ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221004403

Mobile Robot Control - The Subsumption Architecture and occam-pi.

Conference Paper · January 2006

Source: DBLP

CITATIONS
17

READS
80

3 authors, including:

Matthew C. Jadud
Bates College

43 PUBLICATIONS **487** CITATIONS

SEE PROFILE

All content following this page was uploaded by Matthew C. Jadud on 05 February 2015.

The user has requested enhancement of the downloaded file.

Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

Mobile Robot Control
The Subsumption Architecture and occam-pi

Jonathan SIMPSON, Christian L. JACOBSEN and Matthew C. JADUD

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NZ, England.

{js219, clj3, mcj4}@kent.ac.uk

Abstract. Brooks' *subsumption architecture* is a design paradigm for mobile robot control that emphasises re-use of modules, decentralisation and concurrent, communicating processes. Through the use of occam-pi the subsumption architecture can be put to use on general purpose modern robotics hardware, providing a clean and robust development approach for the creation of robot control systems.

Keywords. Mobile robots, Robot control, Subsumption architecture, occam-pi

Introduction

Robotic control can be seen as a mixture of engineering and cognitive science and as such it presents unusual challenges to the programmer. Robotic control methodologies have tended to move from simplistic, predefined actuator actions based on specific input criteria to tight feedback loops with input from the environment, giving more robust solutions. In these environments, where many such control loops are required, the opportunities for the application of parallel programming to create simple and robust solutions are numerous.

Continuous, concurrently running processes are critical to robotics, as a robot typically has a number of inputs and outputs that must be handled simultaneously. If we wish to keep such a robot from running into walls, at least one process must continuously monitor the space between the robot and nearby objects using some sort of range-finder. Whilst this specific behaviour may be important, we cannot focus solely on one sensor at the exclusion of all other behaviours the robot may be designed to perform.

Even simple robots can have many different tasks to do simultaneously. For example, a robot might try to avoid bumping into walls whilst also trying to create the largest treacle pudding in the world. The latter task is the main purpose of the robot, but the first is important for the robot to meet its goal as designed and needs to be handled constantly alongside the robot's main task.

1. Traditional Approaches to Robotic Control

1.1. The Hierarchical Paradigm

A hierarchical approach to robotic control focuses mainly on the planning aspect of a robot's behavioural cycle. The robot senses its environment, plans its next action based on these senses, and then takes appropriate action using available actuators. At every stage, the robot explicitly plans its next action from the the knowledge it has gathered about the environment so far. Essentially these robots are reflex agents [1], selecting actions from rule matches on the current 'perceptions' from sensory input.

This approach traditionally employs a top-down analysis of the desired behaviour of the robot during the design phase and then the implementation of a sequence of modules. These modules work to read values from the sensors available to provide data about the environment (perception), devise strategies to perform the desired behaviours given the environmental state (cognition), and then compose the signals that control the actuators to achieve those behaviours (action).

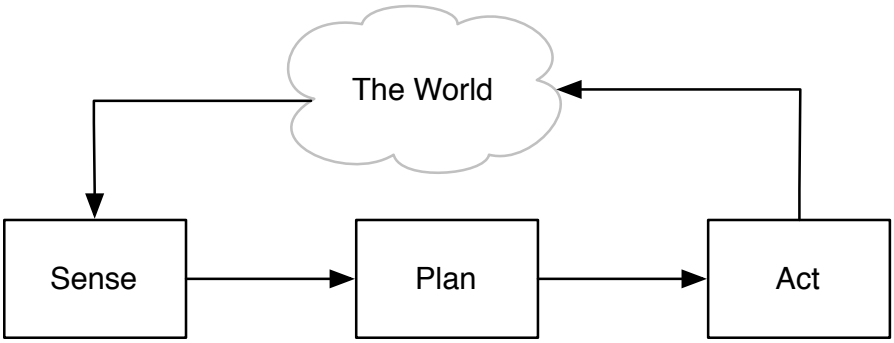

Figure 1. The typical structure of a hierarchical robot control system

Unfortunately, this approach, using a top down design and sequential modules does not encourage a separation of concerns, and can introduce dependencies between functional layers, especially where feedback loops are used or output monitoring is required. Brooks [2] identifies a different model whereby cognition can be observed simply using perceptive and action systems that interact directly with each other in a feedback loop through the environment, effectively a behavioural paradigm for control. He called this biologically-inspired model of robotic control the *subsumption architecture*.

1.2. The Behavioural Paradigm

Behavioural control is focused around the idea of removing centralised control structures and instead linking actions directly to changes in the input sources themselves. This is an approach most fully demonstrated by Valentino Braitenberg's *Vehicles* [3], a set of sensors and actuators connected almost directly together in various combinations to display emergent behaviours that mimic more complex human actions like love, aggression and cowardice.

Emergent behaviour can often occur from the interaction of simple behaviours combined with the complex environment, and being able to take advantage of these emergent behaviours can make the task design of robotics systems simpler and the code involved more robust. Behavioural systems often employ a set of pre-programmed condition-action rules which are run concurrently over the inputs, and the system has little internal state. Development using this architecture can map well to occam-pi robotics, but it can make development of specific and complex robotic controls hard, due to the requirements of responding to changes in the environment for changes of behaviour.

2. Brooks' Subsumption Architecture

The subsumption architecture involves building robot control systems with increasing levels of competence. Each additional level builds upon and potentially interacts with the inputs and outputs of existing, previous levels to add higher levels of competency, leaving the lower levels intact, functional and operational within the overall system.

Levels are constructed from components which make up the architecture as a whole. These components are referred to as 'modules', and consist of small asynchronous processors, sending messages over connecting 'wires' which have a single element buffer. Inputs to these modules can be *suppressed* and their outputs can be *inhibited* by wires from other modules. Unlike occam-pi channels, these wires are assumed to have frequent message loss due to the subsumption/inhibition mechanisms and as such a single element buffer provides constant access to the last successfully received value from an input line.

2.1. Suppression

Suppression is achieved by connecting an additional wire to the input of a 'suppressed' module. Inputs received along this additional wire are sent to the module as replacement input for its usual input channel and other data inputs are ignored whilst the suppression occurs. The period for which this secondary input channel takes precedence is specified in the time constant of the suppression. This process essentially replaces all other inputs to the module with input coming from the 'suppressing' module.

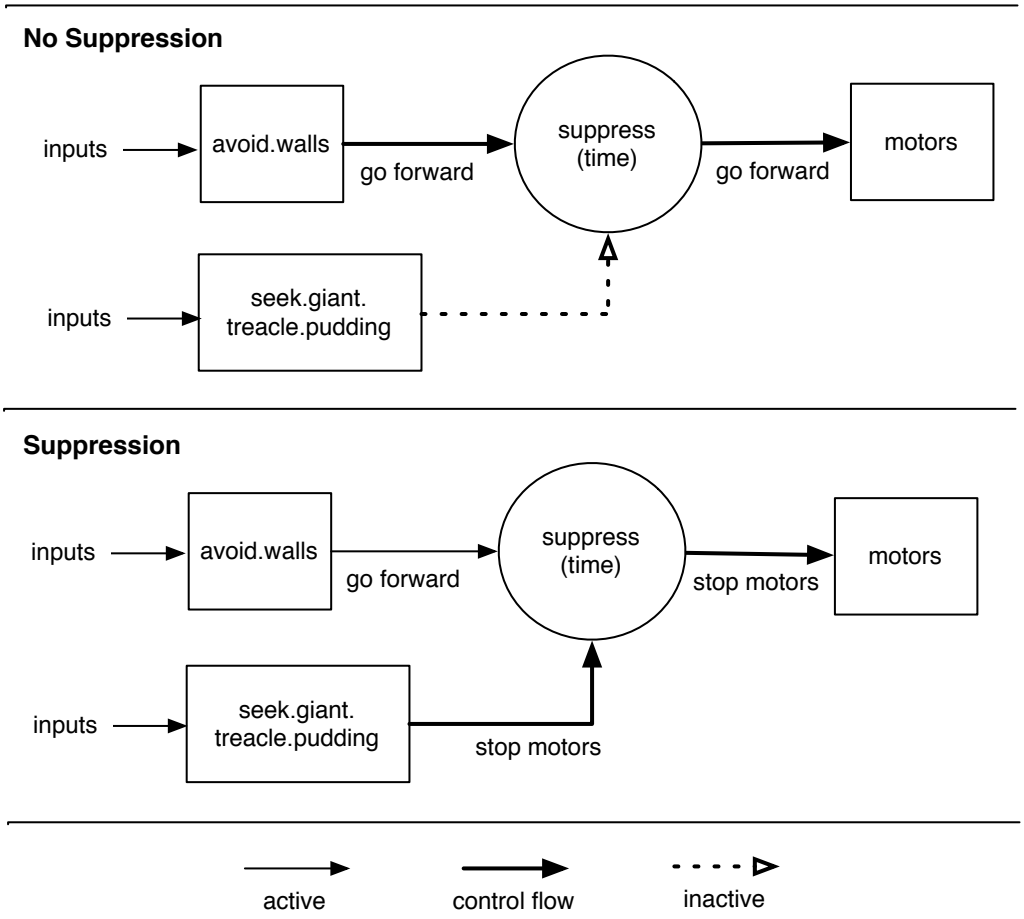

Figure 2. Suppression, whilst seeking giant treacle puddings

To explain suppression, assume our robot has a `seek.giant.treacle.pudding` module, which is used to seek out pre-made, giant puddings (as shown in figure 2). If our robot managed to find a massive treacle pudding, it would then suppress any outputs from our `avoid.walls` module, as our robot no longer needs to avoid walls... because it now possesses a giant treacle pudding, and its task is complete.

2.2. Inhibition

When inhibiting a module, a wire from the 'inhibiting' module which will control the inhibition is connected to the output site of the 'inhibited' target module. If anything travels along this wire, output from the target module will be blocked, and the output is lost for the duration of time specified by the inhibitor. Inhibition is useful for disabling specific behaviours where their activity at a particular time or circumstance is undesirable.

Additionally, inhibition can be used on module outputs where suppression is taking place. If the wire in question is suppressing a behaviour that we desire from another level this will allow it to break free from the control of the suppressing module.

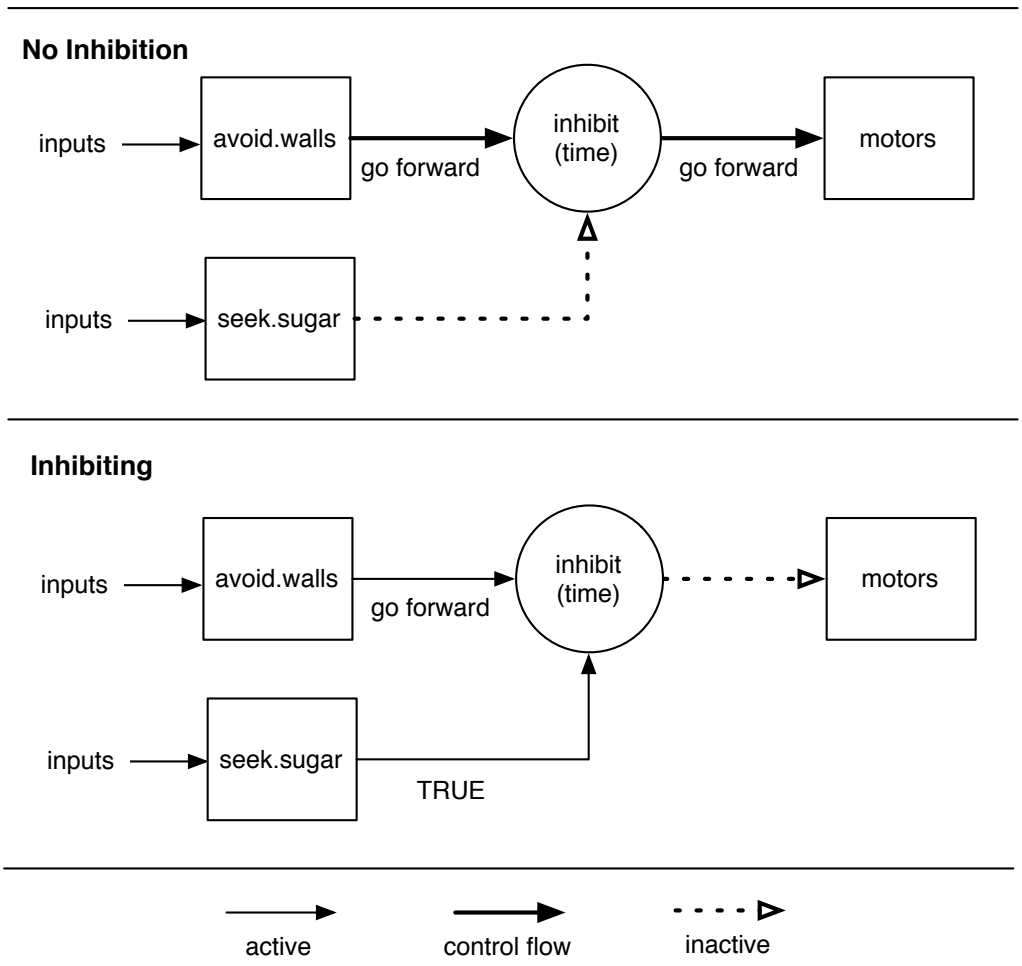

Figure 3. A visual representation of inhibition

For example, as can be seen in figure 3, our treacle-making robot might discover a pile of sugar near a wall. In this situation, a `seek.sugar` module might inhibit the outputs of the `avoid.walls` module to the motors to avoid movement being triggered whilst it collects sugar, even if the current position of the robot is near a wall.

2.3. Subsumption

Use of the subsumption architecture means that a basic control system can be established for the lowest hardware level functionality of the robot and additional levels of competence can be built on top, with little change required to the existing modules and layers. With correct use of suppressors and inhibitors, the system can vary between several different modes

of operation depending on inputs, making the best re-use of already written and debugged modules in existing levels whilst doing so.

For example, a path planning module can coexist with a random heading generator that would otherwise generate a 'wandering' behaviour for the robot. Output from the path planning module could be used to suppress outputs from a random path generator module, allowing it to take control of the robot's motion. This would mean the a robot could establish a target location after exploring an environment and then head towards it.

This style of control can be likened to the biological idea of reflex and cognitive actions. Reflexes occur quickly to protect the body, without any cognitive input before they occur. A simplistic base level of behaviours can simulate this in providing a computationally cheap and prioritised protective layer of functionality for the robot. Above this simulated level of reflexes, more complex behaviours can be added that will fall back if there are not appropriate actions or outputs available.

By using separate inhibitors and suppressors the behavioural modules are isolated from the interaction of the different layers of the system. These modules can be debugged and stay static, making them robust even as the system grows around them.

3. The Subsumption Architecture and occam-pi

A number of the concepts in Brooks' subsumption architecture bear considerable resemblance to primitives or specific abilities of the occam-pi language [4]. The processor 'modules' that make up the system have four specific states that perform different operations, such as sending a message on an output line or making a calculation based on input. These states are switched between to determine the behaviour of the module. occam-pi provides more flexibility than Brooks' processes in allowing the definition of processes with arbitrarily complex behaviour. Implementation of the simplistic operations comprising individual modules in the original subsumption architecture is straightforward using the occam-pi language.

Where the subsumption architecture has lossy/unreliable wires, occam-pi gives us full communication channels with reliable message delivery. Brooks worked around the unreliability of 'wires' in his implementation by using single item buffers to allow access to the last received value on a wire at any given time, ensuring that modules can always execute. If we wished to simulate this behaviour in occam-pi we could build a single item buffer module to allow values to be read at any time on a channel, but it is for the most part more desirable to benefit directly from the reliable communications provided by occam-pi.

Suppression and inhibition in the original subsumption architecture are performed directly at the input and output sites of wire connection, but this is not possible in occam-pi. However by modelling these actions as processes, increased transparency is brought to the network and indeed the network diagrams in Brooks' own report separate these two actions into distinct elements.

3.1. Suppression

Suppression is achieved by inserting a process (as shown in figure 4 on the following page) between two communicating processes that can also receive input on a third `suppress` channel to control the suppression. Under normal conditions, inputs received by the suppress process are routed from `in` to `out`. The first input received on the `suppress` channel whilst the process is in a non-suppressing state triggers suppression for the length of time specified in the time constant given to the process. When suppression is taking place, inputs from the `suppress` channel are routed to the `out` channel. Subsequent values received on the `suppress` channel do not reset the time-out value on the suppression process, although once the process switches back to normal, continued inputs will initiate the suppression once more.

```
PROC suppress.int (VAL INT timeout,
                   CHAN INT suppress?, in?, out!)
  TIMER tim:
  INITIAL INT time IS 0:
  INITIAL BOOL suppressing IS FALSE:
  INT value:
  WHILE TRUE
    PRI ALT
      NOT suppressing & suppress ? value
        SEQ
          suppressing := TRUE
          tim ? time
          time := time PLUS timeout
          out ! value
      NOT suppressing & in ? value
        out ! value
      suppressing & tim ? AFTER time
        suppressing := FALSE
      suppressing & suppress ? value
        out ! value
      suppressing & in ? value
        SKIP
:
```

Listing 1. A process providing suppression for a channel of integers in occam-pi.

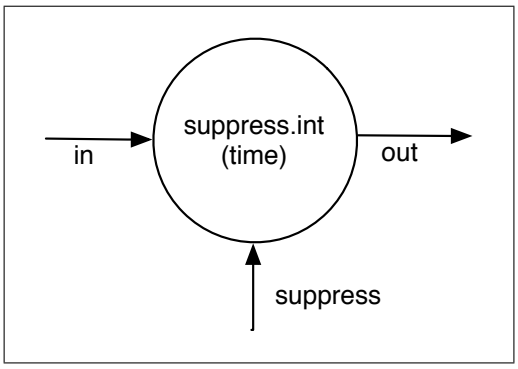

Figure 4. The `suppress.int` process, which provides suppression on an occam-pi channel of integers.

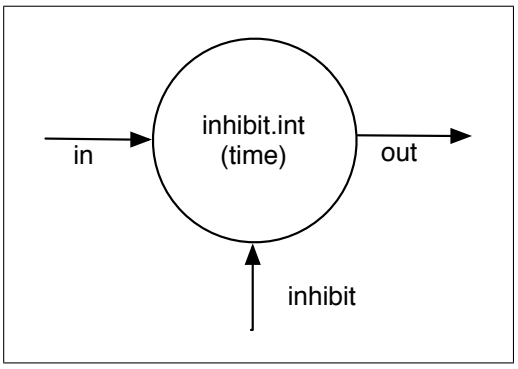

Figure 5. The `inhibit.int` process, which provides inhibition on an occam-pi channel of integers.

3.2. Inhibition

Inhibition can be achieved by placing a process between the target module and a module reading output from it. This process also has a second channel, for receiving signals to indicate when inhibition should take place, and is initialised with a time constraint parameter that determines the amount of time that inhibition will occur for once triggered. Each time a value is sent on the inhibition control channel, the time before the process will stop inhibiting is reset. This process is slightly different for each type it must inhibit, and the code for a version of this process inhibiting integers can be seen in listing 5.

4. Robotics with occam-pi

A language like occam-pi has a natural place in the world of robotics, and in the past its use has been explored on small platforms like the LEGO Mindstorms [5]. In this paper our ex-

```
PROC inhibit.int (VAL INT timeout, CHAN BOOL inhibit,
                  CHAN INT in?, out!)
  TIMER tim:
  INITIAL INT time IS 0:
  INITIAL BOOL inhibiting IS FALSE:
  INT data:
  BOOL flag:
  WHILE TRUE
    PRI ALT
      inhibit ? flag
        SEQ
          inhibiting := TRUE
          tim ? time
          time := time PLUS timeout
      inhibiting & tim ? AFTER time
        inhibiting := FALSE
      NOT inhibiting & in ? data
        out ! data
      inhibiting & in ? data
        SKIP
:
```

Listing 2. A module to allow inhibition of an occam-pi channel.

perimentation takes place on a Pioneer 3 robot[1]. The Pioneer 3-DX, produced by ActivMedia Robotics, has two wheel differential drive, sixteen ultrasonic range-finders arrayed around its circumference, and a high resolution laser range-finder, which provides centimetre resolution to an eight meter distance in a forward-facing, 180-degree arc. Inside the particular robot used in our experiment is a 700MHz PC104 board running Debian GNU/Linux.

4.1. Player/Stage

There are several ways to program a robot like the Pioneer 3. First, it is possible to forego the embedded PC104 and program directly against the robot's hardware control board, connected to the PC via a serial port. Second, the manufacturer provides an object-oriented API (accessible from C, C++, Java, and Python), called ARIA [6], which provides a control interface for all of their robotics platforms. Third, and most interesting, is the open-source Player API—a cross-platform robotics API written in C/C++ [7].

Player is interesting as it provides an abstracted driver interface for motors, sensors, and other devices typically found on a robot, allowing control logic to be ported easily from one robotics platform to another whilst minimal modification of applications on the part of the developer. Additionally, it is built as a client/server application, meaning code written against the client library might then be run on a remote desktop PC, while the server runs on a robot connected via ethernet or a serial port.

This separation also makes authoring a graphical simulator significantly easier; currently, there are two that ship with the Player library. The first is Stage, a 2D simulator capable of displaying dozens of robots simultaneously; second is Gazebo, a 3D simulator which provides a virtual world complete with accurate physics for more detailed testing of control algorithms.

4.2. Player/Stage and occam-pi

Player, like many other robotics control libraries, is written in a sequential language, with no abstractions provided to aid the programmer in dealing with the concurrent programs that

[1]Our development work relies heavily on simulation, with testing being carried out on real robots.

must necessarily be written to control robots engaged in interesting tasks. Player exposes a single control loop, implying that programmers must write their own multi-threaded applications, and be continuously aware of timing issues in polling the driver. Solutions of this nature are often fragile in the face of race hazards and deadlock.

To make the Player library safer for use in control system structures that have the potential to be massively concurrent, we have wrapped the library using our SWIG wrapper interface generator [8]. This allows us to access the C-library directly from occam-pi programs running on the Transterpreter [9], a portable run-time for the occam-pi programming language. Just making Player available as a foreign library is not enough, however. A small accompanying library, written in occam-pi, provides a process-centric interface to the underlying C API [10]. This combination of an occam-pi process-centric interface and library can deliver data between 50 and 60 times faster than the update speeds of the sensors available when running in basic process networks.

The end result is a portable, thread-safe robotics library that allows us to develop code on any robotics platform that the Player API has been ported to, of which there are many.

5. Robot Control with the Subsumption Architecture and occam-pi

To explore how the subsumption architecture becomes one process network layered on top of another in practice, we developed a simple robot control program [11] that has multiple behaviours and two levels of competence. At its first level of competence, the robot avoids colliding with objects it can see using its laser range-finder, wanders an environment and pivots backward away from objects it detects.

The second competence level is added such that when the robot is backing up, it will check the distance behind itself using the four central sonar on the back of the Pioneer, and instead of continuing to back up, will go forward to give it room to complete the turn, whilst still not colliding with objects in either direction.

As discussed previously, the robot's laser range-finder is forward facing and covers a 180 degree arc, meaning that the sonar array must be used for the second level of competence. This example demonstrates the use of both shared and multiple sensor inputs to the control program, and also shows the behaviours that can be achieved by mixing inhibitors and suppressors even with simplistic modules.

Although implementing these behaviours explicitly could be more concise [5], we believe the subsumptive approach can be made to scale to increasingly sophisticated behaviours where a direct implementation cannot.

5.1. Infrastructure

Critical to the operation of our robot is the occam-pi Pioneer Robotics Library [10]. In particular, it exposes a series of `brain.stem` processes which can be used to interact with the robotics library. The laser data channel carries an array of 180 integers ranged [0-800] and the sonar data channel carries an array of 16 integers ranged [0-500], both of which are distances in centimetres. In our example, we declare the end of these channels `SHARED` to enable multiple processes from different levels of the architecture to get access to the data.

The motor control channel takes a `PROTOCOL` of three integers representing the speed of the robot in the X-axis, the Y-axis, and its rotational velocity; in the case of our particular robot there is no Y-axis (the robot cannot scuttle sideways). These control commands are abstracted over by the `motor` process which takes in a channel of integers, mapped to constants for convenience (e.g. `motor.stop`, `motor.forward`, etc.).

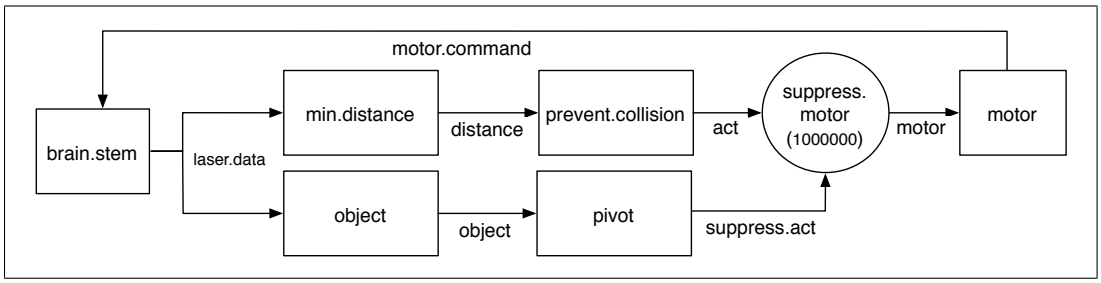

Figure 6. A process network diagram for a robot that avoids colliding with objects whilst also wandering and turn away from objects it encounters.

5.2. First Level of Competence

The first level of competence has two main behaviours and is shown in figure 6. One of its behaviours is to avoid colliding with any objects it can see with the laser range-finder. This behaviour keeps the robot from colliding with objects and acts essentially as a protective reflex, taking action regardless of whatever else the robot happens to be doing. To do this, it uses a combination of two processes: `min.distance` and `prevent.collision`. The `min.distance` process reads through the entire array of laser data as each set of data arrives, and sends the minimum values out on a channel of integers to the `prevent.collision` process.

```
PROC prevent.collision (CHAN INT distance?, CHAN INT act!)
  WHILE TRUE
    INITIAL INT min IS 0:
    SEQ
      distance ? min
      IF
        min < 20
          act ! motor.stop
        TRUE
          act ! motor.forward
:
```

Listing 3. `prevent.collision`, a base level behaviour to prevent the robot from colliding with objects in any direction.

`prevent.collision` watches for objects using the laser range-finder. If the value received from `min.distance` is less then 20cm then an object is 'seen' by the robot, and a `motor.stop` message is sent to the motors. If no object is observed, then the process sends a `motor.forward` commands to the motors, meaning the robot can recover from being halted if the environment subsequently changes.

The second behaviour at this level is that the robot will pivot backwards whenever it detects an object in front of it. Using the `laser.data` channel, another process `detect.object` reads through the central 90 degrees of the laser array containing each scan, and looks for any obstacles closer than 75cm. Upon processing an entire scan, the process sends a boolean value on the `object` channel indicating that it has detected an object in the robot's path.

The `pivot` process sends a back up command over the `suppress.act` channel to the `motor` process whenever a signal is received on its own `object` channel. It does nothing otherwise, as outputs from the process control the suppression line of `suppress.motor`.

The avoid collision, wandering and pivot backwards behaviours are connected by a suppressor, `suppress.motor` which is the same as the `suppress.int` process seen in figure 4 on page 230. This means that when the `pivot` process is active, motor commands from

```
PROC pivot (CHAN BOOL object?, CHAN INT suppress.act!)
  WHILE TRUE
    BOOL is.object:
    SEQ
      object ? is.object
      IF
        is.object
          suppress.act ! motor.back.right
        TRUE
          SKIP
:
```

Listing 4. `pivot`, a process that turns the robot if an object is detected, or goes forward otherwise to provide 'wandering'

`prevent.collision` are dropped (telling the robot to go forward, or stop), and the command to turn right from `pivot` will be sent instead.

The time interval on the suppressor is set to 1000000μ (one second) meaning that the robot will back up for that period before the choice is made again whether to pivot or go forward. When there is a clear path in front of the robot again, the lower level behaviour (of going forward when there is clear space ahead) resumes control of the robot.

5.3. Second Level of Competence

Up to this point, our robot can wander in space, turn away from objects and prevent collisions. However, it has a deficiency: it is possible for the robot to back into walls whilst trying to reverse away from objects in front of it, as shown in figure 7. Following the principles of the subsumption architecture, we can add another behaviour that checks whether the robot has space to back up and turn. When there is no space to back up, this behaviour can inhibit the signals coming from the `pivot` process, and instead allow the `motor.forward` commands from `prevent.collision` in the base level through, as shown in figure 9 on the next page. Causing the robot to travel forward temporarily gives more space for it to back up and pivot into, adding more 'points' to the turning motion, but allowing the robot to successfully complete its backward turn.

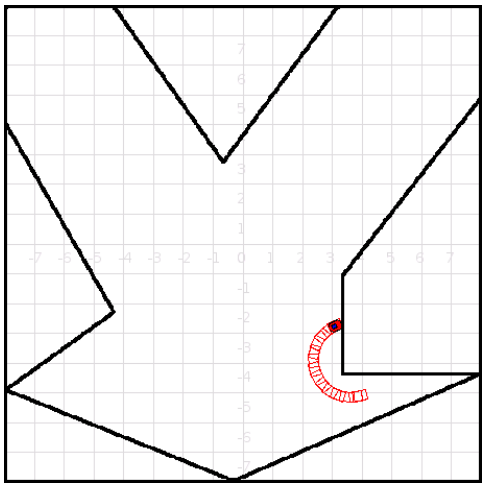

Figure 7. At the first level of competence, the robot is able to reverse into walls while trying to find clear space.

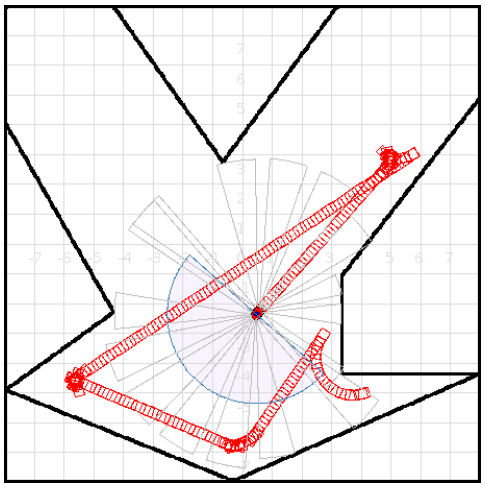

Figure 8. Demonstrating the second level of competence in action, the robot successfully navigates the environment without running into the wall behind it.

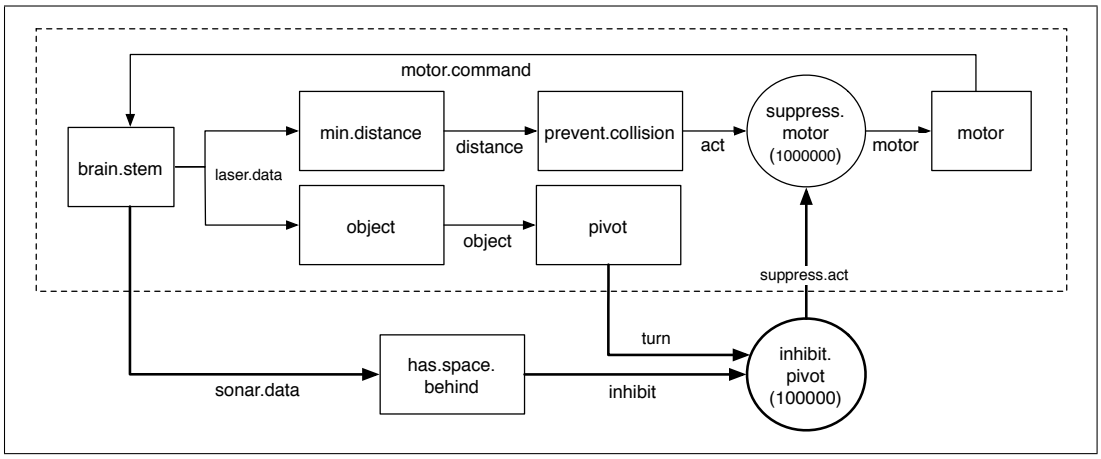

Figure 9. A process network diagram for a robot control system with two competencies, allowing more successful negotiation of an environment based on both sonar and laser data.

The `has.space.behind` process uses the middle four sonar sensors at the rear of the Pioneer 3, and checks that there is room behind the robot.

The `inhibit.pivot` is the same code that was introduced earlier as an example of inhibition on occam-pi channels. Different delays are used for the inhibitor and suppressor, because the `inhibit.pivot` must restrain the outputs of the `motor.suppress` for long enough that the robot can move forward a significant amount. Otherwise, the robot falls into a needlessly long see-saw motion, wobbling back-and-forth when caught between "a rock and a hard place." After adding this additional layer of competence, it is possible to see in figure 8 on the facing page that the robot can successfully negotiate the environment whilst not backing into walls.

In our example, it is possible to see that the processes originally used in lower levels are maintained and the lower-level system is kept intact. New levels of functionality merely augment the system and improve its overall ability to perform the desired task. These levels can be progressively debugged as they are added, and once debugged can be relied upon by subsequent layers, meaning the system should remain robust even as it grows in size and complexity.

6. Conclusions and Future Work

Based on our initial explorations, the subsumption architecture appears to be a natural design paradigm for occam-pi robotic control. We can implement desirable, low-level behaviours for our robots, and then extend those networks with higher-level behaviours, using Brooks' notions of inhibition and suppression. However, further experiments are necessary to convince ourselves of the value of the subsumption architecture as a paradigm for robotic control in the occam-pi programming language.

Given the example levels of competence presented, it would be useful to investigate creating additional levels for the example presented in this paper. Making these levels modular such that others building control systems in occam-pi can make use of them would also seem wise.

Having a stable and debugged core of modules for use in developing subsumption architectures in the library that is used with occam-pi for Player/Stage would be a useful step forward to promote this approach to control with the language. Developing similar sets of modules for the sensors found on smaller, more commonly available robotics platforms like the LEGO Mindstorms would provide additional opportunities for use of this paradigm for teaching purposes.

Additionally, implementing Brooks's subsumption architecture in a manner more closely mirroring its original form, detailed in his technical report [12], would be an interesting challenge to attempt in the occam-pi language.

Acknowledgements

We are very grateful to Damian Dimmich for providing Player client library wrappers, making it possible to program the Pioneer 3 using occam-pi and the Transterpreter [9].

References

[1] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 2.4, pages 46–48. Pearson Education, 2003.

[2] Rodney A. Brooks. *Cambrian intelligence: the early history of the new AI*, chapter Preface, page xi. MIT Press, Cambridge, MA, USA, 1999.

[3] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, MA, USA, 1986.

[4] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[5] Christian L. Jacobsen and Matthew C. Jadud. Towards concrete concurrency: occam-pi on the LEGO mindstorms. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 431–435, New York, NY, USA, 2005. ACM Press.

[6] ActivMedia Robotics. Advanced Robotics Interface for Applications (ARIA) Robotic Sensing and Control Libraries. http://www.activrobots.com/SOFTWARE/aria.html.

[7] B. Gerkey, R. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003), Coimbra, Portugal, June 30 - July 3, 2003*, pages 317–323, 2003.

[8] Damian J. Dimmich and Christan L. Jacobsen. A Foreign Function Interface Generator for occam-pi. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, pages 235–248, Amsterdam, The Netherlands, September 2005. IOS Press.

[9] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In *Communicating Process Architectures 2004*, pages 99–107, 2004.

[10] Christian L. Jacobsen and Matthew C. Jadud. The occam Pioneer Robotics Library. http://www.transterpreter.org/documentation/occam-pioneer-robotics-library.pdf.

[11] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. A bump and wander robot using the Subsumption Architecture in occam-pi. http://www.transterpreter.org/wiki/Subsumption.

[12] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical report, MIT, Cambridge, MA, USA, 1985.

View publication stats