

# Resource Analysis for Hume Box Compositions

Christoph Herrmann and Kevin Hammond  
University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SX

## Abstract

*Hume is a very high-level language for programming resource-critical systems, such as those found in autonomous vehicle control. In Hume, functional boxes are composed to form an asynchronous system of processes. This paper presents a compositional analysis method which calculates the resource consumption of a complete Hume system based on that of individual boxes, and which distinguishes particular program situations. In this paper, we focus on the resource of worst-case execution time. The symbolic treatment that we use allows us to express properties in terms of control parameters. It delivers parameterised formulae which allows Waterfall Solutions' Cerberus system to reconfigure the AV control algorithms so as to guarantee successful mission completion and optimisation of search patterns.*

Keywords: compositional analysis, resource-driven reconfiguration, AV control, embedded systems, worst-case execution time.

## Introduction

This paper describes work that has been carried out in the context of an inter-theme project between Waterfall Solutions Ltd. (SER015) and the University of St Andrews (SEN018) within the Systems Engineering for Autonomous Systems Defence Technology Centre (SEAS DTC). The purpose of this project is to integrate certified software written in the resource-critical Hume language [8] (St Andrews) with the Cerberus framework (Waterfall) for simulating reconfigurable systems. We provide resource models and analyses for Hume programs that are then used by Cerberus for overall mission planning. The combination will provide advanced resource-based mission-planning capabilities [3] for AV platforms.

The main contribution of this paper is to describe a new analysis method for determining the resource consumption of a system of cooperating boxes at the Hume coordination level, assuming that we already know the resource consumption for individual boxes (this can be obtained

using e.g. the amortised analysis we described as part of SEN002 [12]).

An outline of our analysis process is shown in Figure 1.

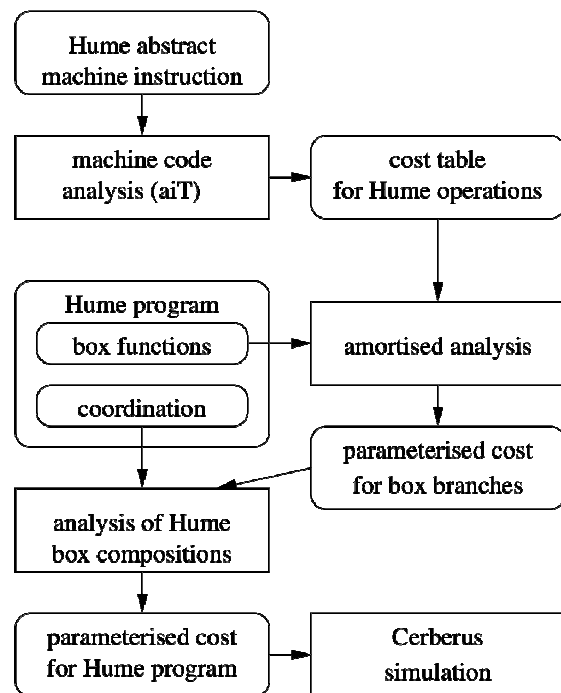


Figure 1: Analysis process

Hume programs are first translated into Hume abstract machine (HAM) code, which is then translated further into native machine code. For each HAM instruction, such as basic integer or floating-point operations, function calls, etc., we obtain primitive resource usage information by measurement / analysis of the underlying machine instructions (for worst-case execution time, WCET, we use AbsInt GmbH's high-quality **aiT** tool [1]). Having done this, we obtain a cost table for each HAM instruction that is parameterised where necessary, e.g. on the sizes of vectors for primitive vector operations. This cost table is machine-dependent, but, unlike previous approaches, has to be constructed only once for each new target architecture configuration.

Hume programs consist of a set of functional *boxes*, mapping some inputs to some outputs. These are wired into a static process network using coordination primitives. The resource usage analysis of a Hume program then takes information about the connections between boxes and the circumstances under which they are activated and calculates information about which boxes are executed, and which branches are chosen. By combining this information with the parameterised cost for each branch, we can then obtain the resource usage cost for an entire Hume program. Finally, the parameterised cost formulae for WCET, memory consumption etc. is used in Cerberus to simulate the effect of reconfiguration decisions and to provide prior knowledge of resource consumption for different choices of mission-planning algorithm, such as the Flocking algorithm we will use in this paper to illustrate our approach.

Hume programs consist of a set of cooperating boxes which exchange data through statically-connected wires, each connecting an output port of one box with an input port of another (perhaps the same) box. For example, Figure 2 shows a

system of two boxes A and B connected using wires x, y and z.

#### Compositional Analysis

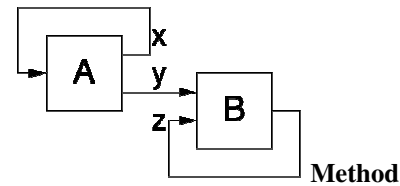


Figure 2: Composition of two boxes

Hume programs consist of a set of cooperating boxes which exchange data through statically-connected wires, each connecting an output port of one box with an input port of another (perhaps the same) box. For example, Figure 2 shows a system of two boxes A and B connected using wires x, y and z.

The behaviour of each box in a Hume program is defined by an ordered set of computational rules. Each rule tests certain input ports for the availability of data or the values of such data. If the test succeeds, then the computation associated with the rule is performed, the input values are removed from the associated wires and some new output values are written to some output wires of the box. If the test fails for this rule, it is the turn of the next rule, and so on.

The starting point for the compositional analysis is a static network of Hume boxes. For each rule in a box, we need to know the (possibly parameterised) worst-case resource usage plus an abstraction of the patterns that will cause the rule to be executed. In the sequential execution model that we have assumed for this paper, the boxes are activated in a round-robin fashion by a scheduler and executed if one of the test patterns matches the box inputs that are available. The chosen rule then determines the resource usage. If a set of inputs could be covered by more than one pattern, then we need to calculate the maximum cost of all the rules that could match.

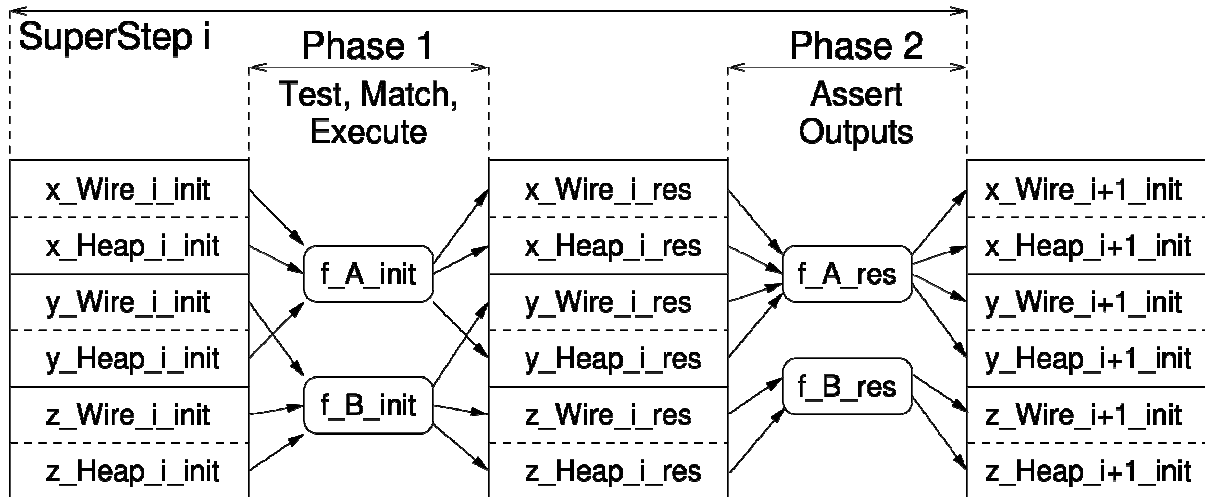


Figure 3: Superstep as a mapping between vectors of variables

The challenge for our compositional analysis is to determine whether a box is executed in a particular scheduling cycle and which pattern has matched. This requires an abstraction of the set of program values and an analysis of how a box maps input values to output values in the abstract domain, i.e., *abstract interpretation* [7]. An example that is commonly used to illustrate calculation in an *abstract domain* is determining the sign of a product from the sign of its factors, without performing the underlying multiplication. Here the abstract domain is the set  $\{-,0,+$  for negative numbers, zero and positive numbers, respectively.

In our case, we must deal with different abstractions for different values of the same type. Some integer numbers carry iteration counts that determine the frequency of box execution. Some wires carry data structures whose size is important for the analysis of individual rules, but which does not affect the compositional analysis other than in needing to propagate the costs. Other wires carry integers whose value is irrelevant to the analysis, so it is not an issue if their value is not available at analysis time.

Several different Hume operational semantics could be considered as an alternative to the round-robin schedule we

have discussed above. If the boxes need to cooperate to implement an algorithm, rather than acting independently, then the *superstep* semantics is preferred. Here, the computation proceeds in a series of scheduling steps (or supersteps). Several boxes may then be scheduled within each superstep, but each box is scheduled at most once. This allows a higher degree of concurrency since the execution of all boxes within a scheduling cycle is completely independent of each other.

In addition, since the boxes do not have a persistent internal state, the entire system state between two scheduling cycles is given by the valuation of the wire buffers. We can thus model the computation (on abstract values) and the resource consumption of a scheduling cycle as a function mapping a vector of wire values (and some other values) onto itself. Figure 3 shows how this mapping is expressed as a function for the box composition from Figure 2. In this diagram, each value is indexed by the wire name (e.g., x, y, z), whether it is on the wire or still in the previous box heap (Wire / Heap), by the index of the cycle (i, i+1,...) and whether it happens at the beginning of the superstep or before the assertion of results (init,res). The treatment of wire values is described in more detail in [10].

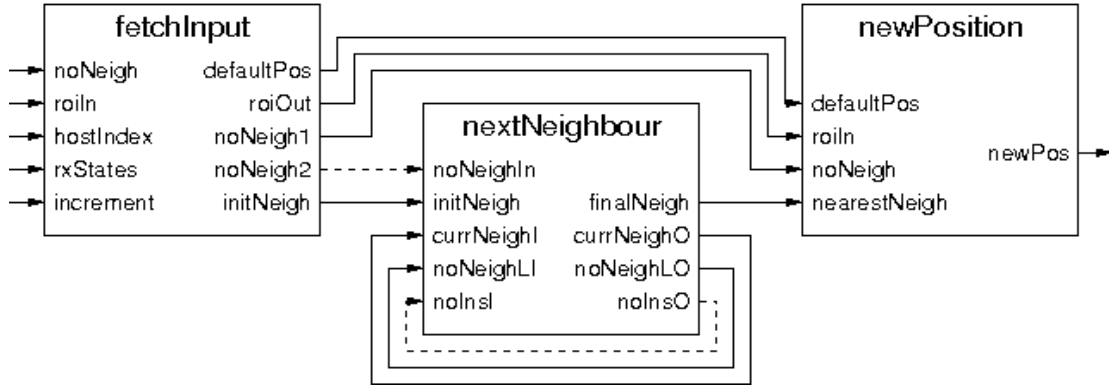


Figure 4: Box composition for the flocking algorithm mission computer

Using the idea of Figure 3, the entire work of a superstep can be expressed as a function on wire vectors, for interpretation with concrete values (*simulation, profiling*) as well as for abstract values (*analysis*). The potential unavailability of wire values is represented by an additional dedicated value in the abstract domain.

### Example: Flocking Algorithm

The flocking algorithm consists of a control and reconfiguration component implemented in Cerberus and several identical copies of mission computers for the autonomous vehicles; which differ however in their current location, speed vector, energy resources and other state properties. In a simulation, the state properties are stored within the Cerberus framework; we focus here on the work of the AV mission computer within one Cerberus simulation cycle, namely: (1) receiving the AV state and reduced information about the other AVs (here only their location) and computation of the default position; (2) calculation of the nearest neighbours; and (3) calculation of the new own desired position and sending this information back to the Cerberus environment. This work is achieved by three Hume boxes named `fetchInput`, `nextNeighbour` and `newPosition` which are wired as shown in Figure 4. We focus here on the worst-case execution time (as simulated real time) between the sending of new input by Cerberus and the

availability of the output. Thereby, we are using the results of the amortised analysis for each rule of each box. Since this analysis is restricted to linear expressions, we have to design our system such that each box incurs a linear cost. The box `nextNeighbour` is linear in the number of AV platforms. We iterate it in line with the number of nearest neighbours we would like to use to calculate the new position of the AV platform. This number is stored in the input / output variables (`noNeighLO/noNeighLI`) and initialised by `noNeighIn`. The total execution cost will then be a product of the number of platforms and the number of nearest neighbours, i.e., it is expressed by a *non-linear* formula.

The task of a single activation of the box `nextNeighbour` is to take a list `currNeighI` (initialised with `initNeigh` and fed back by `currNeighO`) with the AV platform positions in which the first `noInsI` elements are already the closest neighbours and to permute the remaining elements such that in the result the first `(noInsI+1)` elements are the closest neighbours. Since the scheduling of the boxes is transparent to the Hume program, we need to maintain a counter value for the number of box iterations which is initialised by `noNeighIn`, incremented in each iteration and fed back in the loop `noInsO/noInsI`.

This value, shown with dashed lines in Figure 4, plays a major role in the analysis. We show a small part of the Hume program below: a reduced specification of the box `nextNeighbour`. The type declarations for input and output ports have been abbreviated by (...). An asterisk (\*) denotes unavailable data if it appears at an input position and ignored output at an output position. Two rules are specified, separated by |. In the example program, the first rule matches if the first two input ports (appearing at the left-hand side of `->`) do not provide data but the other three do; we name this branch `loop` because it carries values between iterations. The three values come from the previous iterations of this box. The other rule (named `init`) deals with the first execution of the box when the first two input ports carry data and the other three not.

```

box nextNeighbour
in (...)
out(...)
match
(*,*,currNeighI,noNeighLI,noInsl)
-> if noNeighLI>noInsl
    then (*, next currNeighI noInsl,
          noNeighLI, noInsl+1)
    else (take noInsl currNeighI,
          *, *, *)
|(noNeighIn,initNeigh,*,*,*)
-> if noNeighIn>0
    then (*, next initN 0, noN,1)
    else ([],*, *,*);

```

The right hand-side of `->` specifies the computation of the output values for the particular rule, using the actual values of the variables of the left-hand side of the rule. Two functions of the Hume program are applied (whose definitions are not shown here): the function `next` moves the closest element in the rest of the neighbours list at the first position, thereby extending the prefix of the list by one more neighbour, and function `take`

removes the rest of the list in the last iteration to be passed to `box newPosition`.

## Modelling in Haskell

Haskell is a widely-used very high-level functional programming language. It is syntactically similar to Hume and subsumes the type system and execution model of Hume functions, which make it easy to embed Hume functions in Haskell. The extensions made by the Glasgow Haskell Compiler (ghc) [13] provide powerful features that enable symbolic computations in a type-safe manner. We have therefore chosen to model the compositions of Hume boxes in Haskell and specify manually which program values are of interest for the analysis (and which are therefore carried over into the abstract prototype of the composition) and which are ignored. The specification of each box consists of four parts which all depend on the input, precisely expressed symbolically in terms of descriptions of the input values: (1) a predicate which specifies the runnability of the box; (2) the input / output function of the box; (3) which inputs have been consumed by the chosen pattern (in Hume not all inputs present need to be consumed!); and (4) the worst-case execution time (as an example of resource usage). While (1) and (3) must precisely correspond to real behaviour, since otherwise the entire system would behave completely differently, (2) and (4) can be subject to different degrees of abstraction and focus on values of interest as far as it is a proper abstraction of the concrete system, i.e., the boxes / rules executed are the same.

The typed syntax of symbolic expressions is defined by a *generalised algebraic data type* (GADT) in ghc version 6.10.1 [13], e.g., the cost expression

$$\text{if } ((4 < x) \text{ and } y) \text{ then } 0 \text{ else } z+1$$

can be expressed as an element of a GADT as follows:

```
If ((C 4 :<: x) :&: y)
(C 0)
(z :+: C 1)
```

We have provided GADT constructors for constants C, addition (:+), comparison (:<:), logical and (:&:) and ternary case distinction: If. Haskell functions for evaluation and simplification (by term-rewriting) can then simply be defined by matching against these constructors. Type-correctness is established by the following GADT specification:

```
data Sym :: * -> * where
C    :: Rational -> Sym Rational
(:+) :: Sym Rational -> Sym Rational
      -> Sym Rational
(:<:) :: Sym Rational -> Sym Rational
      -> Sym Bool
(:&:) :: Sym Bool -> Sym Bool
      -> Sym Bool
If    :: Sym Bool -> Sym Rational
      -> Sym Rational -> Sym Rational
```

Although structural parameters can only take natural numbers, rational numbers (of arbitrary precision) are used here because they occur in intermediate formulae, e.g. in a cost-increasing iteration:

$$\sum_{i=1}^n i = \frac{1}{2}n + \frac{1}{2}n^2$$

To give an impression of how such expressions can be used in practice we show the abstract input / output function for the nextNeighbour box (in a compressed form with some omissions). Here the five formal parameter names are in the list to the left of (->) and the four result values are returned in a list generated after the in element-wise by the standard zipWith function.

```
\[noNeighIn,initNeigh,currNeighI,
noNeighLI,noInsl] ->
let
branch1a=[noValue,dc,noNeighLI,
noInsl:+:C 1]
branch1b=[dc,
noValue,noValue,noValue]
branch1
= zipWith (If (noInsl:<:noNeighLI))
branch1a
branch1b
branch2 = ...
in zipWith (If (avail noNeighI :&:
avail initNeigh))
branch2
branch1
```

We can see that the value of interest which decides about the number of iterations, namely noInsl is incremented by one in branch1a (noInsl :+: C 1) and output as the fourth component to be fed back into the box. The constant dc stands for don't care and the symbol noValue for the representation of non-availability (\*) in Hume.

### Analysis Results

We carried out the analysis in a symbolic fashion, i.e., all the conditions that decide about availability of values, box execution etc. are, depending on the values of structural parameters such as the number of neighbours of interest, part of our result formulae. We started the analysis with a constant number of  $n=2$  for the number of nearest neighbours. In the specification of the WCET of the boxes we have decided to use symbolic names for boxes and branches instead of concrete numerical values. The analysis result thus contains summands consisting of products of these names with their occurrence counts. We normalise this result in terms of a case distinction in which each case is a maximum of polynomials, but in this analysis run we obtain just a single polynomial: a linear expression in terms of

box activations. We present the integer coefficients of the formula in Table 1. For each number of scheduling cycles we obtain a different result until the computation is finished in Cycle 5.

**Table 1: Analysis results for  $n=2$**

| scheduling cycles  | 1 | 2 | 3 | 4 | 5 |
|--------------------|---|---|---|---|---|
| fetchInput         | 1 | 1 | 1 | 1 | 1 |
| nextNeighbour/init | 0 | 1 | 1 | 1 | 1 |
| nextNeighbour/loop | 0 | 0 | 1 | 2 | 2 |
| newPosition        | 0 | 0 | 0 | 0 | 1 |

We can see that in Cycle 2 only the init branch of the nextNeighbour box is executed whereas in Cycles 3–5 the loop branch as well.

Now, we can generalise and instead of assigning the desired number of nearest neighbours the constant 2 we assign it the free variable  $n$  as a symbolic program value. For a number of four scheduling cycles we obtain a formula distinguishing 3 cases ( $n=0$ ,  $n=1$  and  $n>1$ ) and for each case stating a sum of symbolic execution time counts. Table 2 shows the number of box / rule activation counts in each case. Values in parentheses indicate the general value as it would be if a sufficiently large number of scheduling cycles had been considered in the analysis.

**Table 2: Analysis with unknown value of  $n$**

| #neighbours        | $n=0$ | $n=1$ | $n>1$     |
|--------------------|-------|-------|-----------|
| fetchInput         | 1     | 1     | 1         |
| nextNeighbour/init | 1     | 1     | 1         |
| nextNeighbour/loop | 0     | 1     | 2 ( $n$ ) |
| newPosition        | 1     | 1     | 0 ( $1$ ) |

Generally for the required number of  $n+3$  scheduling cycles, this fits with our expectation that each box / rule is activated once except from nextNeighbour/loop which is activated  $n$  times.

We can now multiply the occurrence counts of these boxes / branches with the results in machine cycles obtained by the rule-level analysis to obtain a formula for the concrete WCET of each box. For the nextNeighbour box it turns out that the amortised analysis delivers a WCET in terms of the length of the list of the AV locations. The complete WCET then contains a product of the value of the control input  $n$  for the number of neighbours and the length of this list.

### Conclusions and Future Work

This paper has presented a new analysis approach for calculating worst-case resource usage for compositions of Hume boxes. We have applied the analysis to one of the Waterfall mission-control algorithms, namely the flocking algorithm, obtaining promising results that can now be fed back into the Cerberus framework to drive mission-planning / simulation results. We intend to do this in the near future, and also to apply the analysis to more complex mission planning scenarios, with the objective of constructing a mini-them demonstrator linking Cerberus with Hume.

Although this paper represents work in progress, we can already report several technical findings:

- We found that the hardest part of the analysis is the appropriate modelling of the Hume semantics. Expressing this in the language of a general-purpose constraint solver as we tried in previous work would be a tedious and error-prone process. An embedding of an abstract specification in Haskell is, however, straightforward. Starting from a simulator of a system of boxes, we have extended this system step-by-step by introducing attributes to the components of our system, by different instantiation of polymorphic types and by generalisation. In the current state, we can automatically extract a low-

level symbolic constraint system from a high-level specification of a Hume program, thereby already performing trivial simplifications like constant folding or normalisation of arithmetic terms to polynomial expressions.

- We found that Hume boxes are a useful mechanism for structuring programs. They were originally invented to separate asynchronous behaviour from purely functional specifications and with the superstep semantics provide a clear mathematical understanding of how the system will behave. However, they also allow us to modularise analysis and to separate analysis into a functional part and a part dealing solely with box compositions (hard enough in the presence of feedback loops). Furthermore, our compositional analysis is not restricted to software boxes but can also be applied to e.g. sensors or actuators in which some operations are cheap, such as asking for the status of a sensor, and others are expensive, such as sending a radar signal for the resource energy.

Finally, we have already spent some effort in simplification after each scheduling cycle, converting the expressions into a *normal form*, applying (simple) algebraic rules, and filtering under the condition that non-availability is irrelevant for structural parameters. Nevertheless the number of combinations of abstract program values in the presence of free variables seems to grow at least exponentially with the number of scheduling cycles. We are therefore considering how the programmer can specify the behaviour of an arbitrary cycle such that the task of the analysis can be reduced to verify all specified situations for an arbitrary, but *single* cycle under the specified constraints. We believe that the theory of dependent types [2] and its implementations and use for programming [5] may potentially be applicable for this

purpose, even if not in the standard way, and we intend to explore this in future.

## References

- [1] AbsInt GmbH, *aiT Worst-Case Execution Time Analyzers*, <http://www.absint.com/ait/>
- [2] T. Altenkirch, C. McBride, and J. McKinna. *Why dependent types matter*, 2005. Submitted for publication.
- [3] C. Angell, D. Myatt, and M. Bernhardt. *Dynamic Reconfiguration in Autonomous Systems*. In this volume.
- [4] A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann. *Worst-case execution times for a purely functional language*. In Z. Horváth, V. Zsóka, and A. Butterfield, editors, *Implementation and Application of Functional Languages (IFL 2006)*, Lecture Notes in Computer Science 4449, pages 235–252. Springer-Verlag, 2007.
- [5] E. Brady, C.A. Herrmann, and K. Hammond. *Lightweight invariants with full dependent types*. In *Trends in Functional Programming*. Intellect, 2008.
- [6] D. Comaniciu and P. Meer. *Mean shift: A robust approach toward feature space analysis*. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, No.5, May 2002, pp.603–619.
- [7] P. Cousot and R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In *Proc. POPL 1977 — ACM Symposium on Principles of Programming Languages*, pages 238-252, 1977.
- [8] K. Hammond and G. Michaelson. *Hume: a domain-specific language for real-time embedded systems*. In *Proc. Intl. Conf. on Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science 2830, pages 37–56. Springer-Verlag, 2003.
- [9] C.A. Herrmann, A. Bonenfant, K. Hammond, S. Jost, H.-W. Loidl, and R. Pointon, *Automatic Amortised Worst-Case Execution Time Analysis*, in C. Rochage (Ed), *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
- [10] C.A. Herrmann and K. Hammond, *Towards Compositional Worst-Case Execution Time*

*Analysis for Hume Programs*, Proc. ERCIM/DECOS Workshop, Newcastle, UK 2008.

- [11] M. Hofmann and S. Jost. *Type-based amortised heap-space analysis (for an object-oriented language)*. In Peter Sestoft, editor, Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems, Lecture Notes in Computer Science 3924, pages 22–37. Springer-Verlag, 2006.
- [12] G. Michaelson, A. Wallace, K. Hammond, A. Bonenfant, Z. Chen, C.A. Herrmann, and B. Gorry. *Embedded Software for Autonomous Vehicle Control Using Optical Sensing: SEN002 Contributions and Futures*, Proceedings of SEAS/EMRS DTC Technical Conference, Edinburgh, June 2008.
- [13] The Glasgow Haskell Compiler V. 6.10.1. <http://www.haskell.org/ghc>

### **Acknowledgements**

The work reported in this paper was funded by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the UK Ministry of Defence.

We would like to thank our collaborators at Waterfall Solutions Ltd. and in the EU FP6 EmBounded project, IST-510255, especially Steffen Jost, Hans-Wolfgang Loidl, Norman Scaife, Edwin Brady and Greg Michaelson.