Chapter 1

The Finite State-ness of FSM-Hume

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

Abstract Hume is a domain-specific programming language targeting resource-bounded computations, such as real-time embedded systems. It is novel in being based on generalised concurrent bounded automata, controlled by transitions characterised by pattern matching on inputs and recursive function generation of outputs. FSM-Hume is a strict finite state subset of Hume, where symbols are constrained to fixed size types and transition functions are non-recursive. Here we discuss the design of FSM-Hume and show that it is indeed classically finite state.

1.1 INTRODUCTION

There is a tension in programming language design between expressibility and decidability. Ideally, we would like to be able to prove automatically the correctness, equivalence, termination, space use and complexity of arbitrary programs. However, these properties are all undecidable for Turing-complete (TC) languages.

Decidability may be achieved by restricting the types and constructs in a language. However, many attempts to do so have resulted in unwieldy languages where the programmer constantly fights such restrictions in search of greater expressibility.

Hume[HM03] is a novel language based on the concept of *language layers*. Full Hume is TC but has been designed to facilitate analyses that identify where program constructs break designated decidable properties.

[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`

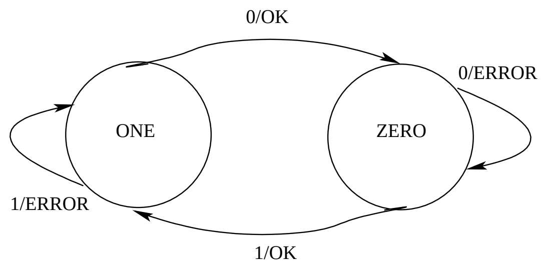

FIGURE 1.1. Mealy machine for alternating `1`s and `0`s.

As well as a research vehicle for programming language research, Hume is also a practical programming tool. Hume is defined by a structural operational semantics[Ham01], and a reference interpreter for the full language, written in Haskell, is near completion.

The following sections discuss Hume's design, the finite state subset of Hume, and its use in simulating an autonomous line following vehicle.

1.2 HUME DESIGN

At the opposite end of the decidability spectrum from TC-ness lie the finite state automata (FSA), for which all of the above properties are decidable. Several programming languages have been based on FSA, for example Esterel[Ber00] and Promela[Hol04] which are used to specify protocols and reactive systems. Decidable properties of programs may then be explored using automatic techniques such as model checking, for example through the Spin tool for Promela. However, because of the use of weak type systems and abstraction mechanisms, programs quickly become large and unwieldy, with vast state spaces when compiled to the low level FSA notations used for automatic analysis.

We have taken FSA as our starting point for Hume's design but with a number of fundamental differences to previous approaches. First of all, Hume is based on a generalisation of standard FSA transition notation, to encompass, at the limit, a full TC language. Secondly, we introduce concurrent processing through explicit multiple communicating FSA, which we call *boxes*. Thirdly, we make an explicit distinction 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 coordination and expression languages share a rich, polymorphic type system. These design decisions enable us to identify layers of language in Hume, with different decidable properties, as we discuss below.

To foreground the design, consider the Mealy machine shown in Figure 1.1, which checks that a binary sequence has alternating `1`s and `0`s: A Mealy machine may be characterised by transitions quadruplets of the form:

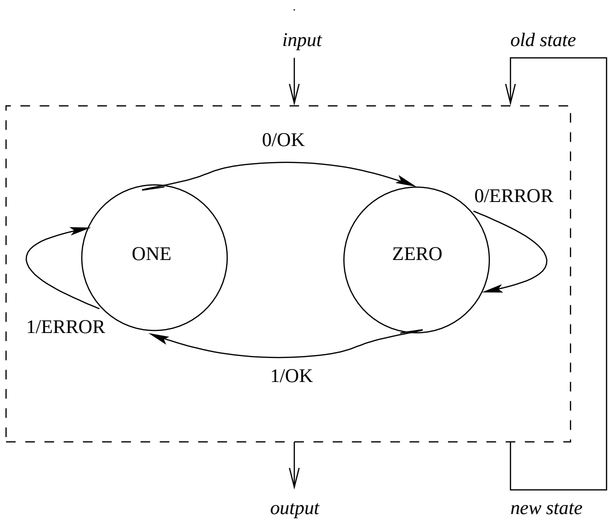

FIGURE 1.2. Mealy machine with explicit I/O and state

$(oldstate, input) \to (newstate, output)$

where *old state*, *input*, *new state* and *output* are finite sets. Thus, the above Mealy machine has transitions:

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

Note that 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 in Figure 1.2. 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.

We noted above that the state and I/O symbol sets for a FSA must be finite. However, these sets 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 bits values are integers within the set

In Hume, we allow values of the form shown in Figure 1.2.

value	$\rightarrow$	*integer*	– finite integer
	$\mid$	*float*	– finite float
	$\mid$	*character*	– character
	$\mid$	*boolean*	– boolean value
	$\mid$	(*value,value...value*)	– tuple
	$\mid$	<*value,value...value*>	– vector
	$\mid$	[*value,value...value*]	– list
	$\mid$	*ident value value ... value*	– discriminated union

FIGURE 1.3. Abstract syntax for Hume values.

patt	$\rightarrow$	*var*	– variable
	$\mid$	*integer*	– finite integer
	$\mid$	*float*	– finite float
	$\mid$	*character*	– character
	$\mid$	*boolean*	– boolean value
	$\mid$	(*patt,patt...patt*)	– tuple
	$\mid$	<*patt,patt...patt*>	– vector
	$\mid$	[*patt,patt...patt*]	– list
	$\mid$	*ident patt patt ... patt*	– discriminated union
	$\mid$	*	– ignore wildcards

FIGURE 1.4. Abstract syntax for Hume patterns.

Here, *integer*, *float*, *character* and *boolean* are finite base types. A *tuple* is a fixed width sequence of values of possibly different types. A *vector* is a fixed width sequence of values of the same type. A *list* is a an arbitrary length sequence of values of the same type. A *discriminated union* is an arbitrary length sequence of values of possibly different types. We also employ the standard string notation for vectors of characters.

The left and right hand sides of traditional transitions are like two-element tuples so we generalise them to:

pattern $\rightarrow$ *expression*

Here the left hand side *pattern* is composed of variables, constants and structures, as shown in Figure 1.2. Note the *wildcard* pattern * which ignores the corresponding input's without consuming it.

Similarly, the right hand side *expression* may involves the components of the *pattern*, in particular the variables it introduces, as shown in Figure 1.2.

Thus, we generalise a FSA to a *box* with input and output *wires*. Note that we allow multiple input and output wires, and that the state is no longer necessarily distinguishable from the input or output. We shall return to this below.

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

exp	→	*value*	– value
	|	*var*	– variable
	|	if *exp* then *exp* else *exp*	– conditional exp
	|	case *exp* of *transitions*	– case exp
	|	let *defs* in *exp*	– local definition
	|	*var exp exp ... exp*	– function application
	|	*exp op exp*	– infix operator
	|	*	– no output

FIGURE 1.5. Abstract syntax for Hume expressions.

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:

```
 1 type BIT = int 1;

 2 data STATE = ZERO | ONE;

 3 stream Input from "std_in";
 4 stream Output to "std_out";

 5 box Bits
 6 in (oldstate::STATE,input::BIT)
 7 out (newstate::STATE,output::string)
 8 match
 9  (ZERO,0) -> (ZERO,"ERROR\n") |
10  (ZERO,1) -> (ONE,"OK\n") |
11  (ONE,0) -> (ZERO,"OK\n") |
12  (ONE,1) -> (ONE,"ERROR\n");

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

Line 1 defines a new type bit to be a one-bit-precision integer.

Line 2 defines a new type STATE with values ZERO and ONE.

Lines 3 and 4 associate the *stream*s Input and Output with standard input and standard output respectively.

Line 5 introduces the box Bits with input wires oldstate and input, and output wires newstate and output.

Lines 9 to 12 are the transitions. Note that the output is generated by an explicit string. Transitions are associated with wires by position. For example, for

Type	Control	Properties
finite	operations	decidable E & T; precise Sp & Ti
finite	conditions	decidable E & T; bounded Sp & Ti
stack + *	operations	decidable E & T; precise Ti
finite + *	operations	decidable T; precise Sp & Ti
finite	prim. rec.	decidable T; bounded Sp & Ti
infinite	gen. rec.	-

FIGURE 1.6. Type, control and decidability. E = equivalence, T = termination, Sp = Space, Ti = Time

the first transition: wire `o`ldstate matches `ZERO`; wire `i`nput matches `0`; `ZERO` sets wire `n`ewstate; `"ERROR\n"` sets wire `o`utput.

Lines 13 and 14 wire the box to itself and to standard input and output. Wiring is by position, with input wiring followed by output wiring, so: in wire `oldstate` is wired to `Bits` out wire `newstate`; in wire `input` is wired to stream `Input`; out wire `newstate` is wired to `Bits` in wire `Bits.newstate`; in wire `input` is wired to stream `Input`.

We will not discuss in further detail the *type system* or *definition language* shared by the expression and coordination languages. Some flavour of both may be gained from the above example.

1.3 HUME AS A LAYERED LANGUAGE

We noted above that interesting properties of TC languages are undecidable [HU69, BL74] and that languages with decidable properties tend to be hard to program in. An important motivation for Hume's design was to enable analyses to identify whether or not arbitrary Hume programs met particular criteria corresponding to required decidable properties. Hume's generalised FSA transitions provide the locus for such analyses: constraining the types of inputs that may be matched by left hand *pattern*s and the control structures used in right hand *expression*s to generate outputs, directly constrains decidable program properties, as summarised in Figure 1.3.

With finite types on wires and corresponding operations in transitions, equivalence, termination, space use and time are decidable. Adding conditions to operations for a finite input FSA loses precise time and space bounds because condition branch choice is unpredictable for arbitrary input. However, it may be possible to establish accurate upper bounds.

With an infinite stack on wires, and the ability to ignore wires (denoted by the pattern `*`), a FSA is equivalent to a Push Down Automata, for which decidable equivalence has just been established. A PDA must terminate or repeat a state on a finite input but space use for an arbitrary input is undecidable.

With a finite type on wires and the ability to ignore wires, a FSA is equivalent to a Linear Bounded Automata (LBA). A LBA must terminate or repeat a state on finite input but equivalence is undecidable for arbitrary input.

Replacing conditions with primitive recursion loses decidable equivalence. Finally, allowing infinite data or unbounded general recursion makes all properties indeterminate in the general case.

Standard type and control flow analyses may be used to determine whether or not an arbitrary box is a FSA, PDA or LBA, or if it is recursive or constructs potentially infinite values. However, in general it is undecidable whether an arbitrary recursive function is primitive or general recursive: deciding primitive recursion with implied termination would solve the halting problem.

1.4 FSM-HUME

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.

FSM-Hume has been implemented through a compiler to *HAM* abstract machine code, with an associated interpreter which runs under RT-Linux, as well as other Unix flavours. A static cost modeler[HM02] has been constructed for FSM-Hume, which gives very accurate predictions of HAM space use.

A number of substantial FSM-Hume programs have been constructed. In particular, a rational reconstruction of Burns and Wellings' mine pump control system[BW01] easily meets their real-time constraints on the HAM interpreter. It consists of 250 lines of Hume, including the simulation environment, compared with 750 lines of Ada, without the simulation environment.

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.

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.

1.4.1 Single box FSMH 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. Furthermore, the sources and sinks of individual wires are irrelevant from the perspective of the box: what is important is the presence of appropriate values on all wires at the appropriate stage in the box execution cycle. 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

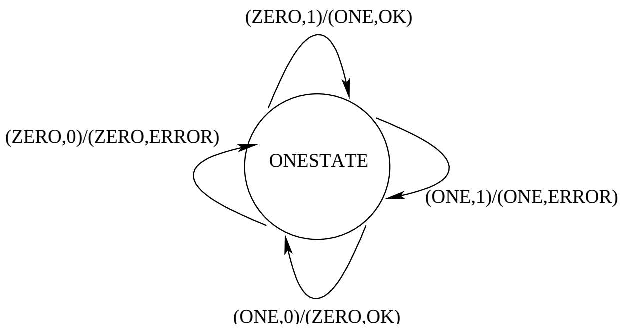

FIGURE 1.7. Single state Mealy machine for alternating `1`s and `0`s.

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:

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

For example, the Mealy machine above might be changed to:

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

as shown in Figure 1.7. Thus, the former state {ZERO,ONE} is now part of the tupled input/output symbol.

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. For example, the transition in the Hume box:

```
box add
in (x,y::int 64)
out (s::int 64)
match
 (a,b) -> a+b;
```

is equivalent to:

```
...
(0,-1) -> -1 |
(0,0) -> 0 |
(0,1) -> 1
...
```
for all $2^{64} * 2^{64} - 2^{64}$ unique combinations of values of `x` and `y`.

1.4.2 Multi-box FSMH 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. This explicit sequential scheduling greatly eases reasoning about Hume programs. However, though not discussed here, it is straightforward to introduce more elaborate scheduling by passing explicit schedule information from box to box.

Our approach to converting a multi-box program to a single box program is to combine the box transitions, and to 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. It is not enough to accept only valid input values for 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. In principle, given that inputs are fixed size, we could generate all possible combinations of input values for all boxes. However, as the simple example above suggested, the overall state space is somewhat large. Instead, we employ Hume variables to generalise arbitrary values, noting that they may in turn be replaced by specific values for pure FSAness as discussed above.

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: $var_{i1}, var_{i2}...var_{iI_i}$

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

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

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. Examples of these cases are shown below.

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 a 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 recirculated.

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" value *.

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. That single box starts with the input state wire initialised to the number of the first box. Thus, only the transitions for the first box can succeed and inputs for any other box are transmitted unchanged as outputs. Furthermore, the output state wire is set so that only the transitions for the next box in sequence will be considered on the next execution cycle. Thus, boxes are executed in sequential lock step.

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.

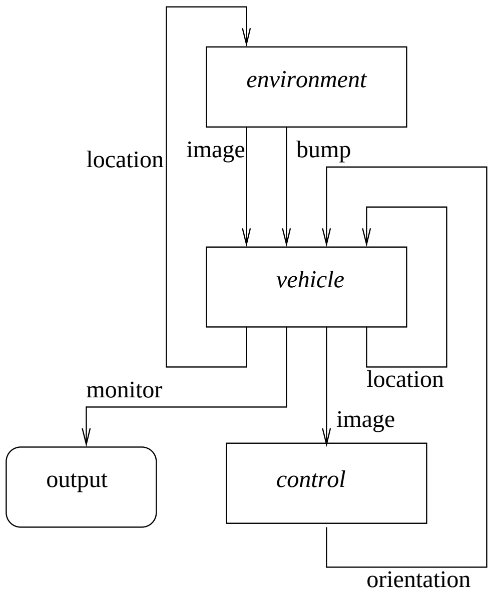

FIGURE 1.8. Vehicle simulation overview.

1.5 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.

Figure 1.8 gives an overview of the simulation. 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.

We now consider the simulation in more detail. We have omitted type and constant definitions where there values are not significant. A scene is represented as a vector of vector of bits, and an image as a vector of bits.

Note that FSM-Hume constructs for vector manipulation demonstrate what would normally be considered poor programming style. Where an iteration or recursion over a sequence would be employed in a TC language, in FSM-Hume such operations must be nominated explicitly, element by element.

First of all, consider the environment:

```
within_scene (y,x,radians) = ...

look (y0,x0,radians) x = ...

lookat loc =
 <<look loc -7,look loc -6,look loc -5,
   ...
   look loc 5,look loc 6,look loc 7>>;

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);
```

The environment accepts a location from the vehicle. If it is within the scene then the image at that position is calculated (`lookat`) and returned with a “no bump” flag. Otherwise, the empty scene and a “bump” flag are returned. Note that `lookat` calculates each element of the image explicitly where it would be “better style” to iterate over the scene.

Next, consider the vehicle:

```
move (y,x,a) da = ...

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) ;

stream output1 to "std_out";

wire vehicle
     (env.v,env.b,vehicle.loc' initially init_loc,
      control.da initially 0.0)
     (env.loc,output1,vehicle.ploc,control.v);
```

The vehicle accepts an image and bump flag from the environment, its current location from its previous cycle, and a new orientation from the control. If it hasn’t "bumped" then it calculates a new location `move`. It then sends that new location to the environment, monitoring information to the output, the new location to itself and the image to the control. If the vehicle has bumped then the simulation is reset to the start state.

Finally, consider the control:

```
box control in (v::image) out (da::real)
 match
   <<_,_,_,_,_,_,_,1,_,_,_,_,_,_,_>> -> 0.0
 | <<_,_,_,_,_,_,1,_,_,_,_,_,_,_,_>> ->
    -1.0*delta_a |
 | <<_,_,_,_,_,_,_,_,1,_,_,_,_,_,_>> ->
    1.0*delta_a |
 | <<_,_,_,_,_,1,_,_,_,_,_,_,_,_,_>> ->
    -2.0*delta_a |
 ...
 | _ -> 0.0 ;

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

The control accepts an image from the vehicle and returns a new orientation to the vehicle. Note the use of explicit pattern matching on all possible positions of a `1`, indicating the line, within the image, where it would be "better style" to iterate over the image. The wildcard _ consumes and ignores the matching value.

Note that the control and environment are one step out of sequence so that the new orientation is computed from the old image at the new location.

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

1.5.1 Single-box FSM-Hume

First we construct the pattern templates using the convention of constructing the 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
```

Next we construct the expression templates using the variable names from the pattern templates:

```
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 `*`:

```
 (<<_,_,_,_,_,_,_,1,_,_,_,_,_,_,_>>,*,*,*,*,*) ->
  (0.0,*,*,*,*,*,*) |
```

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 `*`:

```
(*,loc,*,*,*,*) -> if within_scene loc
                   then (*,lookat loc, false,*,*,*,*)
                   else (*,null_image, true,*,*,*,*)
```

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 `*`:

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

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, top start with the `env` box in state 2:

```
wire vehicle
 (vehicle.s' initially 2,vehicle.v_v',
  vehicle.v_loc initially init_loc,
  vehicle.e_v, vehicle.e_b,
  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.

1.6 CONCLUSION

We have introduced the Hume programming language, surveyed its general properties as a layered language, explored the specific properties of the 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.

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

Longer term, we are investigating properties and analyses for FSM-Hume augmented with higher order functions over vectors and for the primitive recursive layer of Hume. We are also developing the vehicle simulation and plan to successively introduce a single 2-D camera, a pair of 2-D cameras enabling stereo vision in a 3-D scene, and proximity sensors. In the longer term, we hope to mount FSM-Hume under RT-Linux to control a small electric vehicle.

ACKNOWLEDGMENTS

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

REFERENCES

[Ber00] G. Berry. The Foundations of Esterel. In Colin Stirling Gordon Plotkin and Mads Tofte, editors, *Proof, Language, and Interaction*. MIT, 2000.

[BL74] W. S. Brainerd and L. H. Landweber. *Theory of Computation*. Wiley, 1974.

[BW01] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley Longman, 2001.

[Ham01] K. Hammond. The Dynamic Properties of Hume: a Functionally-Based Concurrent Language with Bounded Time and Space Behaviour. In *Proc. Impl. of Funct. Langs. (IFL 2000), Aachen, Germany, Sept. 2000*, number 2011, pages pp. 122–139. Springer-Verlag Lecture Notes in Computer Science, 2001.

[HM02] K. Hammond and G. Michaelson. Predictable Space Behaviour in FSM-Hume. In *Proc. 2002 Intl. Workshop on Implementation of Functional Languages (IFL '02), Madrid, Spain*, number 2670. Springer-Verlag Lecture Notes in Computer Science, September 2002.

[HM03] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. 2003 Intl. Conf. on Generative Programming and Component Engineering, – GPCE 2003, Erfurt, Germany*. Springer-Verlag Lecture Notes in Computer Sciences, September 2003.

[Hol04] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[HU69] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, 1969.