FSM-Hume: Programming Resource-Limited Systems using Bounded Automata

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

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

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

ABSTRACT

Hume is a novel domain-specific programming language targeting resource-bounded computations, such as real-time embedded systems or mobile code. It is based on generalised concurrent automata, controlled by transitions characterised by pattern matching on inputs and (recursive) function generation on outputs. This paper discusses trade-offs between expressibility and decidability in the design of FSM-Hume, a subset of Hume (or Hume *layer*) based on generalised linear bounded automata with statically determinable time and space use. We illustrate our approach with reference to space costing of a simple real-time simulation of a line-following autonomous vehicle.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: [Multi-paradigm languages, Applicative (functional) languages, Concurrent, distributed and parallel languages]

General Terms

LANGUAGES, DESIGN, EXPERIMENTATION

Keywords

Functional language, finite state automata, multi-process simulation

1. INTRODUCTION

There is a tension in programming language design between expressibility and decidability. Ideally, we would like to be able to automatically prove the correctness, equivalence, termination, space use and complexity of arbitrary programs. Such properties are important for resource-constrained systems such as real-time embedded systems, or mobile code. However, these properties are all undecidable for Turing-complete (TC) languages. Decidability may be achieved by restricting the types and constructs in a language. Many attempts to do so have, however, resulted in unwieldy languages where the programmer constantly fights such restrictions in search of greater expressibility. Hume[5] 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. Each increasingly restrictive language layer thus exposes increasingly strong properties. Rather than attempting to apply cost modelling and correctness proving technology to an existing language framework (such as Standard ML or Haskell) by altering the language to a greater or lesser extent, our approach is to design Hume in such a way that we are certain that formal models and proofs can be constructed.

2. FSA AS PROGRAMMING TOOLS

Classical *finite state automata* (FSA), for which all of the properties that we identified in the introduction are decidable, lie at the opposite end of the decidability spectrum from TC-ness. FSA are generally regarded as modeling tools rather than programming notations. They are well-suited to this purpose, since formally-based techniques such as model checking can be used to verify required program properties. For example, the Statecharts notation [7] is widely used to model synchronous systems.

Several programming languages have been based on FSA, for example Esterel [2] and Promela [9] which are used to specify protocols and reactive systems respectively. Decidable properties of programs may then be explored using automatic techniques such as *model checking*, for example through the Spin tool for Promela [9]. However, such languages are relatively inexpressive, typically lacking one or more essential programming features such as asynchronous communication, complex data structures, exceptions or interrupts. Furthermore, 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.

An opposite approach has been taken by the UPPAAL group, who *synthesize* timed automata from pseudo-assembler code in order to verify properties of reachability and schedulability [11, 1]. As with Hume, such an approach allows a choice

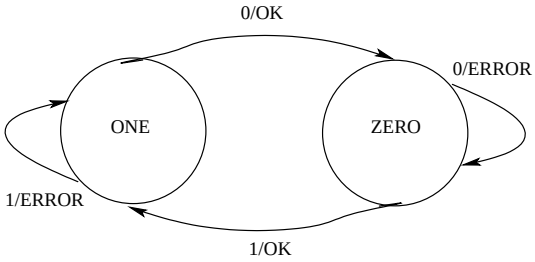

Figure 1: Mealy machine for alternating `1`s and `0`s.

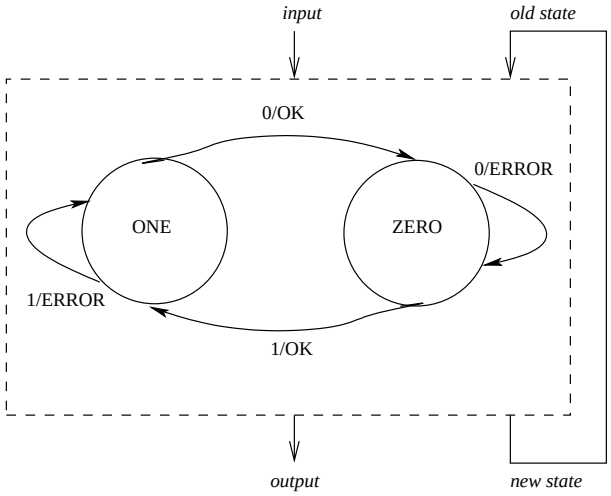

Figure 2: Mealy machine with explicit I/O and state

to be taken between expressibility and decidability. However, with the UPPAAL approach there is a wide semantic gap between the programming notation and the proof technology. Consequently, programs may need major rewriting in order to ensure that required properties are maintained. In Hume, in contrast, the choice of language layer determines which properties can be decided. Moreover, as a high-level notation, it is relatively straightforward to rewrite Hume source code if required.

3. HUME DESIGN

The Hume design starts from FSA but incorporates a number of fundamental differences from previous approaches. Firstly, Hume is based on a generalisation of standard FSA transition notation, to encompass, at the limit, a full TC language. Secondly, in constrast to typical functional languages such as Standard ML or Haskell, Hume incorporates 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, which checks that a binary sequence has alternating `1`s and `0`s. A Mealy machine may be characterised by transition quadruplets of the form:

$$(\textit{old state}, \textit{input}) \rightarrow (\textit{new state}, \textit{output})$$

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 slightly misleading. First of all, it is implicit that an FSA cycles indefinitely, communicating with an external environment to consume single input symbols and generating single output symbols. Secondly, it is implicit that an FSA retains its state in-between cycles.

The external I/O links and state retention are made explicit in Figure 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 an 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 bit values are integers within the set.

In Hume, we allow values of the form shown in Figure 3. 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 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 reminiscent of 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 4. Note the *wildcard* pattern `*` which ignores the corresponding input without consuming it. Similarly, the right-hand side *expression* may involve the components of the *pattern*, in particular the variables it introduces, as shown in Figure 5. Thus, we generalise an 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.

value	→	*integer*	– finite integer
	|	*float*	– finite float
	|	*character*	– character
	|	*boolean*	– boolean value
	|	(*value*,*value*,. . .,*value*)	– tuple
	|	<<*value*,*value*,. . .,*value*>>	– vector
	|	[*value*,*value*,. . .,*value*]	– list
	|	*constr value value . . . value*	– discriminated union

Figure 3: Abstract syntax for Hume values.

patt	→	*var*	– variable
	|	*integer*	– finite integer
	|	*float*	– finite float
	|	*character*	– character
	|	*boolean*	– boolean value
	|	(*patt*,*patt*,. . .,*patt*)	– tuple
	|	<<*patt*,*patt*,. . .,*patt*>>	– vector
	|	[*patt*,*patt*,. . .,*patt*]	– list
	|	*constr patt patt . . . patt*	– discriminated union
	|	*	– ignore input

Figure 4: Abstract syntax for Hume patterns.

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:

```
 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 the first transition: wire `oldstate` matches `ZERO`; wire `input` matches `0`; `ZERO` sets wire `newstate`; `"ERROR\n"` sets wire `output`. 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 out-wire `newstate` of box `Bits`; in-wire `input` is wired to stream `Input`; out-wire `newstate` is wired to in-wire `newstate` of box `Bits`; and in-wire `input` is wired to stream `Input`.

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 5: Abstract syntax for Hume expressions.

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 6: Type, control and decidability. E = equivalence, T = termination, Sp = Space, Ti = Time

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.

4. HUME LAYERS AND FSM-HUME

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[10, 3]. 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 6.

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 `*`), an 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, an FSA is equivalent to a Linear Bounded Automata (LBA). An 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 an FSA, PDA or LBA, or if it is recursive or constructs potentially infinite values. However, it is undecidable whether or not an arbitrary general recursive expression has an equivalent primitive recursive formulation.

Finite State Machine Hume (FSM-Hume) is the Hume subset corresponding to LBA. Thus FSM-Hume subsumes the FSA and PDA layers over bounded, finite input. Only finite types may appear on wires (i.e. base types, tuples and vectors) and only non-recursive control constructs may be used (i.e. operations, conditions, non-recursive functions). Thus, for a given set of inputs, each box cycle must terminate in bounded time using bounded space.

In principle, FSM-Hume might be extended with standard higher order functions over vectors such as `map` and `fold` without losing essential language properties: these functions are known to terminate and have well-defined time and space cost models. Such functions have not yet been incorporated into the language, however.

5. THE AUTONOMOUS VEHICLE SIMULATION

We have constructed a simulation of a basic autonomous vehicle in FSM-Hume. The vehicle 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 application is taken from an EU-funded project which is developing vision-based controllers for the CyCab electric vehicle.

Figure 7 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, and some function definition details, where they are not germane to the discussion. Note that a scene 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.

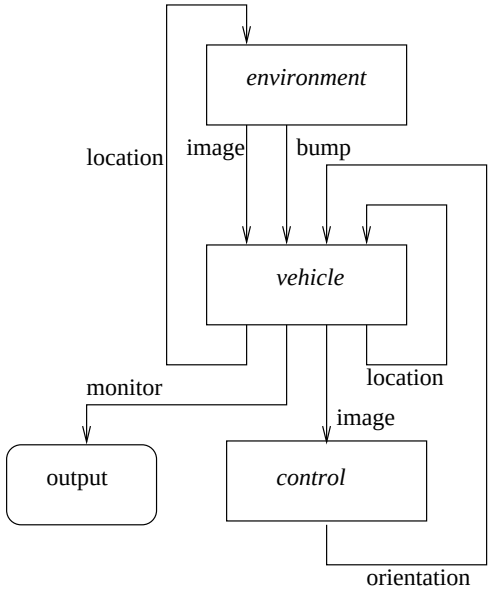

Figure 7: Vehicle simulation structure.

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,_,_,_,_,_,_,_,_,_,_>> ->
    -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.

6. FSM-HUME SPACE COST MODEL

Figures 8–10 outline a space cost model for FSM-Hume boxes, based on an operational interpretation of the Hume Abstract Machine (HAM) implementation. Heap and stack costs are each integer values of type *Cost*, labelled h and s, respectively. Each rule produces a pair of such values representing an independent upper bound on the stack and heap usage. The result is produced in the context of an environment, E, that maps function names to the space (heap and stack) requirements associated with executing the body

$$\boxed{\mathrm{E} \overset{\text{box}}{\vdash} \mathit{box} \Rightarrow \mathit{Cost}, \mathit{Cost}}$$

$$(1)\quad \frac{\begin{array}{c} \mathit{ins} \;=\; (\; \mathit{var}_1 :: \tau_1, \ldots,\; \mathit{var}_n :: \tau_n \;) \\ \forall i.\; 1 \le i \le n,\;\; \mathrm{E} \overset{\text{type}}{\vdash} \tau_i \Rightarrow h_i \\ \mathrm{E} \overset{\text{body}}{\vdash} \mathit{body} \Rightarrow h, s \end{array}}{\mathrm{E} \overset{\text{box}}{\vdash} \texttt{box}\ b\ \texttt{in}\ \mathit{ins}\ \texttt{out}\ \mathit{outs}\ \mathit{body}\ ; \Rightarrow \sum_{i=1}^{n} h_i + h, s}$$

$$\boxed{\mathrm{E} \overset{\text{body}}{\vdash} \mathit{body} \Rightarrow \mathit{Cost}, \mathit{Cost}}$$

$$(2)\quad \frac{\begin{array}{c} \forall i.\; 1 \le i \le n,\;\; \mathrm{E} \overset{\text{patt}}{\vdash} \mathit{patt}_i \Rightarrow \mathit{sp}_i \\ \forall i.\; 1 \le i \le n,\;\; \mathrm{E} \overset{\text{space}}{\vdash} \mathit{exp}_i \Rightarrow h_i, s_i \end{array}}{\begin{array}{l} \mathrm{E} \overset{\text{body}}{\vdash} \texttt{match}\ \mathit{patt}_1 -> \mathit{exp}_1 \mid \ldots \mid \mathit{patt}_n -> \mathit{exp}_n \\ \qquad \Rightarrow \;\; \max_{i=1}^{n} h_i,\;\; \max_{i=1}^{n} (s_i + \mathit{sp}_i) \end{array}}$$

Figure 8: Space cost axioms for boxes

of the function. This environment is derived from the top-level program declarations plus standard prelude definitions. Rules for building the environment are omitted here, except for local declarations, but can be trivially constructed.

Rules 1 and 2 (Figure 8) cost boxes and box bodies, respectively. The cost of a box is derived from the space requirements for all box inputs plus the maximum cost of the individual rule matches in the box. The cost of each rule match is derived from the costs of the pattern and expression parts of the rule. Since the abstract machine copies all available inputs into box heap from the wire buffer before they are matched, the maximum space usage for box inputs is the sum of the maximum space required for each input type. The space required for each output wire buffer is determined in the same way from the type of the output value. The derivation of these sizes is straightforward and is omitted here, but can be found in an earlier paper [4].

Figure 9 gives cost rules for a representative subset of FSM-Hume expressions. The heap cost of a standard integer is given by $\mathcal{H}_{int32}$ (rule 3), with other scalar values costed similarly. The cost of a function application is the cost of evaluating the body of the function plus the cost of each argument (rule 4). Each evaluated argument is pushed on the stack before the function is applied, and this must be taken into account when calculating the maximum stack usage. The cost of building a new data constructor value such as a tuple (rule 6) or a user-defined constructed type (rule 5) is similar to a function application, except that pointers to the arguments must be stored in the newly created closure (one word per argument), and fixed costs $\mathcal{H}_{con}$ and $\mathcal{H}_{tuple}$ are added to represent the costs of tag and size fields. The heap usage of a conditional (rule 7) is the heap required by the condition part plus the maximum heap used by either branch. The maximum stack requirement is simply the maximum required by the condition and either branch. Case expressions (omitted) are costed analogously. sequence (used to calculate the size of the stack frame for the local

$$\boxed{\mathrm{E} \overset{\text{space}}{\vdash} exp \Rightarrow Cost, Cost}$$

(3) $$\frac{}{\mathrm{E} \overset{\text{space}}{\vdash} integer \Rightarrow \mathcal{H}_{int32}, 1}$$

...

(4) $$\frac{\mathrm{E}\ (var) = \langle h, s \rangle \quad \forall i.\ 1 \le i \le n,\ \mathrm{E} \overset{\text{space}}{\vdash} exp_i \Rightarrow h_i, s_i}{\mathrm{E} \overset{\text{space}}{\vdash} var\ exp_1\ \ldots\ exp_n \Rightarrow \sum_{i=1}^{n} h_i + h,\ \max_{i=1}^{n}\ (s_i + (i-1)) + s}$$

(5) $$\frac{\forall i.\ 1 \le i \le n,\ \mathrm{E} \overset{\text{space}}{\vdash} exp_i \Rightarrow h_i, s_i}{\mathrm{E} \overset{\text{space}}{\vdash} constr\ exp_1\ \ldots\ exp_n \Rightarrow \sum_{i=1}^{n} h_i + n + \mathcal{H}_{constr},\ \max_{i=1}^{n}\ (s_i + (i-1))}$$

(6) $$\frac{\forall i.\ 1 \le i \le n,\ \mathrm{E} \overset{\text{space}}{\vdash} exp_i \Rightarrow h_i, s_i}{\mathrm{E} \overset{\text{space}}{\vdash} (\ exp_1,\ \ldots,\ exp_n\ ) \Rightarrow \sum_{i=1}^{n} h_i + n + \mathcal{H}_{tuple},\ \max_{i=1}^{n}\ (s_i + (i-1))}$$

...

(7) $$\frac{\mathrm{E} \overset{\text{space}}{\vdash} exp_1 \Rightarrow h_1, s_1 \quad \mathrm{E} \overset{\text{space}}{\vdash} exp_2 \Rightarrow h_2, s_2 \quad \mathrm{E} \overset{\text{space}}{\vdash} exp_3 \Rightarrow h_3, s_3}{\mathrm{E} \overset{\text{space}}{\vdash} \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \Rightarrow h_1 + max(h_2, h_3), max(s_1, s_2, s_3)}$$

Figure 9: Space cost axioms for representative expressions

$$\boxed{\mathrm{E} \overset{\text{patt}}{\vdash} patt \Rightarrow Cost}$$

(8) $$\frac{}{\mathrm{E} \overset{\text{patt}}{\vdash} integer \Rightarrow 0}$$

...

(9) $$\frac{}{\mathrm{E} \overset{\text{patt}}{\vdash} var \Rightarrow 1}$$

(10) $$\frac{\forall i.\ 1 \le i \le n,\ \mathrm{E} \overset{\text{patt}}{\vdash} patt_i \Rightarrow s_i}{\mathrm{E} \overset{\text{patt}}{\vdash} constr\ patt_1\ \ldots\ patt_n \Rightarrow \sum_{i=1}^{n} s_i + n}$$

(11) $$\frac{\forall i.\ 1 \le i \le n,\ \mathrm{E} \overset{\text{patt}}{\vdash} patt_i \Rightarrow s_i}{\mathrm{E} \overset{\text{patt}}{\vdash} (\ patt_1,\ \ldots,\ patt_n\ ) \Rightarrow \sum_{i=1}^{n} s_i + n}$$

Figure 10: Stack cost axioms for patterns

Name			Actual	Pred.	(P-A)/A	
control	box	heap	57	60	3	5.3%
		stack	36	42	6	16.7%
	wires	heap	47	50	3	6.4%
			0	0	0	-
env	box	heap	49099	49580	481	1.0%
		stack	129	137	8	6.2%
	wires	heap	11	11	0	-
		stack	0	0	0	-
vehicle	box	heap	49164	49648	484	1.0%
		stack	133	140	7	5.3%
	wires	heap	62	65	3	4.8%
		stack	0	0	0	-
	total	heap	98511	99538	1027	1.0%
	total	stack	299	322	23	7.7%

Figure 11: Actual and predicted HAM space use for vehicle simulation (in words).

Finally, patterns contribute to stack usage in two ways (Figure 10): firstly, the value attached to each variable is recorded in the stack frame (rule 9); and secondly, each nested data structure that is matched must be unpacked onto the stack (requiring n words of stack) before its components can be matched by the abstract machine (rules 10 and 11).

Types have been used to capture space usage information in a number of other contexts. For example, region types [12] allow memory cells to be tagged with an allocation *region*, thereby eliminating the need for garbage collection in certain cases. Hofmann's LFPL [8] uses linear typing to identify "diamond resource types" that fulfil a similar purpose. In neither case is the intention to infer bounds for memory Finally, E-FRP/RT-FRP [13] exploit tail-recursion to control resource usage in a similar fashion to Hume boxes, exposing space/time information in the form of difference equations between iterations.

7. SPACE USAGE IN THE SIMULATION

The space cost model described in Section 6 has been implemented as part of the HAM compiler. Figure 11 shows the predicted and actual heap and stack use (calculated in 4-byte words) for the vehicle simulation running on the HAM and using our static cost modeller [4]. This represents the majority of the dynamic memory usage for this application: the remainder comprises the operating system stack used for runtime system calls plus runtime state used to maintain boxes, create communication links etc.

Overall, the predictions are very accurate for both stack and heap usage, with predicted heap usage accurate to 1% and stack usage accurate to 7.7%. The deviations from actual usage are a result of unused conditional branches/cases and conservative I/O buffer allocations. Since these deviations result entirely from the actual dynamic execution paths explored by the specific test scene, they could not in general be eliminated even by manual analysis of the code: indeed, a hand-coded application is likely to be more conservative in memory usage than our analysis allows. The stack usage represents an over-estimate of 23 words (or 92 bytes) of memory. While this is not large in absolute terms, we have found stack costs for other examples to be accurate to

within 2.5% [4]. The larger percentage figure in this example almost certainly results from unexplored conditional paths.

Real-time embedded systems are a harsh execution environment. In order to demonstrate the resilience of the implementation and the accuracy of the cost analysis, the vehicle simulation was run continuously under RT-Linux for a period of 36 hours using our calculated memory settings. The program ran without any memory accesses outside the allocated area and without "leaking" memory: essential requirements for real-time control applications. Total dynamic memory usage (*including code, runtime stack, and runtime libraries*) was 105340 words (412KB) of memory, a level easily supported by single-board computers costing $70 or less (e.g. JK Microsystems' 33MHz 186-based FlashLite 186).

Our real-time test system used a surplus 450MHz Pentium II, a lower-performance processor than that used for the actual Cycab controller. Over our 36-hour test period, maximum execution times were $46\mu s$ for the *control* box, $9ms$ for the *env* box and $3ms$ for the *vehicle* box with a worst-case real-time response to the *control* input of $22.4\mu s$, and worst-case real-time responses to the primary vehicle inputs of $31.2\mu s$. Such figures suggest that even with the current lightly optimised implementation, Hume can be a practical programming language for real-time embedded systems.

We now plan to enhance the work described here by successively introducing simulations of 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 intend to adapt the simulation to deal with real-world interrupt-driven vision data and to mount our application on the CyCab itself. This will involve interfacing to low-level interrupt drivers, and dealing with hard real-time costing issues in addition to the space issues that we have considered here. We believe that our costing approach should be extensible to these situations without major technical complications.

8. CONCLUSION

This paper has introduced the Hume programming language and surveyed its general properties as a layered language. We have focused on the FSM-Hume layer, which is restricted to bounded automata, discussing the implementation and behaviour of a simple autonomous vehicle simulation written in FSM-Hume. We have demonstrated that the space behaviour of FSM-Hume programs can be predicted accurately, and that such programs can be run robustly in embedded systems environments without fear of space leaks or unintended memory accesses. We conclude that FSM-Hume is both highly expressive and allows the determination of strong formal properties concerning space behaviour.

We are continuing to explore the implementation and analysis of Hume layers. Our current focus is the extension of our work to provide cost analyses for primitive recursive definitions and data structures, thereby increasing expressibility for the programmer. This work builds on recently achieved theoretical results on automatic cost analysis for recursive programs [6]. Our experience suggests that entirely accurate costs can be determined for a range of primitive recursive definitions. However, in cases where accurate costs cannot be determined, it may be necessary to assume infinite cost, a situation that cannot arise with FSM-Hume. Accurate costing in the presence of primitive recursion may thus require more careful programming than is necessary with the FSM-Hume layer.

Acknowledgments

This work was partly supported by EPSRC grant GR/R 70545/01 and by a travel grant from the French CNRS.

9. REFERENCES

[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: A tool for modelling and implementation of embedded systems. In *Proc. 8*th *Intl. Conf. on Tools and Algs. for the Construction and Analysis of Systems*, pages 460–464. Springer-Verlag LNCS 2280, 2002.

[2] G. Berry. The Foundations of Esterel. In *Proof, Language, and Interaction*. MIT Press, 2000.

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

[4] 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*. Springer-Verlag LNCS 2670, Sep. 2002.

[5] 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*, pages 37–56. Springer-Verlag LNCS 2830, Sep. 2003.

[6] K. Hammond and P. Vasconcelos. Inferring Costs for Recursive, Polymorphic and Higher-Order Programs. In *submitted to 2004 European Symp. on Programming (ESOP 2004)*.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[8] M. Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming (ESOP 2000)*. Springer-Verlag LNCS, 2000.

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

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

[11] T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs Using Uppaal. In *Proc. IEEE ICDCS Intl. Workshop on Distributed Systems Verification and Validation*, pages E15–E22, 2000.

[12] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 Feb. 1997.

[13] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Proc. 4th. Intl. Symp. on Practical Aspects of Declarative Languages*. ACM, Jan 2002.