unit-delay.

```
LVCC(t)            = T
LGND(t)            = F
INV(o,i)           = ∀t. o t = ¬(i t)
XOR2(o,i1,i2)      = ∀t. o t = i1 t xor i2 t

DFF(q, d, clk, ce, ar, spare) =
 (q 0 = F)
 ∧
 ∀t. if ((rise clk t) ∨ (rise ar t))
       then (if ar(t+1)
               then q(t+1) = F
               else if ce(t+1) then q(t+1) = d t else q(t+1) = q t)
       else (q(t+1) = q t)
```

This model is appropriate when one wants to model transparent latches, or flip-flops triggered on rising and falling edges. If every register is clocked on the positive edge of a single clock line, then a simpler representation is obtained by merging the steps between a positive edge and the following negative edge into a single abstract 'tick'. Thus there is no sequence of times when the clock is high.

4.3 The theory tickTheory

The theory tickTheory is obtained from edgeTheory by regarding the clock as a sequence of abstract ticks: clk t = T means there's a tick at time t and clk t = T means no tick at time t. There is no distinction between positive and negative edges and no interval between successive edges of the same clock phase. tickTheory is a temporal abstraction [6] from from edgeTheory. As with edgeTheory all combinational delays are zero. The model of DFF is

```
DFF(q, d, clk, ce, ar, spare) =
 (q 0 = F)
 ∧
 ∀t. if clk t
       then (if ar t
               then q(t+1) = F
               else if ce t then q(t+1) = d t else q(t+1) = q t)
       else (q(t+1) = q t)
```

This model is appropriate when only one kind of edge is used to trigger flip-flops and there are no transparent latches, but gated or derived clocks need to be modelled. With tickTheory a clock cycle is atomic and is associated with a single time. With edgeTheory a clock cycle can take several unitis of time (e.g. between rising edges).

If every register is clocked on a single clock line, then a simpler representation is obtained by abstracting all signals to their values at successive ticks. This corresponds to a 'cycle-based' interpretation. The theory cycleTheory in the next section interprets the cv2.100 components at this abstraction.

4.4 The theory cycleTheory

If all registers have the same clock line, then the following simplified model of DFF can be used.

```
DFF(q, d, clk, ce, ar, spare) =
 (q 0 = F)
 ∧
 ∀t. if ar t
      then q(t+1) = F
      else if ce t then q(t+1) = d t else q(t+1) = q t
```

Note that the clock line `clk` is ignored[7] – time is modelling succesive cycles. There is thus a further temporal abstraction from the timescale used in theory `tickTheory`. If flip-flops are clocked by more than one clock then translation to HOL will give an incorrect model.

4.5 Selecting a parent theory

The component model to be used is specified as the first argument to `cv2hol`

cv2hol -<*model*> <*source*> <*module*> ··· <*module*> <*theory*>

where <*model*> is one of `sim`, `edge`, `tick` or `cycle`. If no model is specified, then `cycle` is assumed. The theory named <*theory*>`Theory` that is created will have <*model*>`Theory` as a parent.

5 Deriving equations from netlists

When `cv2hol` is invoked, the default is to create a theory just with the translated netlists. If the argument -eqn is given, then each translated module is unwound using the definitions of the `cv2.100` components and other modules in the source that are defined earlier. For example, invoking

```
cv2hol -sim -eqn COUNT2.cv CLK_DIV CLK_DIV
```

creates a theory `CLK_DIVTheory` that, as well as the HOL netlist of `CLK_DIV`, also contains the automatically proved theorem

```
|- CLK_DIV (clk,ce) =
     ∃I100.
       ∀t.
         ((t < 10 ⇒ ¬ce t) ∧
          (if ¬clk t ∧ clk (t + 1) then
             ce (t + 10) = I100 t
           else
             ce (t + 10) = ce t)) ∧
         (I100 (t + 2) = ¬ce t)
```

Invoking

```
cv2hol -edge -eqn COUNT2.cv CLK_DIV CLK_DIV
```

creates a theory containing

```
|- CLK_DIV (clk,ce) =
     ¬ce 0 ∧
     ∀t.
       (if ¬clk t ∧ clk (t + 1) then
          ce (t + 1) = ¬ce t
        else
          ce (t + 1) = ce t)
```

Invoking

```
cv2hol -tick -eqn COUNT2.cv CLK_DIV CLK_DIV
```

creates a theory containing

[7] Currently the clock line is retained as an input variable, but ignored. This is inefficient for model checking, so in the future the clock variable may be eliminated.

```
|- CLK_DIV (clk,ce) =
    ¬ce 0 ∧
    ∀t.
      (if clk t then ce (t + 1) = ¬ce t else ce (t + 1) = ce t)
```

and invoking

```
cv2hol -cycle -eqn CLK_DIV CLK_DIV
```

creates a theory containing

```
|- CLK_DIV (clk,ce) = ¬ce 0 ∧ ∀t. ce (t + 1) = ¬ce t
```

If several modules are specified and the `-eqn` argument given, then all the modules are unwound. For example, invoking

```
cv2hol -cycle -eqn COUNT2.cv CLK_DIV COUNT2 COUNT2
```

creates a theory `COUNT2Theory` containing the HOL netlists of `CLK_DIV` and `COUNT2` and the theorems

```
|- CLK_DIV (clk,ce) = ¬ce 0 ∧ ∀t. ce (t + 1) = ¬ce t

|- COUNT2 (clk,out_1,out_0) =
    ∃ce.
      ¬ce 0 ∧ ¬out_1 0 ∧ ¬out_0 0 ∧
      ∀t.
        (ce (t + 1) = ¬ce t) ∧
        (if ce t then
           out_1 (t + 1) = ¬(out_1 t = out_0 t)
         else
           out_1 (t + 1) = out_1 t) ∧
        (if ce t then
           out_0 (t + 1) = ¬out_0 t
         else
           out_0 (t + 1) = out_0 t)
```

6 Deriving state transition systems

The equations produced using `-eqn` are useful for theorem proving. For model checking, it is convenient to derive a state transition system. To support this `cv2hol` can automatically derive a predicate giving the initial state (which has all the variables initialised to `F`) and a transition relation $\mathcal{R}$ defined so that if s is the vector of boolean state variables and s' is the corresponding vector of primed variables then $\mathcal{R}(s, s')$ means that s' is a possible successor of s.

The state vector of the transition system for a module consists, in general, of a pair $(\mathtt{s}_1,\mathtt{s}_2)$ where $\mathtt{s}_1$ is a vector of the inputs and outputs of the module and $\mathtt{s}_2$ is a vector of the local variables. If there are no local variables, as in `CLK_DIV`, then the state vector is just $\mathtt{s}_1$. For `COUNT2` the vector $\mathtt{s}_1$ is `(clk,out_1,out_0)` and $\mathtt{s}_2$ is `ce`.

Invoking `cv2hol` with argument `-trans` generates definitions of <*module*>`Init` and <*module*>`Trans` for each module specified. For example, invoking

```
cv2hol -cycle -trans COUNT2.cv CLK_DIV COUNT2 COUNT2
```

puts the following definitions and theorems into `COUNT2Theory`

```
|- CLK_DIVInit(clk,ce) = ¬ce

|- CLK_DIVTrans((clk,ce),clk',ce') = (ce' = ¬ce)

|- COUNT2Init((clk,out_1,out_0),ce) = ¬ce ∧ ¬out_1 ∧ ¬out_0

|- COUNT2Trans(((clk,out_1,out_0),ce),(clk',out_1',out_0'),ce') =
    (ce' = ¬ce) ∧
    (if ce then out_1' = ¬(out_1 = out_0) else out_1' = out_1) ∧
    (if ce then out_0' = ¬out_0 else out_0' = out_0)
```

In addition to these definitions, for each module $\mathcal{M}$ a theorem is automatically proved of the form

$$\texttt{|- }\forall P.(\forall s_1\ s_2.\ \texttt{Reachable}\ \mathcal{M}\texttt{Trans}\ \mathcal{M}\texttt{Init}\ (s_1,s_2) \Rightarrow P\ s_1) \Rightarrow \forall v_1\cdots v_n.\ \mathcal{M}(v_1,\ \ldots,\ v_n) \Rightarrow \forall t.\ P(v_1\ t,\ \ldots,\ v_n\ t)$$

This shows that if P is true of all reachable states of the derived transition system then $P(v_1\ t, \ldots, v_n\ t)$ holds at each time t. This theorem is a bridge from model checking to theorem proving [1]. For example, invoking

```
cv2hol -cycle -trans COUNT2.cv CLK_DIV COUNT2 COUNT2
```

puts the following theorems into COUNT2Theory

```
|- ∀P.(∀s. Reachable CLK_DIVTrans CLK_DIVInit s ⇒ P s)
      ⇒
      ∀clk ce. CLK_DIV (clk,ce) ⇒ ∀t. P (clk t,ce t)

|- ∀P. (∀s1 s2. Reachable COUNT2Trans COUNT2Init (s1,s2) ⇒ P s1)
       ⇒
       ∀clk out_1 out_0.
         COUNT2(clk,out_1,out_0) ⇒ ∀t. P(clk t,out_1 t,out_0 t)
```

7 Discussion and future research

cv2hol packages an off-the-shelf hardware compiler (CV3) and a proof engine (HOL) into an easy-to-use turnkey semantics extractor for Verilog. The output can be loaded into other tools (including HOL) for further processing.

In the future it is hoped that the various models could be derived from a single model, ideally a representation in logic of the HDL simulation cycle. The current tool is a pragmatic compromise: the different models correspond to different definitions of the cv2.100 primitives and are not formally related. However, the implementation methodology of cv2hol does insure that if you trust the specified model, then the other derived representations (as selected by -eqn, -trans) are guaranteed to be logically consistent with it. This is a step towards ensuring the different representations needed for different purposes are consistent with each other.

The long term goal is to provide a platform supporting the easy scripting of bespoke checkers that automatically verify special purpose properties. cv2hol is an initial experiment and is part of Cambridge's contribution to the Prosper project [9].

8 Implementation status

Currently cv2hol is implemented as a Moscow ML standalone executable that parses the command line arguments, creates an ML script, and then loads the script into HOL to create the desired theory. The dynamically created script loads one of `simTheory`, `edgeTheory`, `tickTheory` or `cycleTheory` according to the argument given to `cv2hol`. Currently the `-eqn` and `-trans` options are not available if `-sim` (i.e. `simTheory`) is specified. It is possible that this implementation strategy might change in the future (e.g. to use `Holmake` and/or the Prosper tool integration mechanisms).

9 Acknowledgements

This work was also supported by ESPRIT Framework IV LTR 26241 project entitled *Proof and Specification Assisted Design Environments* (PROSPER), EPSRC grant GR/L35973 entitled *A Hardware Compilation Workbench* and EPSRC grant GR/L74262 entitled *A uniform semantics for Verilog and VHDL suitable for both simulation and formal verification.*

References

1. Michael J. C. Gordon. Reachability programming in HOL using BDDs. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 180–197. Springer-Verlag, 2000.
2. Mike Gordon. Synthesizable verilog: syntax and semantics. Draft available at `http://www.cl.cam.ac.uk/users/mjcg/Verilog/V/V.ps.gz`.
3. Mike Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, 1986.
4. Mike Gordon. The semantic challenge of Verilog HDL. In *Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 136–145. IEEE Computer Society Press, 1995.
5. John Harrison. Floating point verification in HOL Light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997. Available on the Web as `http://www.cl.cam.ac.uk/users/jrh/papers/tang.html`.
6. T. F. Melham, editor. *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1993.
7. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
8. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
9. See web page `http://www.dcs.gla.ac.uk/prosper/`.