# Chapter 1

# Rate Analysis and Deadlock Detection for HUME

Robert Pointon[1]

***Abstract:*** HUME is a domain-specific programming language targeting resource-bounded computations, such as real-time embedded systems. Rate analysis is a novel algorithm for HUME that determines the relative rates (i.e. how often events occur, or components need to run) within the whole system, and as such the algorithm possess the following useful properties:

- Deadlock Detection – of *some* cases.
- Scheduling – suggesting priority of components and possible parallelism.
- Static – can be performed before running the program.
- Cheap – polynomial algorithm.

Finally for completeness, runtime deadlock detection within HUME also turns out to be very cheap and the algorithm is given is well.

## 1.1 INTRODUCTION

HUME has been well described in other papers ([MHS03]) of this workshop, but in brief summary it is based on a generalised concurrent automata comprising of multiple concurrent *boxes* connected by point-to-point *wires*.

### 1.1.1 HUME Boxes and Wires

A wire is a single value buffer. If empty a value may be asserted into the wire thus changing it's state to full, while if full a value can be consumed leaving the wire as empty. Each wire connects two boxes in a point-to-point manner.

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

A box is comprised of a set of patterns and upon firing each pattern has some functional transformation to perform. Each box loops through the following sequence of actions:

1. A box blocks until an input pattern matches.
2. It *consumes* values from a subset of the input wires.
3. Some functional evaluation takes place to produce a result for a subset of the output wires.
4. The box blocks until all the required output wires are empty.
5. The box *asserts* it's output values into the wires.

### 1.1.2 Motivation

My personal interest in HUME is applying it to music or signal processing situations. For this it is important to know that data flows at a constant rate through the system, that the output is being asserted at the same rate as input is consumed, and that deadlock is avoided. It may sound obvious, but you don't usually want to record sound from a microphone and then hear it being played at twice the original frequency ('chipmunk' music is hopefully a forgotten pop fashion).

## 1.2 RATE ANALYSIS

Many analysis's attempt to find absolute values, for example; by slightly restricting the functional code and types within HUME leads to resource-bounded computations ([Ham03]), that is, time and space behavior can be predicted. In contrast the rate analysis is about *relative* values – do some boxes need to run at the same rate, or should some boxes run faster than others?

The algorithm has been derived by intuition and subtle interpretation of the coordination of HUME box/wire interactions, as a result the algorithm currently exist with an informal description.

### 1.2.1 Rules

By considering a single box, the following situations result in a set of it's wires transferring values at the same rate:

1. Values from several input wires are always consumed simultaneously as a result of all pattern matches.
2. Values directed to some outputs are always asserted simultaneously whenever evaluation is complete.
3. An output is always asserted as a result of a particular input being consumed.

Considering these situations it becomes clear that there is no distinction between input and output and that it is only necessary to determine if the state of a wire is altered by a consume or an assert. By identifying for each input/output wire if it is altered or not (or is non-determinable) simultaneously or as a consequence of another wire of that box changing, then wires can be grouped into sets that run at the same rate (a *rate set* seems an appropriate name). This can be seen from the example of Figure 1.1*a*.

Each input wire is either unchanged, altered, or is is non-determinable, and so this changeability can be assigned tri-state values with: 0=unchanged; 1=altered; the third state being unknown. It is the equivalence of columns in the table within Figure 1.1 that defines equivalent rates, more formally: for two columns $x$ and $y$, then $x = y$ if $x_i = y_i, \forall i$.

The rules can then be extended to give *relationships* between the rate sets of a box:

1. If a wire is altered for a subset of the cases that another wire is altered, then it must be acting at a less-than or equal rate (see Figure 1.1*b*). More formally, a subset of is defined as: for two columns $x$ and $y$, then $x \subset y$ if $x_i <= y_i, \forall i$.

2. If groups of the above subsets of wire are mutually exclusive and can union to make up another set of wires then the rates must sum (see Figure 1.2*a*).

Now considering the box rather than the wire, then if a wire is altered in all of a patterns of a box then that box must be running at the same rate as that wire (see Figure 1.2*b*). Alternatively the rate of a box must be greater-than or equal-to the rate of each and every of it's wires (see Figure 1.1*c*).

Finally, as wires are connected in a point-to-point manner and can hold only one value, the output of a wire must be consumed at the same rate as input is asserted into it. This final rule allows the combining of the rate sets from the connected boxes.

### 1.2.2 Deadlock Detection

Deadlock is defined as a state of the system which does not have any successor states, in HUME deadlock either occurs when a box is waiting for input to arrive, or waiting for an output to become empty. At the input of a box, deadlock occurs when it is no longer able to match any input pattern, that is; the box stops consuming from the input wires either because it's input patterns are incomplete (1), or a wire is no longer being asserted into (2). At the box output, deadlock occurs when an output can no longer become empty, that is; a box cannot assert on it's output wires because a wire is no longer being consumed from (3).

Case 1 requires some runtime detection whereas cases 2 & 3 are the result of the consume/assert to a wire acting at different rates. Therefore by assuming that both ends of a wire act at the same rate, then once the wiring between boxes has been taken into account, it is the contradictions in the rates of wires that indicate possible deadlock. This is illustrated in Figure 1.3 where a box that produces two outputs for every input is connected back upon itself.

```
box operator
  in  (s::integer, meth::STATE, carbmon::STATE, airflow::STATE)
  out (s'::integer, control::STATE, action::STATE)
match
  (*,*,*,ON) -> (*,OFF,OFF) |
  (*,*,ON,*) -> (*,OFF,OFF) |
  (*,ON,*,*) -> (*,OFF,OFF) |
  (s,*,*,*)  -> let s' = rand s
                in case s' mod OPPROB of
                  0 -> (s',ON,ON)   |
                  1 -> (s',OFF,OFF) |
                  _ -> (s',*,*);
```

Can be mapped to the following:

```
s meth carbmon airflow -> s' control action
0    0    0    1           0    1    1
0    0    1    0           0    1    1     where ['1' -> assert/consume    ]
0    1    0    0           0    1    1           ['0' -> no assert/consume]
1    0    0    0           1    ?    ?           ['?' -> not determinable ]
```

By comparing the columns it is clear that:

$rate_{wire}(\texttt{s}) = rate_{wire}(\texttt{s'})$ $^{(a)}$

$rate_{wire}(\texttt{control}) >= rate_{wire}(x), \forall x \in \{\texttt{meth}, \texttt{carbmon}, \texttt{airflow}\}$ $^{(b)}$

$rate_{wire}(\texttt{action}) >= rate_{wire}(x), \forall x \in \{\texttt{meth}, \texttt{carbmon}, \texttt{airflow}\}$

$rate_{box}(\texttt{operator}) >= rate_{wire}(x), \forall x \in \{\texttt{s}, \texttt{meth}, \texttt{carbmon}, \texttt{airflow}, \texttt{s'}, \texttt{control}, \texttt{action}\}$ $^{(c)}$

**FIGURE 1.1. Operator Box Example**

The final lines of Figure 1.3 show that the wiring insists that the output cannot run at greater rate than the input as is required by the box rate sets. If one rate set cannot be a superset of another, then a possibility is that certain patterns can never fire, for example; if the `(*, Just x) -> (x, Nothing)` pattern of the double box can be removed then rate analysis will find no contradictions. The deadlock detection problem then becomes one of determining if these removed patterns were reachable from the initial system state, but in reality that fact that the rate analysis finds a contradiction should be sufficent warning to the programmer to check their code.

## 1.3 IMPLEMENTATION

Currently the algorithm is very naive and partially incomplete in that it doesn't handle recursion, nor does it deal with the case where rates sum together. The reason that the algorithm still works stems from the fact that, as was noted in Section 1.2.1, tri-state logic is necessary to describe if a wire it altered, unchanged, or is non-determinable. So when in doub (or through algorithmic incompleteness) the algorithm is free to return non-determinable.

The algorithm proceeds as follows:

```
-- determine the box rate sets and relationships
for each box
   -- extract the input/output changed matrix (as shown in the examples)
   for each pattern
       scan the pattern to determine if each input is consumed or not
       scan the function static call graph to see if each output is asserted or not

   -- extract the box rate sets
```

```
box merge
  in  (i::integer, j::integer)
  out (o::integer)
fair
  (x,*) -> x |
  (*,y) -> y;
```

Leads to:

```
i   j -> o
1   0    1
0   1    1
```

Thus:
$rate_{wire}(\texttt{o}) = rate_{wire}(\texttt{i}) + rate_{wire}(\texttt{j})$ (a)
$rate_{box}(\texttt{merge}) = rate_{wire}(\texttt{o})$ (b)

**FIGURE 1.2. Merge Box Example**

```
box double
  in  (i::integer, s::Maybe integer)
  out (o::integer, s'::Maybe integer)
match
  (*, Just x) -> (x, Nothing) |
  (x, Nothing) -> (x, Just x);

wire double
  (double.o initially 0, double.s')
  (double.i, double.s);
```

Leads to:

```
i   s -> o  s'
0   1    1  1
1   1    1  1
```

Thus considering the box only:
$rate_{wire}(\texttt{s}) = rate_{wire}(\texttt{s'}) = rate_{wire}(\texttt{o}) = rate_{box}(\texttt{double})$
$rate_{wire}(\texttt{o}) >= rate_{wire}(\texttt{i})$
Yet the wiring then states that the following MUST hold:
$rate_{wire}(\texttt{o}) = rate_{wire}(\texttt{i})$

**FIGURE 1.3. Deadlocked Example**

```
   for each matrix column
       combine with other equal columns

   -- extract the relationships between the rate sets
   for each rate set
       check if its column was a subset of the others

-- determine the system rate sets and relationships
for each wire
   combine the rate sets that the wire connects
```

From this algorithm pseudo-code it should be clear that the algorithm is polynomial in the number of boxes.

## 1.4 DETAILED EXAMPLE

The main drainage control system has been used as an exemplar in previous HUME papers [MH02], and so it is used again here to illustrate rate analysis.

In summary there are nine boxes: the *pump* box implements the pump of the drainage system; *supervisor* and *operator* simulate human interaction with the system; *environ* provides the mine environment in terms of *airflow*, *methane*, and *carbonmonoxide* simulations; *water* simulates the water in the mine; finally *logger* records what happens.

On running the rate analysis program the following style of output fragment is produced:

```
Wired rate sets: 24
24: ["environ.meth_op_alarm","operator.meth_op_alarm"]
...
7: ["pump.meth_request","environ.meth_request","environ.meth_reply",
     "pump.meth_reply"]
6: ["#BOX.supervisor","supervisor.s","supervisor.s’"]
5: ["water.highlow","pump.highlow"]
4: ["#BOX.logger","#STREAM.output","logger.log"]
3: ["#BOX.water","water.wl","water.wl’","water.log_level","logger.log_level",
     "logger.log_highlow","water.log_highlow"]
... etc

Rate relations:
16 >= [9,24,19,7]
14 >= [10,23]
8 >= [11,22]
4 >= [3,13,12,11,10,9,8,14,16,20]
... etc
```

The first group of data enumerates the 24 distinct rate sets. The members of each set are listed as either #BOX.<boxname>, #STREAM.<streamname>, or <box/streamname>.<portname>. Within each set all members will run at exactly the same rate. The second group of data identifies the relationships between the rate sets, for example; the first line states that rate set 16 runs at greater or equal rate to each of sets 9, 24, 19 and 7.

To simplify illustration the example is continued by focusing only on the rate sets of more than two members, or those involving the boxes. Then by plotting the rate sets only as in Figure 1.4 it can be seen that most of the components run quite independently at their own rate. Notice that the rate analaysis can group wires and boxes together, or any combination of. Also of note is the detection of the request-acknowledge handshaking for methane information between pump and environ boxes, and that the output stream and logger box run at the same rate.

In Figure 1.5 the rate relationship information is displayed where the upper components run at a greater or equal rate to the lower components that connect to them, for example; it is clear that the output/logger runs at a greater or equal rate than methane, carbonmonoxide, airflow, and water – an unsurprising result as logger needs to keep up with most of the other boxes so that it is ready to read their status.

The fact that there are so many rate sets (24 in this example) and that there are so few useful relationships is a consequence of the incompleteness of the current algorithm. A more complete implementation (using ‘pen & paper’) that takes into account the user function calls, gives 17 rate sets. In summary the consequences of this to the rate relationships are: operator is at greater than or equal rates to carbonmonoxide and to water; that pump is at greater than or equal rate to environ-pump; and finally that supervisor continues to be independent.

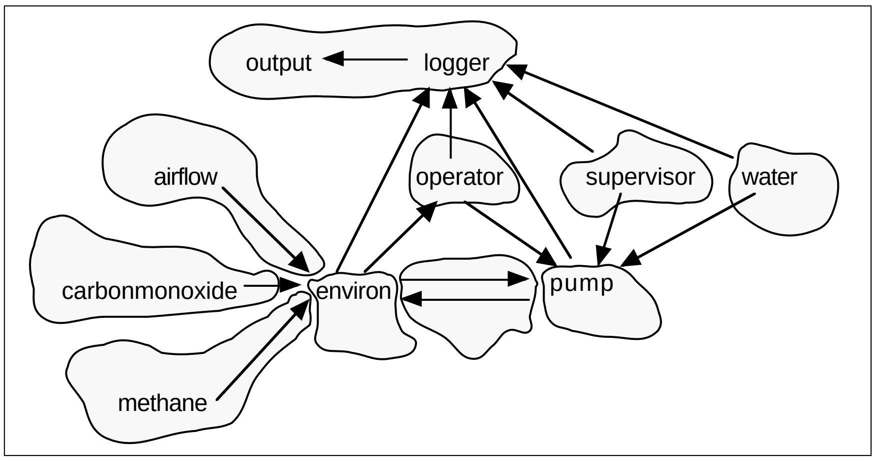

**FIGURE 1.4. Mine Drainage Rate Sets**

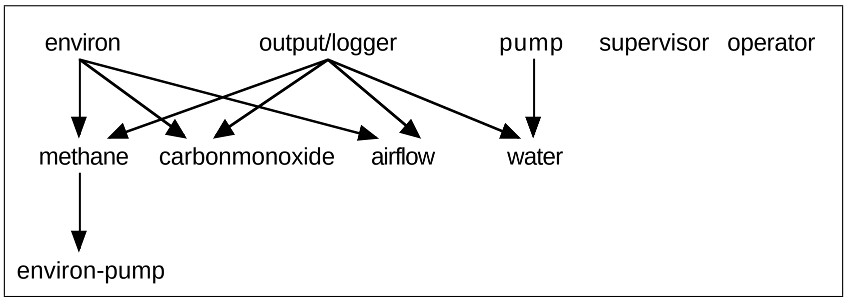

**FIGURE 1.5. Mine Drainage Rate Relationships**

## 1.5 DYNAMIC DEADLOCK DETECTION

In Section 1.2.2 it was mention that dynamic deadlock detection was necessary for some cases. The exact cases are straightforward to enumerate:

1. When a box has all its inputs full, yet no patterns match, then it must deadlock. At this point the box is *terminated* and all connecting wires are *closed*.

2. A box that attempts to assert (or is asserting and thus its output is lost) to a closed wire must also deadlock.

3. For a box to attempt to consume from a closed wire results in the input being treated as if it was full for the deadlock detection purposes.

4. When an input stream closes then its output wire is closed.

Detecting deadlock of the entire system would require some central monitor, an alternative is to allow deadlock information to propogate through the system via the normal communication mechanisms (the wires). In some desirable cases the decentralised approach leads to the notion of controlled program termination, for example; when an input stream closes then there is often a cascaded termination of boxes until there are no boxes left running and so the system stops gracefully. Note that a box does not automatically terminate because its neighbor deadlocks, instead the box will continue until it too has deadlocked – from experience this typically results in values in buffers being flushed through the system rather than being lost.

Implementation proves to be relatively trivial and the additional code necessary for the runtime system is minor as follows:

- When no patterns match in a box then it needs to do the additional trivial check to see if all inputs are full (taking into account any wires that are closed).

- Before (or while) asserting into a wire, there need to be the additional check to test if the wire closed.

- Additional code is needed to terminate the box and mark wires as closed.

- Depending on the execution model, extra events may be needed to wake-up the connected neighbors of a terminated box so that they can check if they too will deadlock.

Initial results using these customisations seem promising, and allow the construction of HUME applications that can do one-off processing of data rather than the more usual non-terminating server applications. *Examples to follow at workshop and in final paper.*

## 1.6 DISCUSSION

This paper has outlined and explained along with example programs and initial results, a novel algorithm for HUME that is a static analysis to assist in the verification of program behavior. This rate analysis algorithm is very cheap and as a consequence is much more scaleable in contrast to many related algorithms involving deadlock detection which require simulation or exhaustive state search.

### 1.6.1 Deadlock Detection

Partial static deadlock detection comes from the rate analysis, though perhaps more important is that the insight from developing the rate analysis has given birth to cheap runtime deadlock detection. Furthermore this runtime deadlock detection can lead to graceful program termination in desirable circumstances.

### 1.6.2 Runtime Optimisation

Once the relative rates between all the wires and boxes have established, then the relationship between the boxes can be used to determine which boxes are more active thus aiding in the generation of priorities for a schedular. Boxes which are at the same rate (and thus with interconnected wires at the same rate), could be either composed into a single monolithic box, or maybe they can be executed in parallel lock-step style.

### 1.6.3 Programmer Aid

By presenting this information graphically in the HUME box/wiring editor then the programmer gains vital feedback about how busy various parts of the system are. In terms of application development the algorithm derives the obvious in many cases, so the programmer can quickly understand if the 'obvious' conflicts with their mental model.

### 1.6.4 Future Work

Currently the rate analysis algorithm exists in a very naive state and its use has not been fully explored.

- Can the notion of 'rate' be applied to other computing problems?
- A formal description of the algorithm is desirable.
- The direction of flow through wires is ignored – an intriguing property of the algorithm.
- Whether a box is fair or unfair is ignored.
- Exceptions are not handled – the set of exceptions that each output can raise would need to be determined, and then the outputs of the exceptions combined with the function output.
- Timeouts on input/outputs have not considered for this algorithm.

## REFERENCES

[Ham03] K. Hammond. An abstract machine for resource-bounded computations in HUME. In *Draft Proceedings of Implementation of Functional Languages*, Edinburgh, Scotland, 2003.

[MH02] G. Michaelson and K. Hammond. The pump. In *somebook/techreport*, somewhere, 2002.

[MHS03] G. Michaelson, K. Hammond, and J Serot. The finite state-ness of FSM-HUME. In *Draft Proceedings of Trends in Functional Programming*, Edinburgh, Scotland, 2003.