Chapter 2

FSM-Hume is Finite State

Greg Michaelson,[1] Kevin Hammond[2] and Jocelyn Serot[3]

Abstract Hume is a domain-specific programming language targeting resource-bounded computations. It is based on generalised concurrent bounded automata, controlled by transitions characterised by pattern matching on inputs and recursive function generation of outputs. Here we discuss the design of FSM-Hume, a strict finite state subset of Hume, and suggest that it is indeed classically finite state.

2.1 INTRODUCTION

We would like to be able to prove automatically the correctness, equivalence, termination, space use and complexity of arbitrary programs but these properties are all undecidable for Turing-complete (TC) languages [1]. Some decidability may be achieved by restricting the types and constructs in a language. Languages based on primitive recursion, such as Turner's elementary strong functional programming [6] or Burstall's inductively defined functions [2], seem unwieldy and to lack clear programming methodologies. Languages based on finite state automata (FSA), such as Promela with the related Spin model checker [4], have proved much more successful, but of relatively limited application and with vast state spaces, constraining verification of substantial programs.

Hume [3] is based on a generalisation of standard FSA transition notation to encompass a full TC language. Concurrent processing is based on explicit multiple communicating FSA, called boxes. Within Hume, an explicit distinction is made between the coordination language, which describes external properties and configurations of boxes, and the expression language, which describes input/output transitions within boxes. Finally, in full Hume, both sub-languages

[1]School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Scotland, EH14 4AS, greg@macs.hw.ac.uk

[2]School of Computer Science, University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9AJ, kh@dcs.st-and.ac.uk

[3]LASMEA, Blaise Pascal University, Les Cezeaux, F-63177 Aubiere cedex, France, Jocelyn.Serot@lasmea.univ-bpclermont.fr

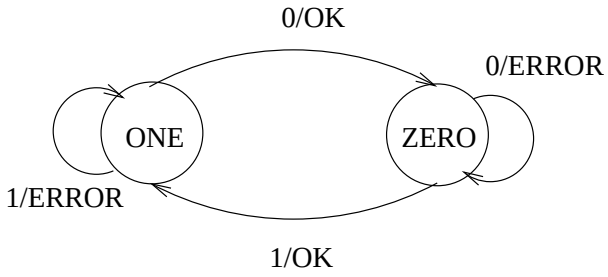

FIGURE 2.1. Mealy machine for alternating `1`s and `0`s

share a rich, polymorphic type system. These design decisions enable us to identify layers of language in Hume, with different decidable properties, which may be supported by high-level cost models [5].

A FSA with output (Mealy machine) is usually characterised by transition quadruplets of the form: (*old state*,*input*) → (*new state*,*output*)
where *old state*, *input*, *new state* and *output* are finite sets, for example, the Mealy machine which checks that a binary sequence has alternating `1`s and `0`s, shown in Fig. 2.1, has transitions:

```
(ZERO,0) -> (ZERO,ERROR)
(ZERO,1) -> (ONE,OK)
(ONE,0) -> (ZERO,OK)
(ONE,1) -> (ONE,ERROR)
```

However, both the diagrammatic and state transition characterisations are misleading. First of all, it is implicit that a FSA cycles indefinitely, communicating with an external environment to consume single input symbols and generating single output symbols. Secondly, it is implicit that a FSA retains its state in between cycles. The external input/output links and state retention are made explicit for the above example in Fig. 2.2.

In general, for one FSA it need not be specified where the input comes from or where the output goes to: both could be linked to arbitrary sources and sinks, including to other FSA. Similarly, in principle, the old and new state need not be a direct feedback link but could again come via arbitrary sources and sinks, including other FSA.

The state and I/O symbol sets for a FSA must be finite but they may also be very big. Given a large enough set that maps to integers, then complex data structures may be encoded using either Gödel numbers within the set, or, more familiarly, structured ASCII sequences whose concatenated bit values are integers within the set.

Noting that the left and right hand sides of traditional transitions are like two-element tuples, we generalise them to: *pattern* → *expression*. Here the left hand side *pattern* is composed of variables, constants and structures. Note the wildcard pattern `*` which ignores the corresponding inputs without consuming it. Similarly,

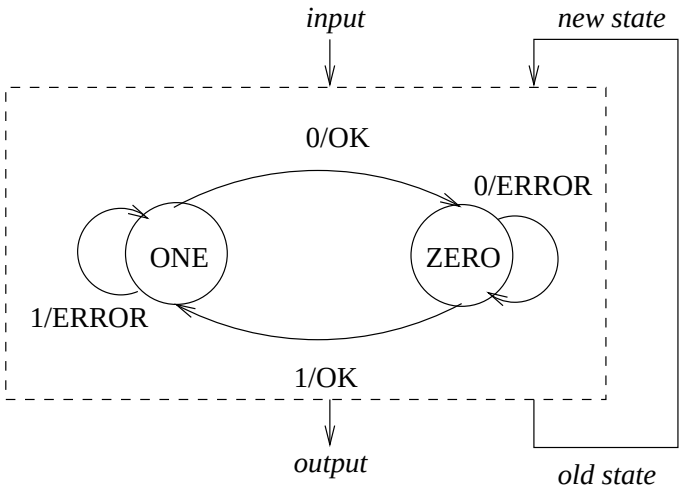

FIGURE 2.2. Mealy machine with explicit I/O and state

the right hand side *expression* may involve the components of the *pattern*, in particular the variables it introduces.

Thus, we generalise a FSA to a box with multiple input and output wires, where the state is no longer necessarily distinguishable from the input or output. Operationally, a box cycles repeatedly, trying to match transition *pattern*s against the current values on the input wires, treated as a single top-level tuple value. For a match to succeed, constants and constructors must appear in the same positions in the pattern and input value. Variables in the *pattern* are then instantiated to corresponding components of the input value. After a successful match, the output wires are instantiated from the tuple of values generated by the transition's right hand side.

For example, we can write the above Mealy machine in Hume as:

```
type BIT = int 1;
data STATE = ZERO | ONE;
stream Input from "std_in";
stream Output to "std_out";

box Bits
in (oldstate::STATE,input::BIT)
out (newstate::STATE,output::string)
match
 (ZERO,0) -> (ZERO,"ERROR\n") |
 (ZERO,1) -> (ONE,"OK\n") |
 (ONE,0) -> (ZERO,"OK\n") |
 (ONE,1) -> (ONE,"ERROR\n");
```

```
wire Bits (Bits.newstate initially ZERO,Input)
          (Bits.oldstate,Output);
```

Full Hume has constructs found in a contemporary polymorphic functional language, including recursive, unbounded, user-defined types. Finite State Machine Hume (FSM-Hume) is the Hume layer with finite types on wires and only simple operations, such as boolean and arithmetic, in transition expressions.

It might be thought that allowing operations whose state space is larger than the input space, such as multiplication, would transcend finite state-ness. However, for fixed precision numbers, it is possible to build a FSA that will carry out multiplication for values whose multiples do not exceed the largest allowed value, for example by encoding the appropriate look up table.

It might also be thought that Hume suffers from the same problems as other FSA-based languages, in particular state space explosion for practical verification of realistic programs. However, given appropriate transformation techniques, it should be possible to convert multiple boxes employing an impoverished expression language to fewer boxes using a richer expression language. Gross properties of box internals would still have to be established, using, say, automated theorem proving, but the state space of the overall box system would have been reduced. The balance between model checking and theorem proving in establishing properties of Hume programs is an interesting avenue of research which is not discussed further here.

A more serious concern is to clarify in what sense a multi-box Hume program is actually still a FSA, given the presence of multiple inputs and outputs, and the withering away of the state. We first discuss the status of a single box program and then explore multi-box programs.

Note that the following sections provide an informal framework for possible formalisation and are intended to convey conviction rather than establish correctness.

2.2 SINGLE BOX FSM-HUME PROGRAMS ARE FINITE STATE

Consider a Hume box with multiple inputs and outputs, and no distinguished state. As noted above, multiple values from finite domains, represented as a fixed width tuple, can be encoded as a single symbol, given a large enough space of symbols. Thus a box with multiple inputs or outputs may be treated as if it had just one input and output, each bearing a tuple value.

A multi-state FSA may be converted to a single state FSA as follows. The state symbol in each transition is combined with the input/output symbols in tuples. Each transition is then extended with a new single state value, in the state position on the left and right hand sides. In general:

$(old\ state, input) \rightarrow (new\ state, output) \Longrightarrow$
$(single\ state, (old\ state, input)) \rightarrow (single\ state, (new\ state, output))$

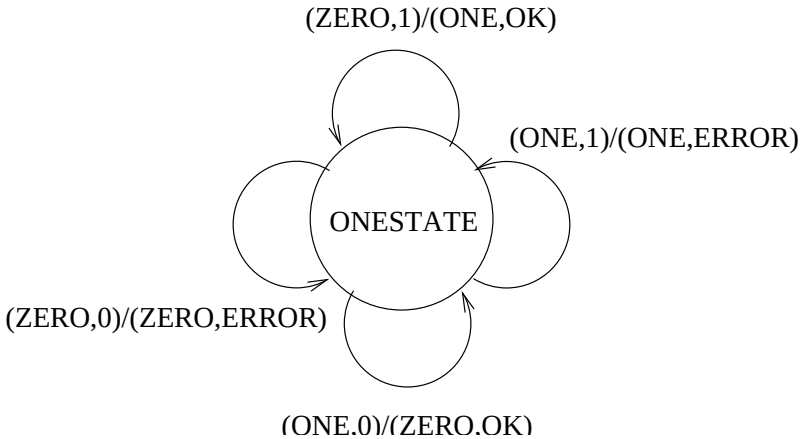

FIGURE 2.3. Single state Mealy machine for alternating `1`s and `0`s

For example, the Mealy machine above might be changed as shown in Fig. 2.3, with transitions:

```
(ONESTATE,(ZERO,0)) -> (ONESTATE,(ZERO,ERROR))
(ONESTATE,(ZERO,1)) -> (ONESTATE,(ONE,OK))
(ONESTATE,(ONE,0)) -> (ONESTATE,(ZERO,OK))
(ONESTATE,(ONE,1)) -> (ONESTATE,(ONE,ERROR))
```

Using this technique, a Hume box with multiple inputs and outputs, and no distinguished state, may be converted directly to a single state FSA with single composite input and output tuples, provided it has no variables in transition *pattern*s. A variable in a *pattern* corresponds to successfully matching any value in the domain for the variable's type. Thus, to fully convert a Hume transition with variables to pure FSA form, it must be replaced by multiple copies, with one copy for each combination of variable type domain values.

2.3 MULTI-BOX FSM-HUME PROGRAMS ARE FINITE STATE

We also need to convince ourselves that a multi-box FSM-Hume program is still finite state. If such a program may be converted into a single box FSM-Hume program then that program is finite state by the preceding argument.

Hume box scheduling is well defined as sequential, round robin where each box takes in it turns to execute once, in fixed sequence. For a multi-box program, we combine the box transitions and introduce an explicit state value to ensure sequentiality. Essentially, each transition for the combined box will correspond to a transition of one of the separate boxes, augmented with additional left hand side *pattern*s and right hand side *expression*s to circulate the wire values for all the other boxes without changing them.

In general, a successful transition for any one box must be able to transmit all possible wire values for the other boxes: any one box must be able to succeed if its inputs are matched successfully, regardless of the values on the wires for the other boxes. We employ Hume variables to generalise arbitrary input values, noting that they may in turn be replaced by all possible values of the corresponding types for pure FSAness, at the cost of a huge explosion in code size.

Suppose there are N boxes and box i has I_i inputs ($in_{i1}...in_{I_i}$) and O_i outputs ($out_{i1}...out_{O_i}$).

For each box, we construct a top level pattern template:

$$P_i\text{: } var_{i1},var_{i2}...var_{iI_i}$$

with a unique variable for each input. We also construct a top level expression template:

$$E_i\text{: } var'_{i1},var'_{i2}...var'_{iO_i}$$

where var'_{ij} is the new variable corresponding to the box input to which output out_{ij} is connected.

We then form a top level template for the transitions of the composite box by concatenating together the box pattern templates on the left and expression templates on the right:

$$(P_1,P_2...P_N) \rightarrow (E_1,E_2...E_N)$$

This template accepts arbitrary inputs and sends them to the appropriate outputs unchanged.

Suppose box i has T_i transitions, where the kth is: t_{ik}: $patt_{ik} \rightarrow exp_{ik}$
Then for each transition of box i, t_{ik}, we make a copy of the composite box's top level template, replace the pattern template P_i with the pattern $patt_{ik}$ and replace the expression template E_i with the expression exp_{ik}:

$$(P_1...patt_{ik}...P_N) \rightarrow (E_1...exp_{ik}...E_N)$$

Where the expression is a condition, the right hand side of the template must be pushed through to the condition options. Similarly, where the expression is a definition, the right hand side of the template must be pushed through to the result expression.

After this stage, where any remaining pattern template has a variable which has been replaced by an expression on the right hand side, then that variable should be replaced by the "ignore" pattern `*`: there should not be an input value present for that variable because a new value has been output for it. Similarly, where any expression template has a variable that was replaced in a pattern template, then that variable must be replaced by the "no output" operator `*`: the input has been consumed and cannot be re-circulated.

We are then left with common variables between left and right hand sides which consume inputs and reproduce them as outputs, to act as the inputs again on the next cycle. The effect is as if the corresponding wires had been ignored.

Thus, all variables on the left/right of a transition which are not in that transition's replacement pattern may be replaced by the "ignore" pattern/"no output" *.

Next, we introduce an explicit state which changes on each transition. We precede each composite pattern with the number of the corresponding box and each composite expression with the number of the next box:

$$(i,*,...,*,patt_{ik},*,...,*) \rightarrow (i+1,*,...*,exp_{ik},*,...,*)$$

or, for the last box, with the number of the first box.

Finally, we combine the wiring for each box, again adding a new feedback wire for the new explicit state.

The effect is two-fold. From a Hume perspective, we have constructed a single box which emulates multi-box scheduling. From a FSA perspective, we can easily convert the composite box into a FSA, with an explicit state, and composite input and output, using the technique described above.

2.4 EXAMPLE: VEHICLE SIMULATION

We now illustrate this transformation with reference to the simulation of a simple autonomous vehicle, which tries to follow a white line by repeatedly analysing a camera image consisting of one row of bits from a two-dimensional bit-map scene, effectively a map of the terrain the vehicle is traversing. The vehicle has a location consisting of its Cartesian coordinates in terrain space and its angle of orientation relative to the horizontal. The vehicle sends its current location to the environment. If the vehicle has not "bumped" into the edge of the terrain then the environment returns an image corresponding to the vehicle's position. The vehicle then sends the image to the control which calculates a new orientation to try to bring the white line back into the centre of the image. Finally, the vehicle changes its position and requests the next image from the environment. The vehicle also sends monitoring information to standard output:

```
box env in (loc::location) out (v::image,b::bool)
 match loc -> if within_scene loc
              then (lookat loc, false)
              else (null_image, true);

wire env (vehicle.loc initially init_loc)
         (vehicle.v, vehicle.b);

box vehicle
 in (v::image,b::bool,ploc::location,c::real)
 out (loc::location,m::monitor,
      loc'::location,v'::image)
 match
  (v, false, pl, c) ->
   let nl = move pl c
```

```
   in (nl, (v,pl,false,c,'\n'), nl, v)
| (v, true, pl, c) ->
   (init_loc, (v,pl,true,c,'\n'),
    init_loc, lookat init_loc);

wire vehicle
     (env.v,env.b,vehicle.loc' initially init_loc,
      control.da initially 0.0)
     (env.loc,std_out,vehicle.ploc,control.v);

box control in (v::image) out (da::real)
 match
   <<_,_,_,_,_,_,_,1,_,_,_,_,_,_,_,_>> -> 0.0
 ...
 | _ -> 0.0 ;

wire control (vehicle.v') (vehicle.c);
```

The simulation runs in real time and the vehicle never deviates more than a few bits to either side of the line.

2.4.1 Single-box FSM-Hume

First we construct the pattern templates and then the expression templates using the variable names from the pattern templates. We adopt the convention of naming template variables by preceding each input wire's name with a letter to denote its box name:

control pattern: `c_v`; env pattern: `e_loc`;
vehicle pattern: `v_v,v_b,v_ploc,v_c`
control expression: `v_c`; env expression: `v_v,v_b`;
vehicle expression: `e_loc,o,v_ploc,c_v`

i.e. the `control` output is wired to the `vehicle` input `c`; the env output is wired to the `vehicle` inputs `v` and `b`; etc.

The overall transition template is:

```
c_v,e_loc,v_v,v_b,v_ploc,v_c ->
 v_c,v_v,v_b,e_loc,o,v_ploc,c_v
```

Consider the first transition for the `control`. In the template, we replace `c_v` on the left with the transition pattern, `v_c` on the right with the transition expression and all other variables with `*`.

Consider the transition for the `env`. In the template, we replace `e_loc` on the left with the pattern. The transition expression is a conditional expression so we leave the condition in place, replace the option expressions with the template right hand side and insert the components expressions in place of the corresponding template variables `v_v` and `v_b`. Again, all other variables are replaced by `*`.

Consider the first transition for the `vehicle`. In the template, we replace `v_v`, `v_b`, `v_ploc` and `v_c` with the pattern components. There is a local definition on the right so we leave the declaration part in place, replace the expression with the template right hand side and insert the components of the expression in place of the corresponding template variables `e_loc`, `o`, `v_ploc` and `c_v`. Again, all other variables are replaced by `*`.

Numbering the boxes `control`/1, `env`/2 and `vehicle`/3, we add state patterns and expressions to each transition:

```
box vehicle
in (s::integer,c_v::image,e_loc::location,
    v_v::image,v_b::bool,v_ploc::location,v_c::command)
out (s'::integer,c_da::real,e_v::image,e_b::bool,
     v_loc::location, v_m::monitor,v_loc'::location,
     v_v'::image)
match
 (1,<<_,_,_,_,_,_,_,1,_,_,_,_,_,_,_>>,*,*,*,*,*) ->
  (2,0.0,*,*,*,*,*,*) |
 ...

 (2,*,loc,*,*,*,*) ->
      if within_scene loc
      then (3,*,lookat loc, false,*,*,*,*)
      else (3,*,null_image, true,*,*,*,*) |

 (3,*,*,v, false, pl, c) ->
  let nl = move pl c
  in (1,*,*,*,nl, (v,pl,false,c,'\n'), nl, v) |
 ...
```

Finally, we amalgamate the box wires and add appropriate wiring for the state, to start with the `env` box in state 2:

```
wire vehicle
 (vehicle.s' initially 2,vehicle.v_v',
  vehicle.v_loc initially init_loc,
  ...
  vehicle.c_da initially 0.0)
 (vehicle.s,vehicle.v_c,vehicle.v_v,vehicle.v_b,
  vehicle.e_loc, output,vehicle.v_ploc,vehicle.c_v);
```

The single box version of the vehicle simulation gives the same behaviour as the multi-box version, on the full Hume interpreter and on the HAM. It is also substantially faster and requires substantially less space.

2.5 CONCLUSION

We have explored the specific properties of the Hume finite state subset FSM-Hume to demonstrate informally that it is indeed finite state. In so doing, we derived a transformation to convert multi-box FSM-Hume programs to a single box and applied it to the simulation of a simple line following vehicle. We now plan to formalise and prove the transformation.

The application of the transformation to the vehicle simulation was performed by hand. We also plan to automate the transformation and to perform further experimentation to determine whether this transformation is a useful optimisation for general FSM-Hume programs.

This work has been partly supported by UK EPSRC grant GR/R 70545/01 and by a French CNRS grant.

REFERENCES

[1] W. S. Brainerd and L. H. Landweber. *Theory of Computation*. Wiley, 1974.

[2] R. Burstall. Inductively Defi ned Functions in Functional Programming Langauges. Technical Report ECS-LFCS-87-25, LFCS, University of Edinburgh, April 1987.

[3] K. Hammond and G. Michaelson. Hume: a Domain-Specifi c Language for Real-Time Embedded Systems. In *Proc. 2003 Intl. Conf. on Generative Programming and Component Engineering, – GPCE 2003, Erfurt, Germany*, LNCS, pages 37–56. Springer-Verlag, 2003.

[4] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[5] G. Michaelson, K. Hammond, and J. Serot. FSM-Hume: Programming Resource Limited Systems with Bounded Automata. In *Proc. ACM Symposium on Applied Computing (SAC'04), Cyprus, March 2004*. ACM Press, to appear.

[6] D. Turner. Elementary Strong Functional Programming. In *Proc. 1st Int. Symp. on Functional programming Languages in Education,Holland*, volume 1022 of *LNCS*. Springer, december 1995.