Notes on the Representation of State Machines in Higher Order Logic

Mike Gordon
Computer Laboratory
University of Cambridge

January 6, 1999

1 Introduction

State machines (or automata) are an important component of hardware design. This note discusses their representation in higher order logic from the perspective of formal verification. Quite a lot of work already exists on representing machines in higher order logic, such as Loewenstein's theories of automata in HOL and the formulation of parts of Hopcroft and Ullman in Nuprl. The discussion here is intended to provide a framework for representing synchronous hardware designs in the HOL logic in a way that supports the application of both automatic tools (equivalence checkers, model checkers etc) and user guided proof.

2 Abstract state machines

State machines are a mathematical abstraction of synchronous digital systems. Such systems have a behaviour determined by their *state*, whose value is drawn from a *state space*. A state machine is assumed to remain in a given state emitting some *outputs* until a *state change event* occurs (usually a clock edge). When this happens a new state is entered which depends on the current state and current *inputs*.

2.1 Moore and Mealy machines

Machines are traditionally classified into *Moore machines* and *Mealy machines*. With Moore machines, the value being output depends only on the value of the current state. With Mealy machines, the output depends on both the current state and on the value being input.

Let *state* be the type of states (so the state space consists of all values of type *state*). Let *input* be the type of input values and *output* be the type of output values.

A Moore machine is represented by a triple $\langle s, \delta, \nu \rangle$, where $s : state$ is the an initial state, $\delta : (state \times input) \to state$ is the next-state function and $\nu : state \to output$ is the output function.

A Mealy machine is the same, except that it has a different type of output function $\nu : (state \times input) \to output$ (i.e. outputs can depend on the input as well as the state).

Note that if the state space of a Mealy machine has just one element, then the only non-trivial component of the machine is the output function, which in this case is essentially just a function from inputs to outputs. Thus Mealy machines include as a special case a representation of *combinational logic*. For pragmatic reasons (minimising state space size, avoiding asynchronous loops), hardware designers sometimes use Mealy machines and sometimes use Moore machine with separate combinational logic.

The kind of state machines just defined are *deterministic* in that a unique next state is determined by the current state and input. Non-deterministic machines are represented by having a set of initial states, replacing the next state function by a next state relation[1] $\delta : ((state \times input) \times state) \to bool$ and the output function by a relation[2] $\nu : ((state \times input) \times output) \to bool$.

2.2 Transition systems

State machines can be represented as *transition systems*. This representation is used for model checking.

A transition system is a pair $\langle I, R \rangle$, where I is a set of initial states and R is a relation between pairs of states. The interpretation of $R(s, s')$ – i.e. s is related to s' by R – is that s' is a possible successor to s.

[1] In higher order logic, a relation between types σ and τ can be represented as the characteristic function of the graph of the relation, i.e. as a function of type $(\sigma \times \tau) \to bool$, where *bool* is the type of Booleans consisting of the two truth-values T (*true*) and F (*false*).

[2] Machines with truly non-deterministic outputs don't seems to arise often, but the representation of outputs with a relation rather than a function is useful when encoding machines in logic.

With the transition system representation, the inputs and outputs are regarded as part of a more general kind of state. A non-deterministic Mealy machine $\mathcal{M} = \langle I_{\mathcal{M}}, \delta_{\mathcal{M}}, \nu_{\mathcal{M}} \rangle$ is represented by a transition system $\langle I_{\mathcal{M}}, R_{\mathcal{M}} \rangle$ where

$$R_{\mathcal{M}}((s,i,o),(s',i',o')) \;=\; \delta_{\mathcal{M}}(s,i,s') \;\wedge\; \nu_{\mathcal{M}}(s,i,o)$$

This is slightly subtle and needs some explanation, but first note that the state space of the transition system is $state_{\mathcal{M}} \times input_{\mathcal{M}} \times output_{\mathcal{M}}$, which has more components (namely inputs and outputs) than the state of the Mealy machine from which it was derived. There is potential for confusion between these two different notions of state space. In the definition of $R_{\mathcal{M}}$ note that the input and outputs in the successor state (i.e. i' and o') are not constrained. This reflects the idea that the input is determined by the environment and that the output is a 'combinational' function of the current state and – in the case of a Mealy machine – the inputs. The physical intuition of $R_{\mathcal{M}}((s,i,o),(s',i',o'))$ is that during stable or 'quiescent' periods of $\mathcal{M}$'s behaviour in which its state is s and input i then the output will be o. When a state change event occurs then the current state will become s' and the output will change to reflect the new state and input.

2.3 Traces

A transition system $\langle I, R \rangle$ determines a set of *traces*. A trace is an infinite sequence of states such that the first member of the sequence is in I and each member of the sequence is related to its successor by R.

If the state space is product, say $State_1 \times \cdots \times State_n$, then a trace is an infinite sequence of n-tuples:

$$\langle \langle s_1^0, \ldots, s_n^0 \rangle, \langle s_1^1, \ldots, s_n^1 \rangle, \;\; \ldots \;\; \rangle$$

It is sometimes convenient to consider instead the transpose of this, namely the n-tuple of infinite sequences:

$$\begin{array}{c} \langle \langle s_1^0, s_1^1, s_1^2, \ldots \rangle, \\ \langle s_2^0, s_2^1, s_2^2, \ldots \rangle, \\ \vdots \\ \langle s_n^0, s_n^1, s_n^2, \ldots \rangle \rangle \end{array}$$

Each infinite sequence in such an n-tuple is the trace of one component of the state.

3 Hardware description languages

To specify a machine for a particular task it is necessary to express the initial state and transition. For simple examples, this can be done using standard mathematical notation (e.g. set theory or higher order logic), but for complex hardware designs this is impractical for several reasons:

- Standard mathematical notation is not formal, so cannot be easily parsed and processed by CAD tools.
- Standard mathematics does not provide notation for hardware oriented operations such as manipulating bitstrings (words, bytes etc).
- Industrial scale machines are very large and need to be structured and parameterised in complex ways (modules, instances etc) requiring programming-like constructs.
- The abstract state machine realised by a hardware design is only one aspect of the design: its function. Other aspects include timing and electrical details.

For these (and other) reasons, machines are usually expressed using a *hardware description language* (HDL). Industry standard languages, like Verilog and VHDL, are designed primarily to support detailed hardware simulation. The function – i.e. abstract state machine – represented by an HDL text is hard to extract from the mass of other detail. Furthermore, there may be several different abstraction levels (RTL, behavioral etc) at which the abstract function can be viewed.

To enable pure functional behaviour to be expressed, a number of languages have been developed that provide a more mathematical way of expressing machines. These include model checker input languages (e.g. SMV) and synchronous languages (e.g. Esterel, Lustre). Such languages stand midway between commercial HDLs and abstract state machines: they enable complex designs to be specified in a structured way, yet have a direct mathematical interpretation.

4 Embedding semantics in logic

There are two main approaches to representing the semantics of HDLs in higher order logic: *deep embedding* and *shallow embedding*. With deep embedding a type, *syn* say, is defined inside the logic to represent HDL texts (values of type *syn* will essentially just be parse trees of texts). A type, *sem*

say, that represents the semantics is also defined, and then a semantic function, Meaning : $syn \rightarrow sem$ say, is defined, usually by primitive recursion over syn. With a shallow embedding there is no type syn or semantic function Meaning inside the logic. Instead a parser (e.g. written in ML) is used to translate HDL texts directly into terms of the logic.

Shallow embedding places less demands on the logic, but doesn't allow certain kinds of properties to be formulated. For example, with a deep embedding formulae of the form

$\forall x : syn.\ \mathsf{Meaning}(x) = \mathsf{Meaning}(\mathsf{Transform}(x))$

can be formulated. With a shallow embedding this cannot be expressed.

On the other hand, with a deep embedding the semantics must be expressible with a function inside the logic, and this semantic function must have a type. If different members of syn need a semantics represented by values of different types, it may not be possible to find a type for a semantic function (especially in simple type theory – in set theory or dependent type theory this is less of a potential problem). There is no corresponding problem with a shallow embedding, because the process of assigning meanings to texts does not have to encoded as a function inside the logic. A metalanguage program can easily compute differently typed terms for different HDL texts.

Shallow embedding tends to be more efficient because neither the type of program texts not the semantic functions need to be represented in logic. For example, defining a datatype in ML to represent Verilog parse trees is easy, but making a type definition inside HOL to represent such trees stresses the HOL system almost to its limits.

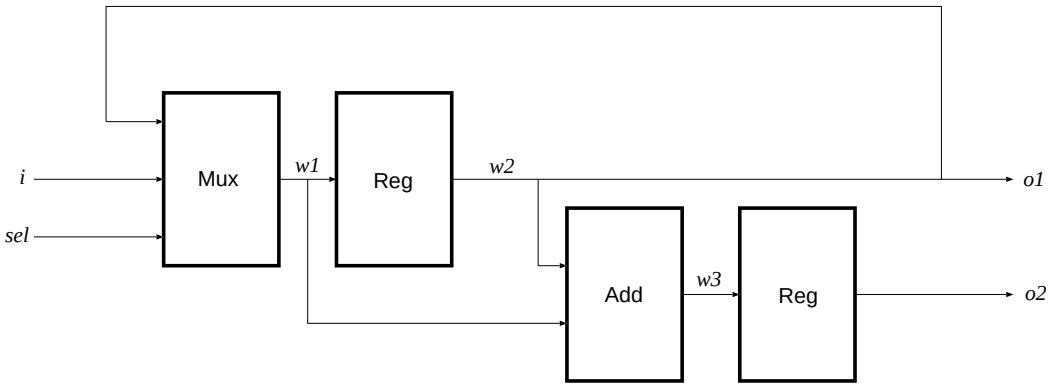
Thus deep embedding allows richer properties to be expressed, but shallow embedding allows a richer choice of semantics and is less computationally demanding.

5 Representing machines in logic

In order to define the semantics of an HDL in logic it is necessary to devise a way to represent machine behaviours as logical formulae.

For model checking is is necessary to extract the transition system (i.e. the model), which is then encoded in a compact form (e.g. as a BDD). The usual logical representation of a transition system is to use primed state variables to denote the successor state.

For example: consider the (arbitrary) example below.

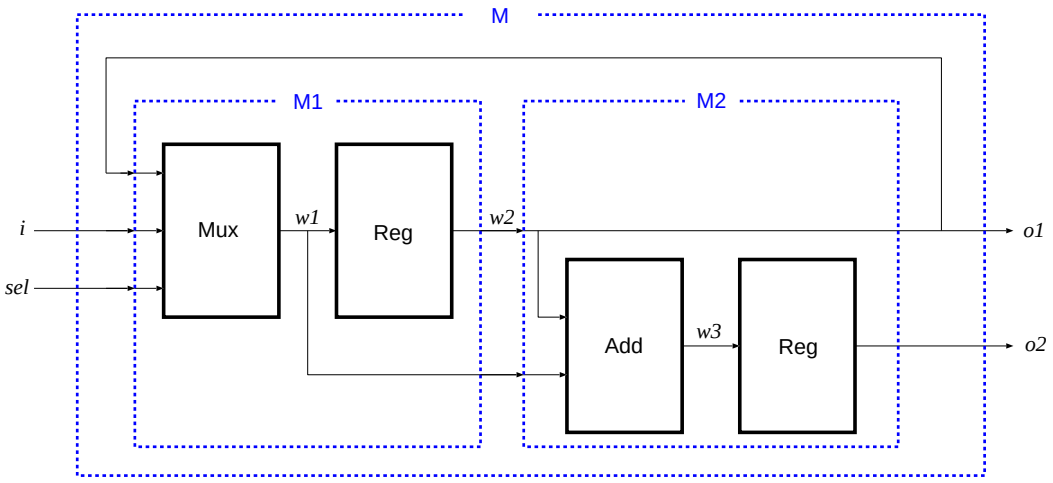


The state vector is $\langle i, sel, w1, w2, w3, o1, o2 \rangle$. If instances of the register Reg are modeled as unit delays (i.e. the output is the input in the preceding state) and Add and Mux are a combinational adder and multiplexer, respectively, then the transition relation can be represented as the formula:

$$(w1 = sel?i\!:\!o1) \wedge (w2' = w1) \wedge (w3 = w1 + w2) \wedge (o1 = w2) \wedge (o2' = w3)$$

A formula like this can be directly encoded as a BDD and used for model checking, however it is 'flat' and does not show any module hierarchy. Such structure is normally expressed in an HDL using a variety of linguistic devices for modularisation, module instantiation and interconnection, and localisation (hiding) of state and wires.

Consider the following hierarchical version of the diagram:



A simplified pseudo-Verilog specification of this is:

```
MODULE M(sel,i,o1,o2)
 INPUT sel,i;
 OUTPUT o1,o2;
 WIRE w1,w2;
  Mux(sel,i,o1,w1);
  Reg(w1,w2);
  M1(w1,w2,o1,o2)
END

MODULE M1(i1,i2,o1,o2)
 INPUT i1,i2;
 OUTPUT o1,o2;
 WIRE w;
  ASSIGN o1=i1;
  Add(i1,i2,w);
  Reg(w,o2);
END

MODULE Mux(sel,i1,i2,o)
 INPUT sel,i1,i2;
 OUTPUT o;
  ASSIGN o = sel ? i1 : 12;
END

MODULE Reg (i,o)
 INPUT i;
 OUTPUT o;
  ALWAYS @(posedge clock) o<=i;
END

MODULE Add(i1,i2,o)
 INPUT i1,i2;
 OUTPUT o;
 ASSIGN o = i1+i2;
END
```

A cycle-based semantics of this hierarchical structure (but not the localisation of wires) can be represented inside logic using the definitions below (which is an abstraction based on state transitions occurring on the positive edge of `clock`).

$$\texttt{DEFINE}\ \ \mathsf{M}((sel, i, w1, w2, w3, o1, o2), (sel', i', w1', w2', w3', o1', o2')) =$$
$$\quad \mathsf{M1}((sel, i, o2, w2), (sel', i', o2', w2'))\ \wedge$$
$$\quad \mathsf{M2}((w1, w2, w3, o1, o2), (w1', w2', w3', o1', o2'))$$

$$\texttt{DEFINE}\ \ \mathsf{M1}((sel, i1, i2, , o1, o2), (sel', i1', i2', , o1', o2')) =$$
$$\quad \mathsf{Mux}((sel, i1, i2, o1), (sel', i1', i2', o1'))\ \wedge$$
$$\quad \mathsf{Reg}((o1, o2), (o1', o2'))$$

$$\texttt{DEFINE}\ \ \mathsf{M2}((i1, i2, w, o1, o2), (i1', i2', w', o1', o2')) =$$
$$\quad \mathsf{Add}((i1, i2, w), (i1', i2', w'))\ \wedge$$
$$\quad \mathsf{Reg}((w, o1), (w', o1'))\ \wedge\ (o2 = i2)$$

$$\texttt{DEFINE}\ \ \mathsf{Mux}((sel, i1, i2, o), (sel', i1', i2', o')) =$$
$$\quad (o = sel?i1 : i2)$$

$$\texttt{DEFINE}\ \ \mathsf{Add}((i1, i2, o), (i1', i2', o')) =$$
$$\quad (o = i1 + i2)$$

$$\texttt{DEFINE}\ \ \mathsf{Reg}((i, o), (i', o')) =$$
$$\quad (o' = i)$$

This modularisation is over-the-top for such a simple example, but modular specifications scale much better than flat ones. For example, the BDDs for the whole machine can be built incrementally following the module structure.

The obvious redundancy of having both primed and unprimed variables could could be eliminated by using some syntactic conventions, such as $\langle x_1, \ldots, x_m \rangle$ for $((x_1, \ldots, x_m), (x'_1, \ldots, x'_m))$. Although this might work, experience (e.g. with Z) suggests that such hidden priming conventions can sweep under the carpet tricky proof issues, because the logical form of the specification becomes obscured (e.g. $\mathsf{Reg}\langle w, o1 \rangle$ has four free variables, not two).

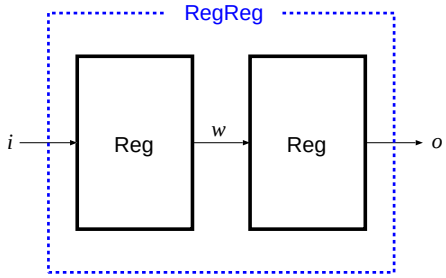
The localisation of variables is usually mimicked in logic by existential quantification. At first sight this might appear to work with the convention that the next state is represented by primed variables. Consider M2. The wire from Add to Reg (w in the definition of M2 which is instantiated to $w3$ in the diagram) can be hidden by existential quantification:

$$\texttt{DEFINE}\ \ \mathsf{M2}((i1, i2, o1, o2), (i1', i2', o1', o2')) =$$
$$\quad \exists w\ w'.\ \mathsf{Add}((i1, i2, w), (i1', i2', w'))\ \wedge$$
$$\qquad \mathsf{Reg}((w, o1), (w', o1'))\ \wedge\ (o2 = i2)$$

From this it follows that:

$$
\begin{aligned}
&\mathsf{M2}((i1, i2, o1, o2), (i1', i2', o1', o2')) \\
&\quad = \exists w\ w'.\ (w = i1 + i2)\ \wedge\ (o1' = w)\ \wedge\ (o2 = i2) \\
&\quad = \exists w.\ (w = i1 + i2)\ \wedge\ (o1' = w)\ \wedge\ (o2 = i2) \\
&\quad = (\exists w.\ (w = i1 + i2)\ \wedge\ (o1' = w))\ \wedge\ (o2 = i2) \\
&\quad = (\exists w.\ (o' = i1 + i2)\ \wedge\ (o1' = w))\ \wedge\ (o2 = i2) \\
&\quad = (o' = i1 + i2)\ \wedge\ (\exists w.\ (o1' = w))\ \wedge\ (o2 = i2) \\
&\quad = (o' = i1 + i2)\ \wedge\ \mathsf{T}\ \wedge\ (o2 = i2) \\
&\quad = (o1' = i1 + i2)\ \wedge\ (o2 = i2)
\end{aligned}
$$

which is what is wanted. However, consider now two registers in series:



This represented in logic by:

$$
\begin{aligned}
&\texttt{DEFINE}\ \mathsf{RegReg}((i, o), (i', o')) = \\
&\qquad \exists w\ w'.\ \mathsf{Reg}((i, w), (i', w'))\ \wedge\ \mathsf{Reg}((w, o), (w', o'))
\end{aligned}
$$

From this it follows that:

$$
\begin{aligned}
&\mathsf{RegReg}((i, o), (i', o')) \\
&\quad = \exists w\ w'.\ (w' = i)\ \wedge\ (o' = w) \\
&\quad = (\exists w'.\ w' = i)\ \wedge\ (\exists w.\ o' = w) \\
&\quad = \mathsf{T}\ \wedge\ \mathsf{T} \\
&\quad = \mathsf{T}
\end{aligned}
$$

This is clearly wrong! The problem is that the values of i and o don't determine the value stored in the first register (the one with input i and output w) at the current cycle, though the stored value becomes i on the next cycle. Since any value could be being stored, the value of o' could be any value too. Thus (i, o) does not constrain (i', o') – which is exactly what the calculation above verifies. The value output by the second register (the one with output o) is actually the value input to the first register two cycles earlier. This suggests the need for something like $o'' = i$, where o'' is another variable. This is a slippery slope – consideration of three registers in series would require o''' etc. Instead of having a cumbersome (and potentially

infinite) sequence of variables o, o', o'' etc. ranging over the values during all cycles, it is neater to have a single variable ranging over traces. Traces are naturally represented as functions from the type *num* of natural numbers to values. Operators like + and the conditional $-?-:-$ can be 'lifted' pointwise to operate on sequences, for example $f_1 + f_2 = \lambda t.f_1(t) + f_2(t)$ etc. Instead of the priming notation, an operator next is defined by $\mathsf{next}(f) = \lambda t.f(t+1)$. This also eliminates the duplication of variables due to priming. With this approach, two registers in series are modelled by:

$$\texttt{DEFINE}\ \ \mathsf{Reg}(i,o) \quad = \quad (\mathsf{next}(o) = i)$$

$$\texttt{DEFINE}\ \ \mathsf{RegReg}(i,o) \quad = \quad \exists w.\ \mathsf{Reg}(i,w)\ \wedge\ \mathsf{Reg}(w,o)$$

Now a correct result is obtained:

$$\begin{array}{l}\mathsf{RegReg}(i,o)\\ \quad = \ \exists w.\ (\mathsf{next}(w) = i)\ \wedge\ (\mathsf{next}(o) = w)\\ \quad = \ \exists w.\ (\mathsf{next}(\mathsf{next}(o)) = i)\ \wedge\ (\mathsf{next}(o) = w)\\ \quad = \ (\mathsf{next}(\mathsf{next}(o)) = i)\ \wedge\ (\exists w.\ \mathsf{next}(o) = w)\\ \quad = \ (\mathsf{next}(\mathsf{next}(o)) = i)\ \wedge\ \mathsf{T}\\ \quad = \ (\mathsf{next}(\mathsf{next}(o)) = i)\end{array}$$

Similarly with three registers in series, the relation $\mathsf{next}(\mathsf{next}(\mathsf{next}(o))) = i$ would follow, and so on.

With variables ranging over traces, the hierarchy M can now be defined, with both modularisation and localisation, as follows:

$$\begin{array}{l}\texttt{DEFINE}\\ \quad \mathsf{M}(sel,i,o1,o2) \quad = \quad \exists w1\ w2.\ \mathsf{M1}(sel,i,o2,w2)\ \wedge\ \mathsf{M2}(w1,w2,o1,o2)\end{array}$$

$$\begin{array}{l}\texttt{DEFINE}\\ \quad \mathsf{M1}(sel,i1,i2,,o1,o2) \quad = \quad \mathsf{Mux}(sel,i1,i2,o1)\ \wedge\ \mathsf{Reg}(o1,o2)\end{array}$$

$$\begin{array}{l}\texttt{DEFINE}\\ \quad \mathsf{M2}(i1,i2,o1,o2) \quad = \quad \exists w.\ \mathsf{Add}(i1,i2,w)\ \wedge\ \mathsf{Reg}(w,o1)\ \wedge\ (o2 = i2)\end{array}$$

$$\begin{array}{l}\texttt{DEFINE}\\ \quad \mathsf{Mux}(sel,i1,i2,o) \quad = \quad (o = sel?i1:i2)\end{array}$$

$$\begin{array}{l}\texttt{DEFINE}\\ \quad \mathsf{Add}(i1,i2,o) \quad = \quad (o = i1 + i2)\end{array}$$

$$\begin{array}{l}\texttt{DEFINE}\\ \quad \mathsf{Reg}(i,o) \quad = \quad (\mathsf{next}(o) = i)\end{array}$$

The pseudo-Verilog description given above can be compared with this purely logical representation. It is clear that a shallow embedding can easily take the former into the latter.

The transition system can be derived by purely logical manipulation:

$$\begin{array}{l}\mathsf{M}(sel, i, o1, o2) \;= \\ \quad (\mathsf{next}(o1) = (sel?i{:}o2) + o2) \;\wedge\; (\mathsf{next}(o2) = sel?i{:}o2)\end{array}$$

which is the transition system:

$$\langle sel,\ i,\ o1,\ o2\rangle \longrightarrow \langle sel,\ i,\ (sel?i{:}o2) + o2,\ sel?i{:}o2\rangle$$