IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D27 (WP8b): Real-time Computer Vision Algorithms

Due date of deliverable: July 2007
Actual submission date: January 2008

Start date of project: 1st March 2005 Duration: 36 months

Lead contractor: LASMEA Revision: 1.5

Purpose: Describe the use of Hume to develop a computer vision algorithm for controlling a RobuCAB automated guided vehicle. The vision algorithm itself has been described in deliverable D07 [3], here we explain how this algorithm was adapted for a RobuCAB vehicle and used as input for a simple vehicle guidance system.

Results: We describe the construction of a vision-based control system for the RobuCAB electric vehicle. We use our Hume programming methodology as the basis for the construction of the two major Hume components: the lane-tracking algorithm already presented in D07 (WP8a) (here we reengineer the lane-tracking to match our methodology) and the messaging system used to control the vehicle itself. We provide a resource analysis of these components as far as the current analysis tools are capable. Finally, we present the construction of the completed system using a single box to glue the components together and discuss the performance of the complete system in the context of the EmBounded project.

Conclusion: We are not interested in how advanced or state-of-the-art the control system itself is, but we conclude that Hume is an ideal language for the construction of real-time systems at the level of complexity of this code. We are only partially able to verify the usefulness of resource analysis for this application since many parts of the code are outwith the capabilities of the current analysis despite the analysis being extremely advanced compared to other methods.

Project co-funded by the European Commission within the 6th Framework Programme (2002-06)			
Dissemination Level			
PU	Public		⋇
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Real-time Computer Vision Algorithms

Jocelyn Sérot <Jocelyn.Serot@lasmea.univ-bpclermont.fr>,
Norman Scaife <Norman.Scaife@lasmea.univ-bpclermont.fr>

Laboratoire LASMEA, Blaise Pascal University,
Les Cezeaux, F-63177 Aubiere cedex, France

Abstract

We describe how a lane-tracking algorithm developed under a vehicle simulation environment was adapted to control a RobuCAB electric vehicle. The lane tracking is a simplified version of the model-based lane-tracking method of Aufrère *et al.* [1]. This provides estimates of the vehicle's lateral position within a lane bounded by intensity dicontinuities at the lane edge and of road curvature. The vehicle itself is controlled via an internet socket by a message protocol specific to the RobuCAB vehicles. We describe the lane tracking algorithm and message protocol algorithms, both written in Hume. We also present an overall design for the control system which also includes a low-level program written in C to access the camera data and perform low-level image processing tasks plus a graphical monitor program for overseeing the execution of the system. The performance of Hume in terms of its ability to express complex control code for real-time system is evaluated.

1 Introduction

Deliverable D07 (WP8a) describes a series of small to intermediate-sized real-time and computer vision applications. As indicated in that document, we are pursuing our investigation into applications for the Hume language on three fronts; 1) small to intermediate vision applications to illustrate the expressivity of the Hume language, 2) hard real-time applications to verify Hume's usability for embedding on resource-limited architectures and 3) one large-scale application bringing all aspects of the Hume language into play. D07 reported our results for activities 1) and 2). Here we present the large-scale application which has been chosen to be vision-guided control of the RobuCAB[1] automated vehicle.

Since the RobuCAB vehicles represent a significant monetary investment[2] and due to insurance restrictions we developed the majority of the software used in this application using simulations. A simple simulation of a vehicle was presented in D07 and consists of a racing track arrangement built using the Raydium[3] games engine. The characteristics of the vehicle in the simulation, such as maximum speed and the height of camera mount-point, were matched to the RobuCAB as closely as possible. This simulation was then used to develop the lane-tracking component and subsequently modified to read data from the camera attached to the vehicle for the final implementation.

The RobuCAB itself is a four-seater vehicle (Figure 1) capable of speeds up to 30km/h and equipped with a range of sensors including cameras and various range-finding devices which can be used for collision detection.

[1]`http://www.robosoft.fr/eng/sous_categorie.php?id=1025`
[2]60k to 80k euros per vehicle in 2007
[3]`http://raydium.org/`

2 Objectives

In accord with LASMEA's overall goals in the EmBounded project:

1. **Test** the Hume compiler and analysis tools being developed elsewhere as part of EmBounded.
2. Verify the **effectiveness** of Hume in constructing embedded and real-time systems.

we have already addressed, in D07 (WP8a), the development of small to intermediate scale computer vision algorithms and have demonstrated the effectiveness of Hume for *small-scale* embedded applications such as PID control loops.

We have two main objectives in this deliverable:

1. Continue to evaluate Hume's effectiveness for constructing *medium- to large-scale* real-time systems (note, *not* embedded systems).
2. Demonstrate the use of Hume's resource analysis tools in the design and implementation of such systems.

To achieve these goals we use the RobuCAB electric vehicle as the development platform. Building on previous work we adapt the existing vision algorithm to guide the vehicle by generating suitable control inputs from the lane model and feeding this through a control subsystem which can communicate with the vehicle.

This requires substantial new Hume components, modified support code and needs to be constructed using software component system building methods. Simply building this system successfully will in itself meet the requirements of Objective 1. However, we go further than this and use the Hume programming methodology we have developed as part of EmBounded to show structured design of the two major subsystems.

We also attempt to meet Objective 2 by analysing our code with the resource analysis tools. Because of the tension between having code of respectable quality and complexity and using prototype analysis tools which require simplified applications we are only able to analyse some of the simpler subsystems. For these, however, we show that producing resource bounds is possible during the development process but, since this application is not embedded they are of limited use in the design process.

In Section 3 we describe the target platform. Following this, we outline the methodology in Section4 and then proceeed to show how our the two major components were constructed using the methodology in Section 5. In Section 6 we present the construction of the completed system. We conclude with a discussion of the completed system and the process used to build it.

3 The RobuCAB electric vehicle

3.1 Hardware architecture

Figure 2 shows the overall structure of the RobuCAB architecture from the point of view of an applications program. The wheels are independently controlled (steering, power and brake) but are attached in pairs to two low-level control systems. These are powered by a 555 PowerPC and are linked in a master-slave arrangement via a CAN bus. Other devices also reside on this bus such as the joystick and the low-level PC. The software running on the controllers allows various modes of operation, for example front-wheel steer only, four wheel steer where front and back wheels steer in opposite directions for tight cornering and a parking mode where front an back wheels steer in the same directions for easier parking. The controllers have various safety features built into them such as speed limiters and they will not allow damaging control operations such as pointing both front wheels in different

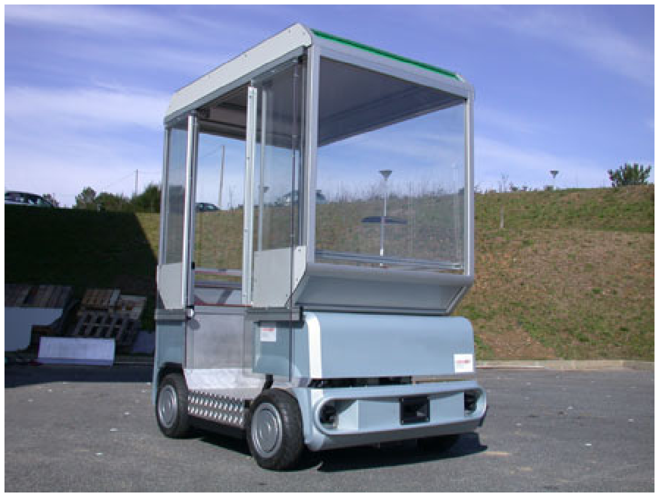


Figure 1: RobuCAB electric vehicle

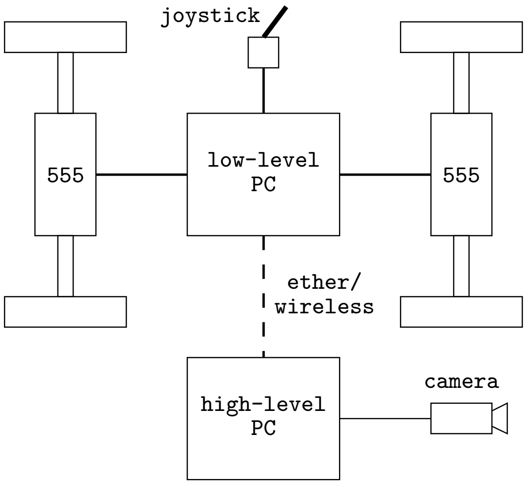


Figure 2: RobuCAB architecture

directions with the power on, or conflicting power commands such as having reverse power to the rear wheels with forward power to the front wheels.

In addition, there is a sophisticated dead-reckoning system implemented whereby the speed of each of the four wheels is controlled precisely to prevent slippage. This works differently for the different modes, for example the centre of reckoning in front-wheel steer mode is the centre of the rear axle whereas it is the geometric centre of the vehicle for four-wheel steering. This system is accurate to a small number of centimetres for normal road conditions.

Above the low-level controllers is the low-level PC. This runs a proprietary operating system and coordinates commands from the joystick and from external sources and sends commands to the low-level microcontrollers. In manual mode, the vehicle can be controlled directly by the joystick but in automatic mode the joystick data is available to the external program.

The low-level PC is also a server which can be accessed via ethernet or wireless, each RobuCAB having its own IP address. The vehicle can be controlled via commands sent down this link, the client program composes messages for sending to the RobuCAB either for sending commands or for requesting status information. The low-level PC responds with similarly structured messages.

User applications normally exist on an external PC, the high-level PC, which also hosts all the sensors. A camera is mounted on the RobuCAB and other sensors such as an infra-red collision detection system are also available. However, the camera is not part of the normal RobuCAB operating system although power for the camera is supplied by the RobuCAB's systems. In fact, there is a mains-voltage supply which can also be used to power the external PC. Under light usage, the vehicle can be autonomous for up to 6 hours.

3.2 Software architecture

The code to construct the messages between the low- and high-level PCs exists in a small C++ library developed at LASMEA and consists of about 200 lines of code. Originally, we had intended to develop a set of foreign function interface (FFI) calls to this library for controlling the RobuCAB from Hume but considering the relatively small size of this code, the availability of the source code and its relative simplicity, we decided to reimplement this library in Hume. This has two advantages. Firstly, the Hume FFI is relatively unwieldy and would require maintenance should either side of the interface change. Secondly, using pure Hume code to perform this function may allow the use of the Hume analysis tools in the development process. With the lane-tracking application already having been simplified to eliminate foreign function calls this would mean we could use pure Hume code for all the functionality on the RobuCAB application.

Currently, however, we retain the separate functions in the simulated environment of the synthesized camera data and the Hume lane tracking code. For the final RobuCAB application the simulation is replaced by a small C program which reads the camera data and performs the same low-level image processing computation as carried out by the simulation. This allowed a high reuse of code since very little reprogramming was required to develop this program from the original simulation. The camera is accessed using the Linux 1394 firewire suite of tools[4] with the Coriander CCD camera utility[5]. Additional support is provided by SDL library code which was adapted from the Coriander sources (itself being based on libSDL).

4 A Hume programming methodology: Costing by construction

We have developed a simple methodology for constructing Hume programs, incorporating cost analysis [2]. The design of the RobuCAB control system presented in this deliverable was developed in tandem with the methodology and was used as an exemplar for it. The design we present here is that which evolved alongside the methodology.

4.1 Summary of the methodology

We do not reproduce the methodology here, since this will be incorporated into deliverable D33 (Evaluation of the Hume language). We will, however, refer to the steps of the methodology in the methodology paper [2] and thus provide a brief summary.

Essentially, the novel feature of our methodology is the incorporation of costing analysis into the design process. In considering our methodology, it is important to remember that costing sometimes requires writing the code in a particular way which enables analysis. For example, where a system could be built into a single box with abstract code under more traditional methods, we may have to concretise the code and split large boxes into smaller ones to enable the resource analysis (which is on a box-by-box basis).

The methodology consists of three phases, a system-level phase where we first determine the overall characteristics of the system, a box-level phase where we specify the contents of each box and a valida-

[4] `http://sourceforge.net/projects/linux1394`

[5] `http://damien.douxchamps.net/ieee1394/coriander/index.php`

tion phase where we verify that the design meets the resource bounds criteria and attempt a structured re-design process if these criteria are not met. The methodology can be summarised as follows:

Phase 1: System-level Phase.

(**1**) Ascertain the overall resource requirements for the whole program.

(**2**) Consider external characteristics of the inputs/outputs.

(**3**) Define the box structure for the inputs and outputs.

(**4**) Based on (**2**), add timing constraints for each input and output.

(**5**) Outline the overall box structure and I/O box links.

(**6**) Refine box structure, if necessary.

(**7**) Define any required types and datatypes.

Phase 2: Box-level Phase.

(**8**) Specify the functionality of each box.

(**9**) Revisit the design, if necessary, looping back to Step (**5**).

(**10**) Implement the top-level functionality for each box.

(**11**) Implement functions that successively refine the functionality of each box.

(**12**) Apply costing. This must be done on a *whole-program* basis.

Phase 3: Validation Phase.

(**13**) Refine the functionality to respect costs. This may involve:

- returning to Step (**11**) and adapting functions so that they are more time/space efficient;
- returning to Step (**8**) and revising the box functionality with a similar objective; or
- refining the top-level box structure, Step (**6**), or modifying the overall box structure Step (**5**).

(**14**) If the constraints still can't be met:

- Relax margins on constraints, if this is acceptable.
- Revise the application specification.
- Update the hardware requirements.

(**15**) If it is still not posible to meet the costing requirements, then the application design criteria must be revised and the design process restarted, taking on board any relevant lessons from the original, failed design.

In this deliverable, we present our design and outline the process used to arrive at that design. For most of its development, however, the analysis tools were not available so we had to retrospectively apply the analysis to existing code requiring a small amount of re-engineering of the code.

Note also that the RobuCAB control system is a large program requiring features which the current version of the analysis is not able to handle. We therefore present the analysis applied to selected sub-systems which are capable of being analysed. It is relatively easy, however, to apply the methodology to code which is analysable and we show how our methodology would arrive at the design we implemented in practice.

5 RobuCAB control code

The overall design of the control system is presented in Figure 3. Items external to the Hume code are shown in dashed lines. There are five major components in the system:

1. The camera data and low-level image server `robucab_camera`.
2. The messaging system which talks to and controls the RobuCAB (the subsystem consisting of boxes `split` to `robucab_{in,out}`).
3. The lane-tracking module, `lane_track` and attendant protocol box for talking to the camera server, `camera`.
4. The TCL monitor program.
5. A central box which ties all the system components together, box `command`.

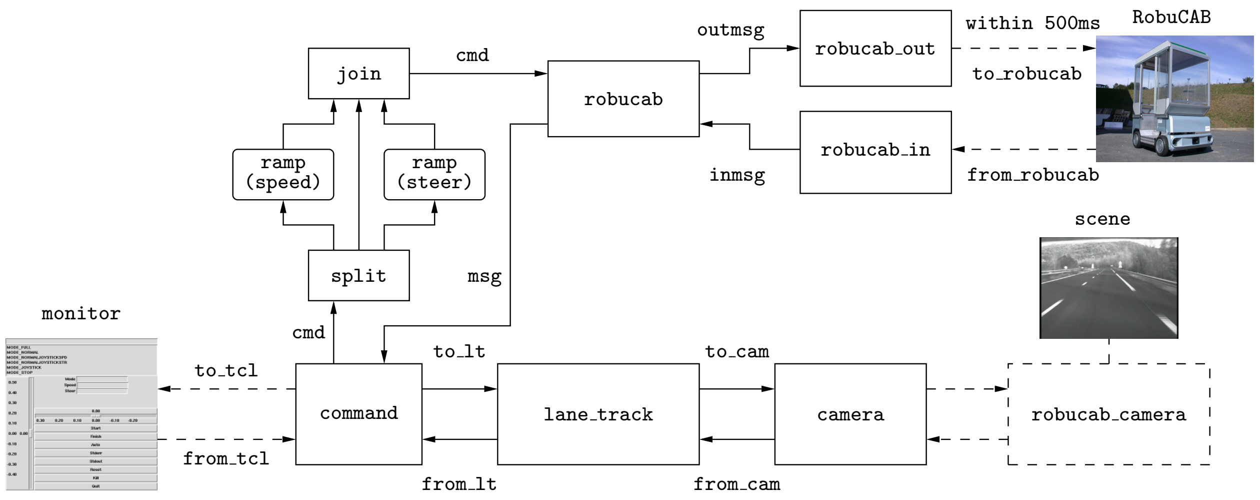


Figure 3: RobuCAB control system Hume boxes

There are also several multiplexors/demultiplexors for debug streams not shown in this figure. These can be switched on and off by the monitor program and produce detailed and verbose output which was used during development. The external components are not of interest here. We present the development of the messaging subsystem and the lane-tracking module and then demonstrate how these were built into a complete system.

5.1 RobuCAB link and control messages

Communications between the high-level user PC and the low-level vehicle monitoring and control PC are effected across an internet socket via an ethernet link. An option for the vehicle is wireless access but this option is not present on the RobuCABs used for this project. Each vehicle has its own IP address and responds to user messages on port 30000. Once the link has been established messages have to be constructed according to a predefined and *ad hoc* format (see section 5.1.1 below). Note that each connection to the vehicle has a priority and this priority is used to resolve conflicting messages from different sources. For example, the joystick has priority 2 so for external control we need a priority of 3 or higher to override the joystick commands. However, one option for the overall mode of the vehicle is to have, for example, joystick control of speed and external control of steering.

The original C++ library for vehicle access has classes for connections (`io_robucar`), messages (`RobucarCommand`) and for storing the current state of the vehicle (`RobucarData`). This last class is not a copy of the last message sent, it is possible for the vehicle to be in a different state than that implied by the last command request sent. An example of this might be if the collision detection sensor causes the vehicle to execute an emergency stop while the external code is still commanding forward movement. The link thus operates in a type of pseudo-asynchronous mode whereby a command message is sent to the vehicle telling it to continuously send updated status information (for example the current sensor data) until these messages are switched off with another command message.

There is also a “keep-alive” feature such that a message has to be sent to the vehicle down the link roughly every half-second or, for safety reasons, the link shuts down.

Note that there is no protection on the vehicle itself from sudden changes in steering or speed. In theory, we should not be able to send any commands to the machine which would damage it in any way. In practice, however, it would be useful to be able to attenuate the control values, partly to aid in debugging the code when it is actually controlling the vehicle and partly as a secondary safety level in case the control code generates any “noisy” outputs.

5.1.1 Message structure

The only major complication in the communications system is the complexity of the messages passed between the vehicle and the external control program. A message has the following structure:

```
<----magic number---->  <eid> <------msg size------>  <---crc---> <------msg body------>
0x4f, 0xea, 0x10, 0x98, 0xe3, 0x00, 0x00, 0x00, 0x04, 0x07, 0x70, 0x01, 0x02, 0x03, 0x04
```

The first four bytes are always the magic number `0x4fea1098`. The next byte `<eid>` is the endian ID code, `0xe3` for little endian and `0x8b` for big endian. The endianness applies to the remaining values in the header and to any values in the message body, from two byte integers to eight byte floats. Note that the vehicle itself can understand messages of both endianness but there is no way to define which one it uses for its own messages. It appears (however this is undocumented) to respond in the same endianness as the messages it receives. Strictly speaking, however, client code should also be able to decipher both types of messages which is a complication for Hume.

Following the endian ID code, there is a four-byte integer indicating the size of the message body, followed by a two byte CRC value for the message body. Both are sent in the endianness implied by the `<eid>` code.

The message body consists of one or more message blocks. Each block consists of a single-byte header code followed by a fixed format of message for that code. Some messages are for reception, some are for transmission. The blocks currently used in the Hume implementation for receiving messages from the vehicle server are (indicated here by the `#define` name of the header code byte used in the C++ header files):

`FRAMEID_PRIORITY_REPLY` is followed by a single 16-bit integer indicating the current priority of the link.

`FRAMEID_SENSORS_REPLY` is followed by a complex block of data from the current sensors. These include the current joystick values, the steering for front and back sets of wheels, the current speed and motor current for each wheel individually and the elapsed time since the last reset.

`FRAMEID_RCMD_REPLY` contains a single byte indicating the current mode plus the steering values and the speed values. In fact, this is actually a `RobucarCommand` structure with the additional header code byte prefixed. The speed value may be either a single 64-bit float for modes where the wheels are coupled or four individual floats for modes where the wheels are controlled independently.

For sending messages to the vehicle server:

`FRAMEID_PRIORITY_REQUEST_{0,1,P}` controls the sending of priority reply blocks. Note that 0 implies that sending of these blocks should be switched off, 1 means send a single one-shot message and P says to send the block continuously.

`FRAMEID_SENSORS_REQUEST_{0,1,P}` controls the `FRAMEID_SENSORS_REPLY` block.

`FRAMEID_RCMD_REQUEST_{0,1,P}` controls the `FRAMEID_RCMD_REPLY` block.

`FRAMEID_ACMD_REQUEST_{0,1,P}` controls the `FRAMEID_ACMD_REPLY` block.

`FRAMEID_PUTCOMMAND` is followed by a `RobucarCommand` block and indicates the current desired mode, speed and steering values. For relative control it is possible to simply echo the last received `FRAMEID_RCMD_REPLY` block with suitable increments or decrements.

`FRAMEID_CHANGEPRIORITY` contains a new, desired priority value for the link.

5.1.2 Design of the Hume RobuCAB link implementation

From the description of the operation of the link above, we therefore, in normal operation, need to be able to:

- arrange for the status messages to be started and stopped as required,
- decode the incoming messages and build correctly formatted messages for output,
- manage the status information being sent asynchronously by the vehicle and store it in a form accessible to the rest of the program,
- allow outgoing command messages to be inserted into the current message traffic,
- send a regular (possibly null) message at least every half second to prevent link shutdown and
- implement a safety mechanism preventing spikes from the control algorithm from stressing the vehicle hardware.

Hume's programming model is highly suitable for constructing such a system. There are, however, several possible ways of structuring such a program. Since the code is simple and concerns communications code, a principally box-based design is probably the most suitable.

Apart from the methodology itself, our experience with the Hume language to date has given us an additional set of guidelines for developing Hume code. For example, we have found that when writing code which has to communicate asynchronously with other processes it is almost always useful to have a Hume box attached to each input and each output. This is for several reasons:

- Such boxes act as buffers for outgoing/incoming information.
- Data items can be easily queued and/or prioritized in these buffers.
- As a result of the buffering, synchronisation is less of a problem between the Hume process and the external data source/sink.

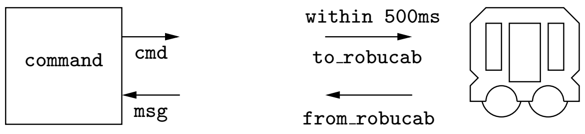


Figure 4: External I/O structure for messages

We have also found it important to separate the input and output boxes for systems with any significant complexity, even if the two resultant boxes require synchronisation or coordination. This results in much clearer code and less chance of allowing the two data streams to interfere with each other.

Philosophically, we are using Hume's asynchronous component to deal with the asynchronous world outside Hume and using the synchronous nature of boxes to clarify and simplify the functionality of these links.

5.1.3 Applying the methodology

The overall memory requirements, Step (**1**), are that the program must run within the available memory of the external PC that is used to control the RobuCAB.

The system inputs and outputs for the messaging component, Step (**2**), in our methodology, are:

1. to and from the RobuCAB;
2. to and from the central coordination box

For the messaging component, we only have direct links to the vehicle itself and to a central Hume box which contains the overall program logic. The central box (`command`) has links to all the other components.

Since there is a hard deadline of 500ms on outgoing communications, we must generate a message within 500ms, store incoming data, write-through any control commands and provide the current vehicle status asynchronously from the vehicle itself (Figure 4).

Step (**3**) is therefore very simple. We use a single box to communicate with the `command` box, but because of the complexity of the decoding/encoding function for the RobuCAB messages, we use separate boxes for input and output. Figure 5 shows the resultant structure, the `robucab` box contains any inter-stream logic and stores the vehicle state. The `robucab_{in,out}` boxes perform the encoding/decoding functions, ie. generate/check message headers, sizes and CRCs. Timing constraints are added to these boxes (Step (**4**)).

The overall box structure, Step (**5**), is complicated by the need to ramp the speed and steering values from the control algorithm. This logic could be incorporated into the `robucab` box but would entail an unwieldy number of patterns. Instead we make use of Hume templates, defining a `ramp` template which is instantiated twice, and place these instances between splitter/joiner boxes on the `cmd` stream. It also turned out to be neater to place a type conversion box on the `from_robucab` stream. The ramp templates are shown in Figure 6. Note that we have a bypass for non-ramped commands.

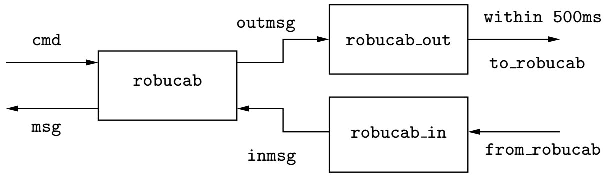


Figure 5: IO box structure for RobuCAB messages

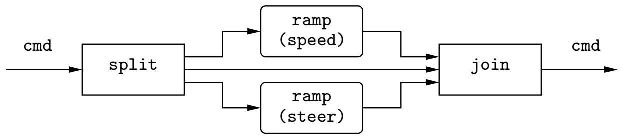


Figure 6: ramp templates

Box refinement, Step (**6**) is not necessary in this case. We need to take some care in selecting the data structures in Step (**7**). To enable size analysis we have to use a fixed data message size. Fortunately, the maximum size can be deduced from the documentation for the message contents. We then store messages as a pair of a fixed maximum-size message and a size indicating the number of bytes in the message. Finally, the vehicle can understand both little- and big-endian data and tags its messages with a code for endianness which we have to attach to incoming messages (normally this is in the message header but we strip the header upon reception). This gives us the following structures:

```
constant maxMsg = 260;
type Msg_t = vector maxMsg of uint8_t;

--                 message  size
data MsgDef_t = MD Msg_t    int16_t;

--                message  endian_code
data InMsg_t = IM MsgDef_t uint8_t;
```

Thus we arrive at our system-level characterisation with box headers and defined types.

Top-level function specification and implementation, Steps (**8**) and (**10**), are now straightforward. The `robucab_{in,out}` boxes are complex but well-defined encode/decode functions. The `ramp` template is best implemented as a set of `if-then-else` expressions forming part of a single function that is parameterised by direction. The `split` and `join` boxes test the `cmd` value (a structured data value containing speed, steering and other minor control data) and direct the output accordingly. The `robucab` box is mostly a write-through with some prioritisation of input over output and interleaving of I/Os. This box also generates a dummy message every 250ms if no outgoing message has been sent in that time. Since it thus has to share the real-time clock input with the `ramp` template instantiations, it follows that we need a multiplexor for the clock stream.

For the messaging component, there is no need to revise this simple design, Step (**9**), so we can proceed to functional refinement, Step (**11**), giving us our box-level system ready for validation.

In applying costing, Step (**12**), however, a complication arises in that our original version of the `robucab_in` box is too complex for the current analysis tools. We therefore need to rewrite the box using fewer function calls and a larger number of simpler patterns in the box match, Step (**13**). Having done this, we were able to meet both our timing and space requirements, Steps (**14**) and (**15**), and so complete both the refinement stage and therefore our entire design methodology.

5.1.4 Costing the communications component

Since our analysis tool is still in a prototype form, it is currently necessary to apply some minor code manipulations in order to obtain analysis results. For example, we need to add type headers for all functions (this is, of course, good engineering practice, anyway):

```
crc :: Msg_t -> int32_t -> uint16_t;
crc dat sz = ...
```

It also helps if we minimise the complexity of patterns in function and box definitions (this is something that can be easily be handled by a more sophisticated version of the analysis in future). For example, in the `robucab_in` box we could use a generic input method which counts the values that have been input:

```
match
 (w,(RIE Magic n enid size crc)) ->
  if n >= 1 && n <= 4 && w == (magic@n)
  then
   if n == 4
   then (*,(RIE Enid 1 enid size crc))
   else (*,(RIE Magic (n+1) enid size crc))
  else (*,robucab_in_state_init)
```

In order to facilitate the analysis, we rewrite the box on a one-pattern per input byte basis:

```
match
   (0x4f,(RIE Magic0 n enid size crc msg)) ->
    (*,(RIE Magic1 0 lID 0 0x0 zmsg))
 | (0xea,(RIE Magic1 n enid size crc msg)) ->
    (*,(RIE Magic2 0 lID 0 0x0 zmsg))
 | (0x10,(RIE Magic2 n enid size crc msg)) ->
    (*,(RIE Magic3 0 lID 0 0x0 zmsg))
 | (0x98,(RIE Magic3 n enid size crc msg)) ->
    (*,(RIE Enid   0 lID 0 0x0 zmsg))
```

In effect, we are trading more abstract and readable code for code which is more concrete but analysable. Currently, we are able to analyse the `robucab_in`, `robucab_out` boxes and the `ramp` template instantiations with associated `cmd_split` and `cmd_join` boxes (these are simple multiplexers with trivial costs). Because of its size, we are unable, however, to analyse the `robucab` box for worst case execution time using the prototype analysis. We present heap and stack analyses for this box, however.

Name	Resource	Cost expression
`robucab_in`	Heap	9.86*n+1473*n1+58*n2+90
	Stack	3+0.418301*n+58*n1+9
	Time	2667.75+232.466*n+20129.3*n1+8542*n2+4911
`robucab_out`	Heap	1010.12*n1+4*n3
	Stack	1010.06*n1+n2+5.5+15*n3+8
	Time	132+1011.31*n1+1428*n3+1543
`robucab`	Heap	1.40969*n1+997.899*n2+440.537*n3+18*n4
	Stack	1009.78*n1+21*n2+440.388*n3+15
	Time	N/A
`ramp`	Heap	653.608*n1+2*n2+4*n3
	Stack	653.562*n1+9
	Time	N/A

Table 1: Guaranteed Resource Bounds Obtained using our Prototype Analysis

Table 1 summarises the results for heap, stack and execution time usage for the components in the messaging subsystem, for which we currently have results. Although these resource bounds are not relevant to the current application, we have successfully generated guarantees which could be useful in an embedded context. The size expressions need to have the variables (n1, n2 etc.) instantiated for a particular architecture and input size giving concrete bounds for the application.

5.2 Model-based lane-tracking

We have already partially described the lane-tracking application in Deliverable D07 (WP8a): Real-time Testbed Applications. This is based on the lane-tracking algorithm by Aufrère *et al.* [1]. Here we reverse engineer the application to show how it could have been developed using our methodology and, in the next section, we present how this code was integrated into the RobuCAB control system. Note that this is not a specious rewrite of the code since the lane-tracking application was used as one of the prime motivations for the methodology.

5.2.1 Lane-tracking

The basic structure of the lane-tracking is a tree-search over the space of possible matches between detected edges and the regions within the image for which which edges can be expected. This terminates when a sufficient number of points have been detected. The camera input data is preprocessed by a module written in C (`robucab_camera` in Figure 3) which provides edges consistent with the current model. The model itself is managed in the Hume code, Kalman filters are used to update the model from detected data and used to provide estimates of the future position of the vehicle and therefore update the current set of interest zones (IZs) for the tree search.

5.2.2 Design of the Hume lane-tracking implementation

As explained in D07 (WP8a), downloading images into Hume proved to be prohibitively slow due to the nature of Hume's structured IO input so the image data itself is kept external to the Hume program which requests low-level processing of the image according to the current model on a server program which can host either:

1. a simulated OpenGL environment,
2. a replay of prerecorded image data or
3. from the camera mounted on the RobuCAB.

The low-level camera server exists in a separate process and we have a fairly complex protocol which we use to exchange data between this process and the main Hume code. Thus we require a dedicated box to run the protocol between the Hume program and the camera server. The lane-tracking computations, however, are very mathematical in nature and we prefer to implement this using Hume's expression language. In fact, we have a top-level box into which we have embedded the logic to perform the tree search. This box calls a complex set of Hume functions to perform the model updates. Thus we arrived at a Hume program consisting of two boxes. One to perform the tree search, the other to run the protocol to the camera server.

5.2.3 Applying the methodology

Step (**1**) is exactly the same as for the RobuCAB control messaging component. Here, the system inputs/outputs, Step (**2**), Figure 7 are:

1. to and from the low-level camera server;
2. to and from the central `command` box.

Step (**3**), as explained above, consists solely of two boxes, one for the tree search and one to talk to the camera server. The protocol with the camera server is a simple token-passing system. Although

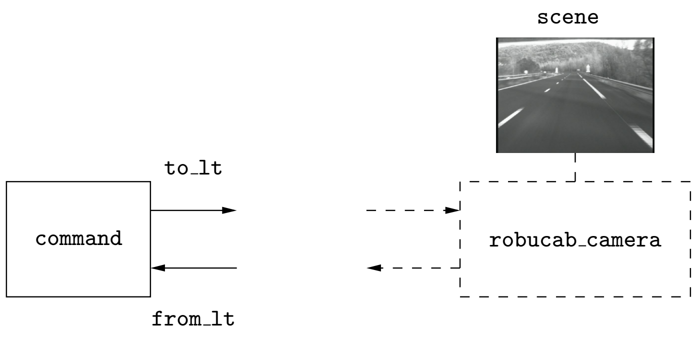


Figure 7: External IO structure for lane-tracking

the contents of the messages passing between the lane-tracking code and the camera server are very complex, the protocol itself is simple enough that separate boxes for input and output are not necessary.

While there is a theoretical time limit on the frequency of the information produced by the lane-tracking code for control purposes, this limit is complicated, firstly by the fact that it is difficult to define given that it depends on the road conditions at the time and secondly it is highly dependent upon the speed of the vehicle. Currently, Hume has no provision for dynamic deadlines like this so we do not attempt to define any time constraints in this design, Step (**4**). We have also decided, based on the structure of the system specification, that the main box, `lane_track`, will be principally implemented as functions owing to its mathematical nature so the overall box structure, Step (**5**), Figure 8, is two Hume boxes.

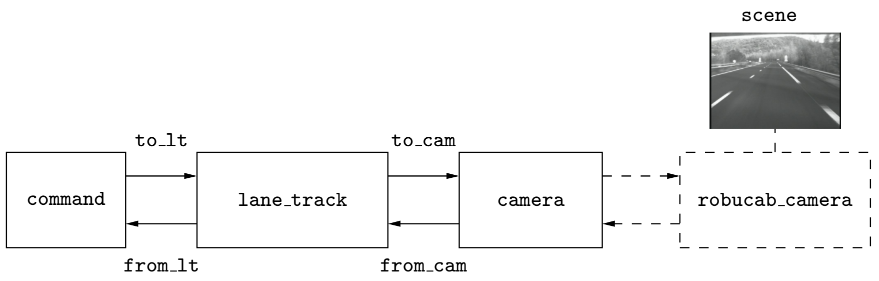


Figure 8: IO box structure for lane-tracking

We do not require box refinement, Step (**6**), for this simple design but our data structures, Step (**7**) are extremely complex having to store the lane model, the tree search position and parameters, plus data relating to the termination condition. The state information for the `lane_tracking` box is as follows:

```
-- Search mode
data mode =
    Init  -- Init mode, reset internal state and go back to beginning
  | Loop  -- Carry on at current level, move to next IZ at current level
  | Back  -- Backtrack, go back up one level retrieve state for that level
  | Updt; -- Update the model and move down to next level

-- Stack element: (level p,IZ no. at level,IZs processed,model,nl,nr)
type stk_t = (int8_t,int8_t,is_t,_Xd_Cxd_t,int16_t,int16_t);

-- We actually save the stack at each level
-- so we have a mode and a list of stack elements
-- (mode,model stack,tracking params.,counter)
type ctlr_lvs_t = (mode,[stk_t],params_t,int16_t);
```

We control the action in the tree search by the `mode` datatype and at each node of the tree we record the current model (`_Xd_Cxd_t`), the depth `p`, the number of the IZ at that level, a list of IZ's processed in previous levels (`is_t`) and the total number of points accumulated on the left (`nl`) and right (`nr`) edges. These are wrapped up in the `stk_t` type. This is held in a list-based stack implementation and is stacked or unstacked as the tree search proceeds. In the overall state, we also record the lane tracking parameters (`params_t`) which can be adjusted online and an overall box execution count which is used to terminate the algorithm if it takes too long to process a particular image.

The state information for the `camera` box is also quite complex since it needs to store the current model in order to correctly format outgoing messages. In fact, the `camera` box is charged with keeping the low-level server up-to-date with the model information. This is so that we can use summaries of the required area specifications rather than lists of points to be checked resulting in a big saving in communications and processing times. We have to be careful, however, that the same model is being referred to by the `lane_track` box, the `camera` box and the camera server.

Once these data structures are defined, however, we have our system-level description of the lane-tracking component. However, the bulk of the work in this subsystem is actually in the functional specification Step (**8**) and implementation Step (**10**).

These were very difficult since the original method as described in [1] is capable of numerous variations in implementation and can be tuned or parameterised in different ways. In fact, we had to revise the top-level functionality several times (Step (**9**)) which also required more work in lower-level functionality, Step (**11**). We do not describe this whole process but we give an overview of the resulting system functional design.

Initialisation There is a fairly complex set of initial values which are computed in Hume's initialisation code. These include: fixed values dependent upon the image size, the fixed set of control points, the initial model and variance matrix and the initial states for the main boxes. Note that we could actually have an "init" box which computes these values once and then passes on to the other boxes but, as for the lane-tracking code, this mostly constitutes linear algebra and is more conveniently implemented as functions. Since their results do not change throughout execution, it is simpler to make these initial expressions. Our methodology does not give any guidelines for what is initialisation code and what is computed in normal operation - this decision is left to the Hume programmer.

Controlling the camera server Since this is intimately linked with the tree search, it is the `lane_track` box which commands data to be sent to the camera server and processed results to be requested. This, however, is done by sending simple commands to the `camera` box which is supported by the necessary functions needed to format and decode messages to and from the camera server. These functions do not have much structure since there is one for each type of message and processing is mostly transformation of coordinate systems.

Tree search The `lane_track` box consists mostly of patterns which implement the required tree search. There are clauses for initialisation, moving to the next IZ at the same level if no edge is found in the current IZ, moving to the next level if the IZ is processed successfully, backtracking to the previous level if no edges are found in the current model, terminating on a successful exit criterion and several error exits, for example termination on backtrack at the top level or forcing a move to the next image if processing one image takes too long.

Model update The stack is managed directly by the `lane_track` box but the model update routines are implemented as functions. In fact, there is only one entry point to these but they are implemented as a hierarchy where each phase of the algorithm is implemented as function and the resultant functions are composed to give the complete processing.

Termination condition The termination predicate is encoded as a separate function since it has to be tuned for different road or lighting conditions. Included in the termination functionality is a summarisation function which produces the curvature and lateral position estimates used by the control system. Note that the control functionality itself is part of the RobuCAB messaging system. Here we simply send the control parameters (via the `command` box) to the messaging system.

Once the functional refinement has been completed we have our box-level system suitable for validation, Step (**12**). For the lane-tracking code, however, we are beyond the capabilities of the current analysis so we can only analyse some of the simpler subsystems it contains.

5.2.4 Costing the lane-tracking components

Name	Resource	Cost expression
`comp_v`	Heap	3365*n1
	Stack	129*n1
	Time	234*n1
`_Xd_init`	Heap	9938*n1
	Stack	131*n1
	Time	474*n1
`_Cxd_init`	Heap	203469*n1
	Stack	185*n1
	Time	474*n1
`get_ctlr_lvs_init`	Heap	213448*n1
	Stack	157*n1
	Time	474*n1

Table 2: Guaranteed Resource Bounds for Lane-tracking code sections

Analysing the entire lane-tracking system is outwith the capabilities of our current prototype analysis. Here, we present resource bounds analysis for the initialisation code for the main `ctlr` box. This can be divided up into the functions `comp_v` which computes the fixed horizontal lines for the model control points, `_Xd_init` which generates the initial model control points X_d, `_Cxd_init` building the initial co-variance matrix C_{X_d} and `get_ctlr_lvs_init` which calls all three of these functions to construct the main loop variable for the `ctlr` box incorporating the model stack used by the tree search and other loop variables such as counters and flags.

While only a small fragment of the whole program, this is code of significant complexity. It required considerable alterations from its original version before analysis. This included developing specialised versions of builtin functions such as vector access and vector length. However, analyses within reasonable time were possible given the recently improved analysis tool.

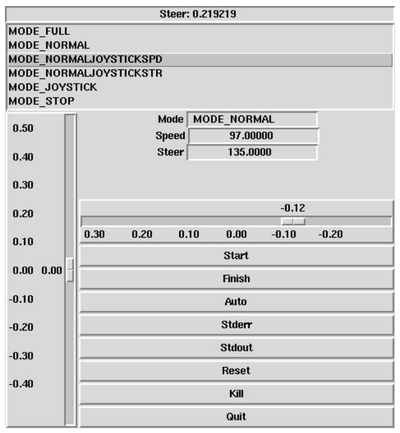
6 Design and implementation of a vision-guided vehicle program

Figure 3 is a schematic of the completed system. We have described the development of the RobuCAB messaging code and of the lane-tracking module. The remaining components are briefly described along with the completed system.

In effect, thanks to our methodology we have independently verified subsystems so in building the completed system we can approximate the overall resource costs for the system by summing the costs of the individual components. External components are difficult to handle, however, and we do not at this stage in the development of the Hume analysis tools attempt to incorporate costs generated by external code.

6.1 Building the complete system

6.1.1 Monitor program


Steer: 0.219219

Mode MODE_NORMAL
Speed 97.00000
Steer 135.0000

Figure 9: TCL-based monitor program

The monitor program, Figure 9, is a simple TCL script with:

1. a text box which displays the last message received from the messaging system (top line),
2. text boxes which display the last mode, speed and steering values received from the RobuCAB (may differ from those in the messaging system),
3. a list-box for selecting the RobuCAB mode (`MODE_NORMALJOYSTICKSPD` is the most commonly used where the steering is controlled by the lane-tracking but the speed is controlled by the joystick),
4. sliders for setting the speed (vertical) and direction (horizontal) in manual mode,
5. buttons for initialising (`Start`) and terminating (`Finish`) the RobuCAB link,
6. a button to switch between automatic (under control of the lane tracking) or manual mode (under control of the builtin sliders),
7. buttons to switch the debug streams on and off,
8. a reset button which zeros the speed and steering sliders (as a soft emergency stop button, the vehicle itself has several hard-wired emergency stop buttons) and
9. buttons to terminate the Hume control program (`Kill`) and the monitor program (`Quit`).

The monitor program is setup as a server and we talk to it via the same data stream mechanism as to the camera server or the RobuCAB.

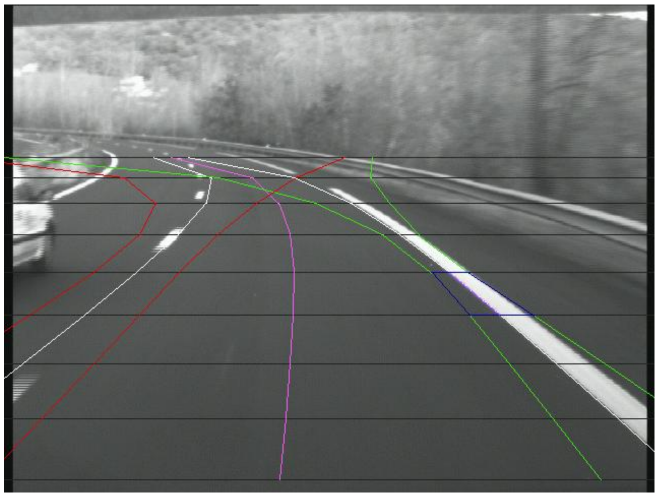


Figure 10: Lane-tracking overlay

6.1.2 Low-level camera server

There are actually two versions of the camera server. The original one ran the OpenGL simulation of a vehicle moving around a track and was used to build and test the lane-tracking code in isolation from the vehicle. This was subsequently rewritten to take image data from the RobuCAB camera and (optionally) to replay sequences of pre-recorded images for testing the code and demonstrating the lane-tracking outside of the RobuCAB itself.

The structure of this server is very simple since it does not incorporate any command logic of its own, being completely dependent upon the Hume control system for its operation. After initialization, it remains in a loop which waits for commands from the Hume code (requests for image processing, updated model data) and actions them as they arrive. Pausing of the system is possible here by simply delaying the return of requested information so we have added a terminal interface to control the apparent generation of image data. The basic components are:

1. The server interface. This implements the protocol for linking to the Hume control program and is quite complex since it is required to synchronise several different types of data with that program.

2. Camera data interface via the 1394 firewire and the libdc1394 library. This reads data from the camera and copies it into a libSDL image surface which is then available to the rest of the program.

3. Image replay system. In place of the camera input we can arrange for the display surface to be filled by reading sequences of images. The image directory and file name format can be set externally. Mostly, we use data from the VELAC vehicle which is a road-going vehicle equipped with a set of cameras and other sensors plus a substantial computer platform for recording the data.

4. Display interface. Using an SDL library display surface, a window is opened to display the current image read from the camera or the current image from the replay sequence.

5. Display overlay subsystem. Again using the SDL library, a set of routines are provided for overlaying the current model and search zone on the display, Figure 10. This is mostly used for tuning the lane-tracking and for debugging.

6. Terminal interface. The program can be controlled by and provides additional debug information to a standard terminal interface. This is used to pause and step through images in the sequence and displays the current control parameters.

6.2 Assembling the components

The lane-tracking was developed first using the track simulation environment. It is completely independent and the only visible outputs for the control system are the control parameters (vehicle lateral position and road curvature estimate) to be sent to the RobuCAB control code. Similarly, the monitor program and the messaging system form an independent unit which was written in isolation and can be used to control the RobuCAB without automatic control. For this module the external interface visible to the overall system is more complex but still only consists of a set of simple control commands.

We thus require some "glue" code to connect these components together. This is in the form of the `command` box mentioned earlier. This box does no actual computation itself but actually has a moderately complex data routing function.

6.2.1 The `command` box

The command box connects all the other components together. It has inputs and/or outputs to:

1. to and from the RobuCAB via the messaging system,
2. from the lane-tracking module (it only reads the control parameters and does not send any commands to the lane tracking),
3. to and from the monitor program,
4. to the switches on the debug multiplexors and
5. from a supplementary box `cmds` which reads in commands from standard input and generates the parameters for the lane-tracking.

The `cmds` box is a low-level feature used in tuning the lane-tracking for a new image type. The lane-tracking needs to be adjusted for the geometric layout of the lanes in the scene and for the lighting conditions. This was originally done, in the early stages of development, by changing constants in the program and recompiling but, as the program grew larger, became cumbersome and slow. The `cmds` box was quickly added to allow the parameters to be changed online without complicating the, then primitive, interface to the lane-tracking box.

The `command` box Hume source is presented in Figure 11. Despite its central role in the system this box is relatively simple. It accepts commands from the monitor program in the form of text messages and relays them to the messaging system or actions them itself, as required. It also has links to the debug (standard output and standard error) stream multiplexors which can enable and disable them. Finally, the control input from the lane-tracking is simply echoed to the message system and messages from the messaging system are relayed to the monitor program. Auto/manual mode is implemented in this box using a boolean flag which switches the control values for the messaging system between the lane-tracking control input and the monitor program slider values.

In wiring this box into the rest of the program there were a number of minor coordination issues and some ancillary boxes such as debug stream multiplexors had to be merged. Apart from these trivial problems the system was composed into a working system with little effort thanks to the ease of connecting components written in Hume using the wiring mechanism.

```
type command_lvs_t = bool;
constant command_lvs_init = true;

box command
  in  (msg::string, intcl::intcl_t, inlt::cp_t, lvs::command_lvs_t)
  out (outtcl::string, cmd::robucarcmd_t, stdout_switch::bool, stderr_switch::bool,
       lvs'::command_lvs_t, exit::int16_t, stdout::string)
match
   (*,  *,           cp,false->(*,       Ctrl cp,                *,    *,    false,*,*)
 | (*,  *,           cp,true)->(*,       *,                      *,    *,    true, *,*)
 | (*,  ('b',_,   _),*, lvs) ->(*,       RCStart,                *,    *,    lvs,  *,*)
 | (*,  ('e',_,   _),*, lvs) ->(*,       Stop,                   *,    *,    lvs,  *,*)
 | (*,  ('m',mode,_),*, lvs) ->(*,       Mode (mode as uint8_t),*,    *,    lvs,  *,*)
 | (*,  ('v',spd, _),*, lvs) ->(*,       Speed spd,              *,    *,    lvs,  *,*)
 | (*,  ('d',str, _),*, true)->(*,       Steer str,              *,    *,    true, *,*)
 | (*,  ('d',str, _),*, false->(*,       *,                      *,    *,    false,*,*)
 | (*,  ('x',_,   _),*, lvs) ->(*,       *,                      *,    *,    lvs,  0,*)
 | (*,  ('y',_,   _),*, lvs) ->(*,       *,                      *,    false,lvs,  *,"err: off\n")
 | (*,  ('Y',_,   _),*, lvs) ->(*,       *,                      *,    true, lvs,  *,"err: on\n")
 | (*,  ('z',_,   _),*, lvs) ->(*,       *,                      false,*,    lvs,  *,"out: off\n")
 | (*,  ('Z',_,   _),*, lvs) ->(*,       *,                      true, *,    lvs,  *,"out: on\n")
 | (*,  ('r',_,   _),*, _)   ->(*,       *,                      *,    *,    false,*,"auto\n")
 | (*,  ('R',_,   _),*, _)   ->(*,       *,                      *,    *,    true, *,"manual\n")
 | (*,  (_,  _,   _),*, lvs) ->(*,       *,                      *,    *,    lvs,  *,*)
 | (msg,*,           *, lvs) ->(msg++"\n",*,                     *,    *,    lvs,  *,*);

wire command (robucar.msg, from_tcl, ctlr.control_out, command.lvs' initially command_lvs_init)
             (to_tcl, cmd_split.cmd, stdout_mux.switch, stderr_mux.switch, command.lvs,
              exit_mux.exit2, stdout_mux.stdout4);
```

Figure 11: The `command` box Hume code

6.3 Performance of the RobuCAB control system

The completed system has been installed and tested on the RobuCAB and proved able to guide the vehicle between two well-marked lane edges.

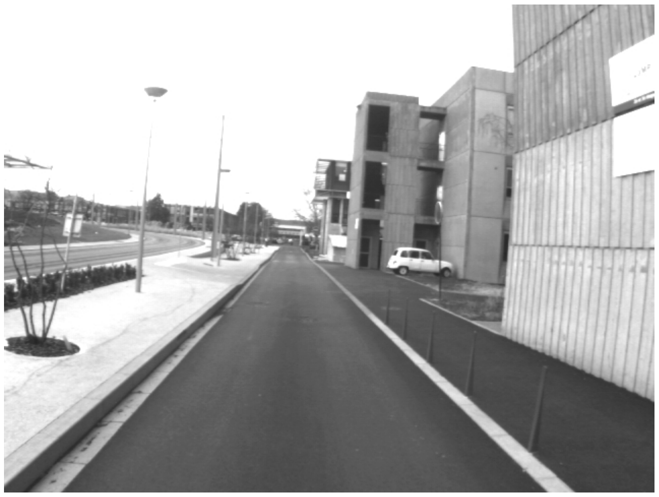


Figure 12: Typical scene for Cycab testing

Figure 12 show a typical scene for use in testing the Cycab control system. The right-hand edge has much better definition than the left-hand edge but the edge detection can be adjusted to cope with this.

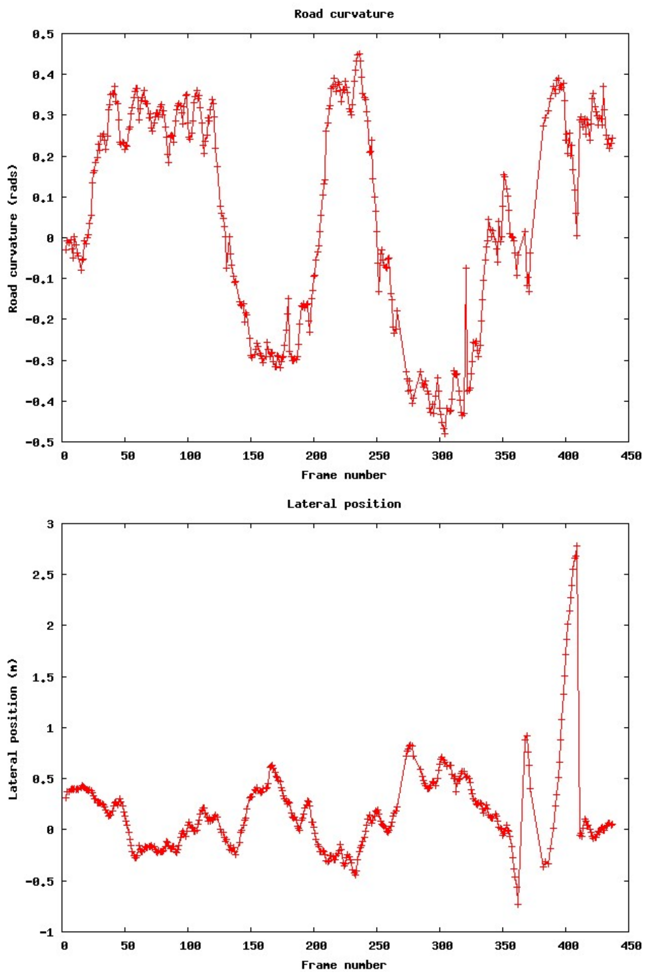


Figure 13: Control parameters for lane tracking

Figure 13 shows the control parameters (road curvature and lateral position) for a short sequence of frames. These follow the actual vehicle positions very well except for the spikes at the end of the sequence where the vehicle actually changes lanes and the algorithm has to recover the correct edges.

Table 3 shows the measured heap, stack and various timings for the running lane tracking program. The heap usages are mostly very small apart from the `ctlr` box which contains the stacked models for the tree search and requires about 183k of RAM. The `circuit` box also uses about 11k of heap to store its copy of the model and boxes `robucar` and `robucar_in` need a few kilobytes to store the data needed to decode the messages. Stack usage is almost trivial since there are very few recursive functions and those that do exist only iterate a very small number of times. These results were for a total of 65528 invocations of the superstep loop. Some boxes execute each time, for example boxes `circuit` and `robucar_out`, whereas others hardly ever execute, for example, the `command` box which only relays results between other components. Note that some of these values are actually streams rather than boxes which can have timeouts for example, but which do not use any heap or stack. We are currently unable to verify these values for 368-based architectures.

The main hume program is about 1.37 Mb in size. This is reasonable for the functionality of the code but small embedded processors such as the M32C would have problems with data of this size. The program compiles in about 113 seconds on a 1.8GHz PC with 500 Mb RAM. The camera server adds an additional 111 Kb of code but most of this is debug and display code. The core of this program

Box	hpMax	spMax	withinMax	timeoutMax	active	cnt	Level
ctlr	187716	364	19470	220980	719699	24320	PR
circuit	11335	341	39875	95865	788854	65528	PR
clock_mux	2	8	37570	95887	286454	65528	FSM
cmd_join	0	8	0	0	0	0	FSM
cmd_split	0	6	0	0	0	0	FSM
cmds	0	7	0	0	0	0	FSM
command	14	18	11	1449273	111	59	FSM
exit_mux	0	8	2	19868714	2	1	FSM
ramp_speed	14	17	39044	95969	549560	65527	FSM
ramp_steer	14	17	38882	95973	464428	65527	FSM
robucar	2756	186	39853	95904	540225	65528	FSM
robucar_in	6275	69	22502	1645077	66804	3248	PR
robucar_in_ch	2	7	41484	1645094	187236	3249	FSM
robucar_out	25	30	39282	95912	542596	65528	PR
stderr_mux	0	9	0	0	0	0	FSM
stdout_mux	0	13	17037	170785	26549	2651	FSM
Stream							
from_tcl	0	0	0	0	0	0	
to_tcl	0	0	24043	1895188	52552	12	
from_sim	0	0	0	0	0	0	
to_sim	0	0	76177	170549	2208789	1021	
from_robucar	0	0	0	0	0	0	
to_robucar	0	0	568	13841870	1298	17	
stderr	0	0	0	0	0	0	
stdin	0	0	0	0	0	0	
stdout	0	0	0	0	0	0	
clk	0	0	0	0	0	0	
exit	0	0	1	19868751	1	1	

Table 3: Heap, stack and execution times for lane tracking Hume code

would be very small indeed.

The table also shows the level of the Hume boxes as determined by the compiler. For the most part these are in FSM-Hume with one or two which call small PR functions which are themselves therefore PR. PR boxes tend to use more stack than FSM boxes.

7 Conclusions

We have successfully built and tested a vision-based control system for the RobuCAB vehicle. This is a major achievment for the Hume language since the system is both complex and has real-time characteristics. The system has actually been developed throughout the EmBounded project and is the culmination of two years of effort. As such there is a slight lack of uniformity in the subsystems in this design because they were developed at different stages of the EmBounded project when the tools were at various stages of development. Nevertheless, we were able to put the whole system together with relative ease since all the components conform, as a minimum requirement, to the semantics laid down before the EmBounded project started.

Objective 1, which involves the continuous assessment of Hume's suitability for real-time (but not embedded) systems has been substantially met. Using our programming methodology we demonstrated how Hume could be used to construct complex real-time systems. We also showed how resource analysis could play a role in this process. Treating the individual Hume subsystems as components we built the completed system by a simple process of wiring them together into a system of respectable size and complexity.

Objective 2 has been partially met. It has to be remembered that the core technology used by the Hume analysis tools is at the forefront of current technology. The analysis tools have improved continuously throughout the project and will continue to be developed afterwards. The present state of these tools is of prototype quality but even so we are beginning to see how it can be integrated into the real-time systems design process. We were able to analyse, for heap, stack and execution time, a significant proportion of the code developed for the RobuCAB control system. Future work will include validation of the analysis results for the code developed here.

References

[1] R. Aufrère, R. Chapuis, and F. Chausse. A model-driven approach for real-time road recognition. *Machine Vision and Applications*, 13:95–107, 2001.

[2] N. Scaife, K. Hammond, S. Jost, H.-W. Loidl, G. Michaelson, and J. Sérot. Costing by construction: Compositional design and verification of embedded applications using the hume software development methodology. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium*, St. Louis, MO, United States, April 2008. IEEE. Submitted.

[3] Jocleyn Sérot and Norman Scaife. Real-time testbed applications. EmBounded Project Deliverable D27 (WP8a), EmBounded: Automatic Prediction of Resource Bounds for Embedded Systems, Laboratoire LASMEA, Blaise Pascal University, Les Cezeaux, F-63177, Aubière, France, December 2007. IST-510255.