Process-Oriented Subsumption Architectures in Swarm Robotic Systems

Jeremy C. POSSO [a], Adam T. SAMPSON [b], Jonathan SIMPSON [c] and Jon TIMMIS [d,1]
[a] *Department of Computer Science, University of York, UK*
[b] *Institute of Arts, Media and Computer Games, University of Abertay Dundee, UK*
[c] *School of Computing, University of Kent, Canterbury, UK*
[d] *Department of Electronics, University of York, UK*

Abstract. Previous work has demonstrated the feasibility of using process-oriented programming to implement simple subsumption architectures for robot control. However, the utility and scalability of process-based subsumption architectures for more complex tasks and those involving multiple robots has not been proven. We report our experience of applying these techniques to the implementation of a standard foraging problem in swarm robotics, using occam-π to implement a subsumption control system. Through building a system with a realistic level of complexity, we have discovered both advantages and disadvantages to the process-oriented subsumption approach for larger robot control systems.

Keywords. process-oriented, robotics, subsumption, swarm robotics.

Introduction

In the process-oriented programming model, based on Hoare's Communicating Sequential Processes [1], programs are constructed as networks of concurrent processes communicating with each other over channels. The occam-π [2] language directly supports process-oriented programming, with processes and channels as first-class features. The process-oriented model is not limited to occam-π, being available through library extensions to languages such as Java (JCSP [3]) and Python (PyCSP [4]).

Robot control is inherently a concurrent problem: sensing the environment, reacting to the environment, and controlling actuators to take action. The concurrent programming facilities of the process-oriented model have previously been used to implement a number of common architectures used for robot control [5]. Brooks' subsumption architecture is one example: a subsumptive control system is implemented as "a set of small processors which send messages to each other" [6], which can be expressed directly using process-oriented techniques.

The use of occam-π as an implementation language for process-oriented robot control systems is driven by the availability of runtime support for robot platforms and simulation environments, primarily through the Transterpreter [7]. The Transterpreter is a small, portable occam-π virtual machine designed for embedded devices, with robotic control systems being a key application. The Transterpreter runtime system allows concurrent programming at scales ranging from a few dozen processes on a robot platform like the LEGO Mindstorms RCX (16MHz Renesas H8300 CPU, 32kB RAM) to thousands of processes on more powerful platforms such as the Surveyor SRV-1 (500MHz Analog Devices Blackfin BF537 CPU, 32MB RAM).

[1]Corresponding Author: *Jon Timmis, University of York*. E-mail: `jtimmis@cs.york.ac.uk`.

We are interested in the applicability of the process-oriented subsumptive control model to swarm robotic systems: those in which many simple robots work together to produce complex emergent behaviours. For reviews of the work in this area, the reader is referred to [8,9]. In this paper, we describe our experience with the development of a process-oriented subsumptive control system.

We review previous work on process-oriented subsumption architectures, which has concentrated on simple, isolated devices. We then explore the design of a relatively complex subsumptive system to solve the foraging problem and make use of the Player/Stage simulation tool [10] in which to preform our experiments. We evaluate this architecture in terms of its problem-solving effectiveness, and in terms of the flexibility and maintainability of the resulting control system.

1. Previous Work

The choice of runtime system and implementation language for a robotic control system is important, since it places constraints on where the control system may execute. The combination of occam-π and the Transterpreter allows the robot's hardware to run relatively complex software that can react to external events in flexible ways. Using process-oriented techniques, we can construct reusable concurrent software components that may be predictably composed to build complex programs. Furthermore, we can design systems involving interactions between multiple control systems – such as between robots within a swarm. The use of a lightweight runtime allows the use of a high degree of local intelligence on the robot platform itself, rather than relying on teleoperation and remote operation from a host computer. This untethered approach is an attractive option for swarm robotics, where autonomy and reliability are key attributes.

Subsumption architectures [6] are what are called *reactive* architectures that allow for the decomposition of control (or behaviour) of a robot into smaller, simpler modules. Simpson et al. [11] applied an early version of Brooks' subsumption architecture [6] as a methodology for structuring process-oriented robotics programs. This work yielded occam-π implementations of the fundamental subsumptive components, along with rules for their use. This initial investigation served as a proof of concept for combining the process-oriented and subsumptive techniques in robotic control.

A follow-up study [5] considered subsumption along with other architectures in a broad comparison of design principles and components of behavioural control architectures. The authors identify difficulty in scaling subsumption architectures owing to levels of interdependency between layers, developed as control systems become more complex. A later revision to Brooks' work on subsumptive control revised the principles of the subsumptive model upon which the process-oriented subsumption work was based; the authors identify these later changes to subsumption itself as addressing a number of the scaling difficulties.

The work of Simpson et al. primarily identifies design principles and patterns for the construction of robotic control systems using process-oriented techniques through the examination of small, straightforward case studies. These techniques and hypotheses have not yet been tested or verified in larger and more complex real-world tasks, nor have the requirements of interaction between multiple robots been considered – areas that the work described in this paper attempts to address.

2. A Case Study: Foraging

In this section, we will use occam-π and the Transterpreter to develop a process-oriented subsumptive control system for a standard robotics control problem, in order to identify the

strengths and weaknesses of the method for larger programs. Foraging is a typical task in swarm robotic systems, acting as a simple platform for the evaluation of systems.

2.1. The Foraging Task

Foraging involves a robot starting from its home base and venturing out into the world to gather specific *attractor* objects [12]. Upon finding such an object, the robot picks it up and returns it to the base. The robot repeats this action until all the attractors in the environment have been collected, at which point the task is complete. Foraging with multiple robots introduces a dynamic element to the domain and so provides a good test of occam-π's real-time communication facilities.

The foraging variant used in this work is garbage collection, where the task requires a team of several identical robotic agents to find items of rubbish in an environment, pick them up and deposit them in a single bin.

2.2. Behaviour Architecture

We can identify six high-level behaviours for each robotic agent: explore, avoid collisions, acquire rubbish, deposit rubbish, recharge and collaborate. Applying the Subsumption model, each of these high-level behaviours can be decomposed into multiple low-level behaviours of a lower complexity. These lower-level behaviours describe the basic actions which a robot is able to perform, and become "layers" in the subsumption architecture. The low-level behaviours are as follows:

Wander
: The robot wanders the environment at random.

Obstacle Avoid
: The robot steers away from physical obstructions in its path, halting when necessary to avoid a collision. Obstacles are treated as generating a repulsive force that directs the robot's steering. There is no need to identify obstacles as discrete objects; a simple direction and distance to the object provides sufficient information.

Go-to Rubbish
: The robot sees pieces of rubbish in the environment and moves towards them.

Pick-up Rubbish
: The robot detects when a piece of rubbish is within its grasp and picks it up.

Go-to Bin
: The robot sees the bin in the environment and moves towards it when carrying rubbish.

Drop Rubbish
: The robot detects when it is at the bin and drops any rubbish it is carrying into the receptacle.

Go-to Power
: The robot sees charging stations in the environment and moves towards one if its battery level is running low.

Recharge
: The robot detects when it is at a power station and docks with it, charging until its battery is sufficiently replenished.

Separation
: The robot discerns other robots that it is too close to and moves away accordingly. This behaviour brings the level of collaboration within the system to informed coexistence, as described in [13]. Actively recognising other robots and maintaining a safe distance is necessary to keep interference from limiting the performance of the team.

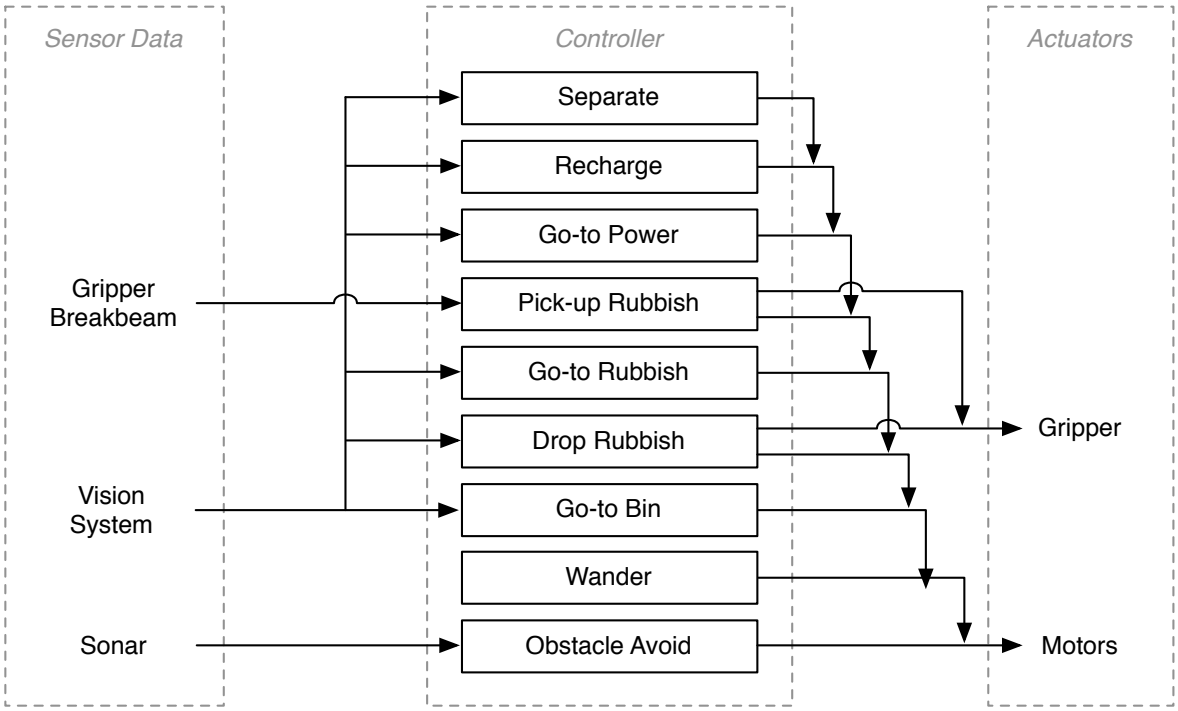

Figure 1. Overall subsumption architecture for garbage collection.

2.3. occam-π Sensor/Actuator Interfaces

Figure 1 shows the sensor input to each behaviour and the output to the robot's actuators. The robot is equipped with two forward-facing sonar sensors and a forward-facing camera. A gripper is mounted on the robot's nose that can both grab and lift items. Two infra-red break beams between the gripper's paddles detect when an item is within grasping range.

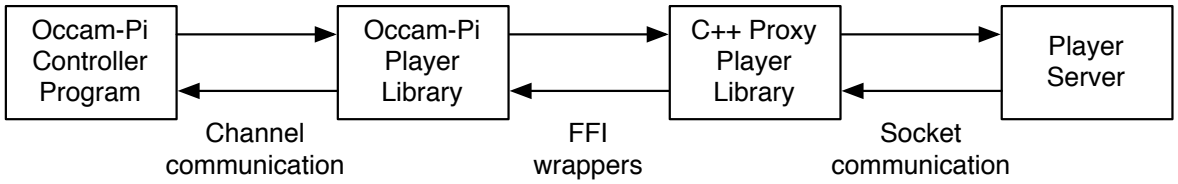

Figure 2. occam-π interface to Player.

Interaction with the hardware is handled by the Player robotics platform, a network-based abstraction layer that provides a standard interface to real and simulated robotic control hardware [10]. Player's C/C++ API is made available to occam-π programs using a wrapper generated automatically by the occam-π SWIG module [14]. Since this low-level API is not designed for concurrent programming, the occam-π `player` module provides a higher-level process-oriented interface, where Player devices are exposed as processes that communicate with the user's program using appropriate protocols [15]. This module was originally developed for the RoboDeb robotics software distribution, and includes predefined "brain stem" processes for several robotics platforms. The resulting architecture is shown in Figure 2.

2.4. Overall Design

Figure 3 shows a process network diagram for the garbage-collecting robotic control system, with dotted lines dividing the layers of the subsumption architecture. Processes are isolated as usual in a process-oriented system; they do not interact except through the channels shown.

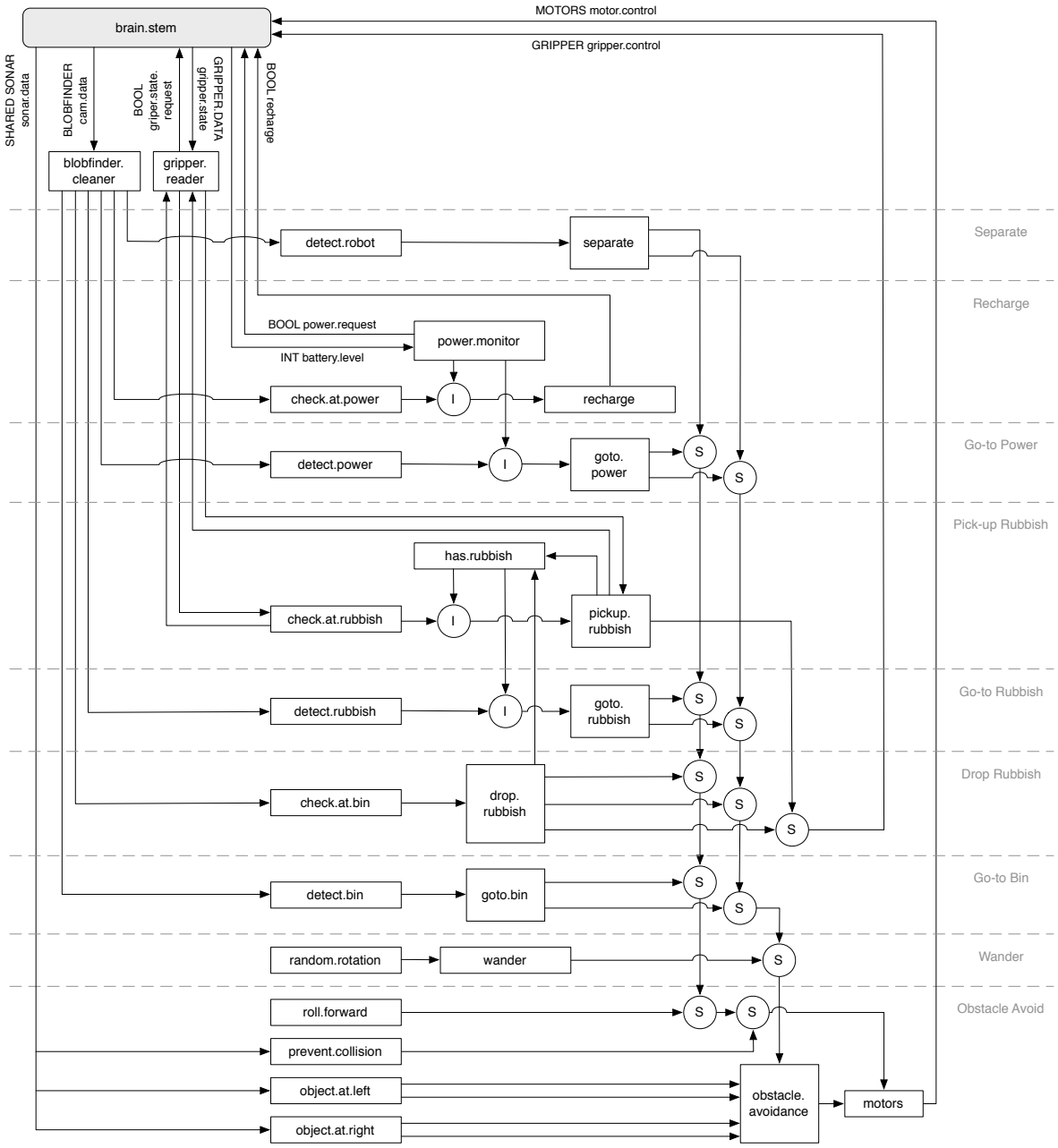

Figure 3. Process diagram for the subsumptive garbage collection controller.

A subsumption architecture works by inhibiting and suppressing control signals. Inhibitor ("I" in Figure 3) and suppressor ("S") components usually pass signals received on their input channel to their output channel, but differ in how they react to signals on their control channel. An inhibitor discards input signals when it receives a signal on its control channel, causing the output of upstream components to be lost. A suppressor instead replaces its input signals with those received on its control channel.

In Figure 3, the layers are shown in order of priority: the behaviours of the lower layers will be overridden by those of higher layers in turn in reaction to sensor inputs. We will now consider the design of the individual layers.

2.4.1. Obstacle Avoid and Wander

Figure 4 shows the process network for this behaviour. The `random.rotation` process produces a random rotation velocity at arbitrary intervals, output for a randomly generated duration. If

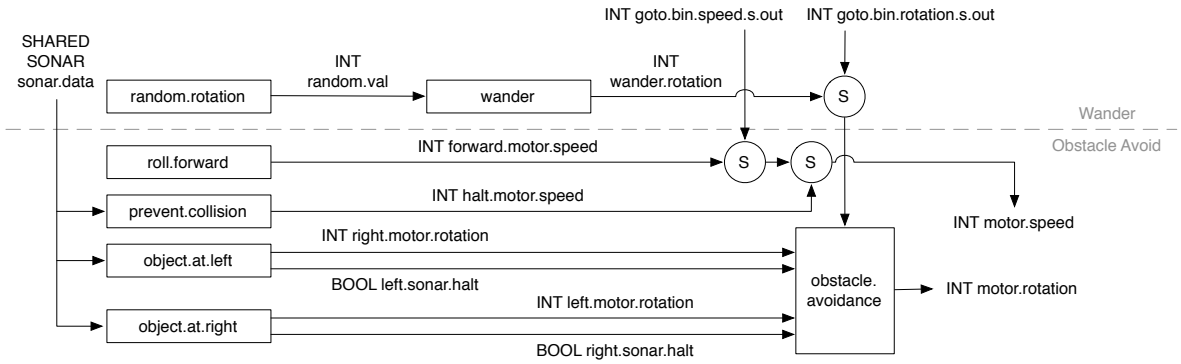

Figure 4. Process network for obstacle avoidance.

`wander` receives input from the `random.val` channel the process forwards the rotation speed, otherwise it outputs a rotation of zero. The `wander.rotation` output channel is suppressed by the rotation speed from the go-to bin layer and, via a transitive path through the suppressor chain, rotations from all behaviours that output motor commands.

The `obstacle.avoidance` process reads in both repulsive rotations as well as the output from the `wander.rotation` suppressor. An overall direction for the robot is calculated by combining the avoidance rotations with the rotation generated by higher behaviours (hereafter referred to as the behavioural rotation). The two avoid rotations are summed to produce a joint vector, allowing the robot to steer directly forwards when it detects obstacles on both sides – such as when travelling down a corridor. The repulsive vector is then compared with the behavioural rotation; if the two are in the same direction and the behavioural rotation is greater than the evasive rotation then the behavioural rotation is routed to the output. Otherwise, the avoid vector is routed to the output.

Summing the repulsive rotations does not produce the correct vector if moving towards a head-on collision. Objects directly in front of the robot cause the left and right range-finders to return diametrically opposing repulsions which sum to zero and produce a null turning vector. To counter this, the object detection processes transmit a Boolean halt message to `obstacle.avoidance` if an obstacle is sensed within a given "danger" range (defined as 0.5m). If either of these signals is received, `obstacle.avoidance` routes only the rotation from the corresponding object detection process to the output. If a signal is received on both `left.sonar.halt` and `right.sonar.halt` the right rotation is given arbitrary preference. This ensures that the robot turns away sharply from obstacles within a dangerous proximity, pivoting right in the face of a head-on collision. The robot is moved forwards by the `roll.forward` process, which outputs a constant motor speed on the `forward.motor.speed` channel. This channel is suppressed by two other outputs. The first suppressor takes input from the Go-to Bin motor speed output, providing a suppression path for motor speed outputs from all other behaviours in the system. Output from this suppressor is further suppressed by `prevent.collision` before being sent to the motors. The `prevent.collision` process monitors both sonar sensors, and outputs a motor speed of zero if either reports an obstacle within the defined danger range, halting the robot.

2.4.2. Go-To Bin and Drop Rubbish

Figure 5 shows the process network for this behaviour. The `detect.bin` process constantly examines data from the camera, searching for the bin. If a matching object is found, the process outputs its heading and range. Upon receipt of a heading, the `goto.bin` process outputs motor commands that suppress those of the wander and obstacle avoidance layers to move the robot towards the location described. The `check.at.bin` process monitors the camera to determine when the bin is close enough to deposit rubbish. This event is communicated to

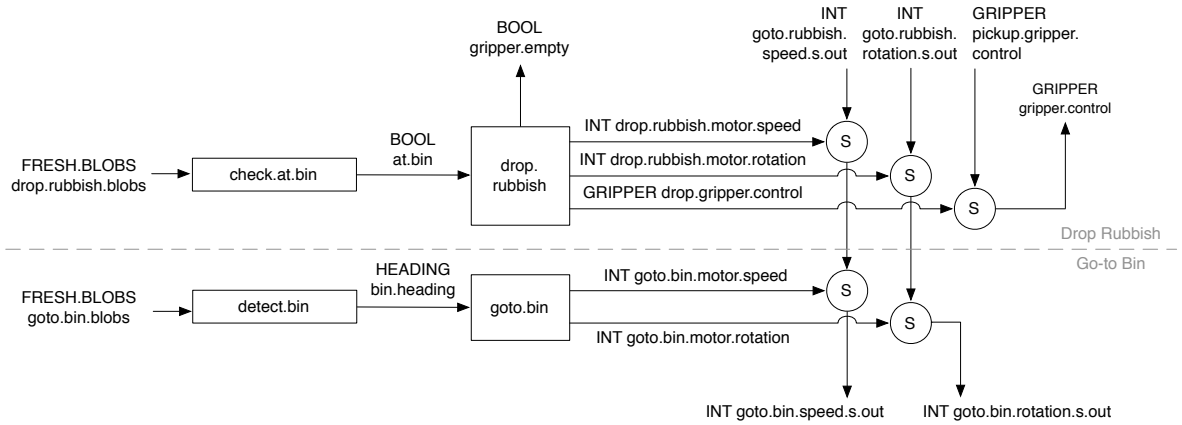

Figure 5. Process network for seeking bin and dropping rubbish.

the `drop.rubbish` process which in turn sends a command to the gripper to open its paddles. The process also outputs a message to the `has.rubbish` process in the Pick-up Rubbish layer to indicate that the gripper is free. This communication is made to a higher priority layer and is necessary to coordinate the acquire/deposit rubbish behavioural sequence.

2.4.3. Go-to Rubbish and Pick-up Rubbish

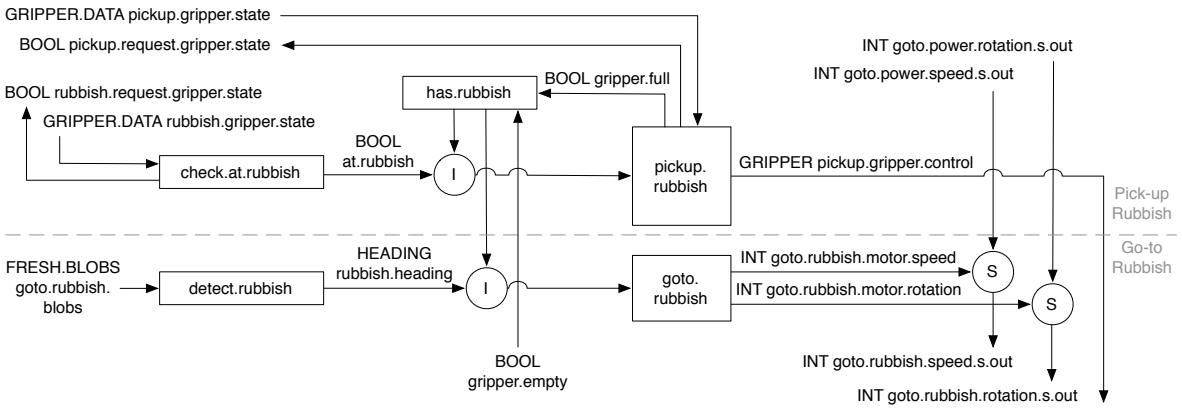

Figure 6. Process network for finding rubbish.

The Go-to Rubbish behaviour operates in much the same way as Go-to Bin, steering the robot towards rubbish rather than the bin; see Figure 6. When multiple items of rubbish are detected, the robot moves towards the nearest item. The `goto.rubbish` process manoeuvres the robot into a position where it is able to pick up rubbish, halting the robot directly in front of an item. The `check.at.rubbish` process constantly monitors break-beams between the paddles. If anything cuts these beams a message is sent to the `pickup.rubbish` process which then outputs a gripper command to close the paddles, suppressing the gripper output of the drop rubbish behaviour.

After the command has been sent, `pickup.rubbish` checks the state of the gripper. If the gripper's paddles are closed and the break-beams are broken, a confirmation message is sent on the `gripper.full` channel. Otherwise, a command is sent to the gripper to force the paddles open. As a result, if the robot fails to pick up the rubbish, its gripper will open again and the behaviours will loop until a successful collection is made or, if the rubbish is in an unobtainable position, the robot overshoots.

Once the robot has successfully picked up an item of rubbish, the `has.rubbish` process receives a message from `pickup.rubbish` and outputs parallel signals to inhibit the output lines

of `detect.rubbish` and `check.at.rubbish`. This prevents the Pick-Up Rubbish behaviour from taking control of the actuators, and so the robot will not move to pick up any more rubbish. The resultant effect is for the Drop Rubbish behaviour to have priority until the rubbish is dropped in the bin, at which point `has.rubbish` stops inhibiting and the rubbish-collection behaviours regain priority.

2.4.4. Go-to Power and Recharge

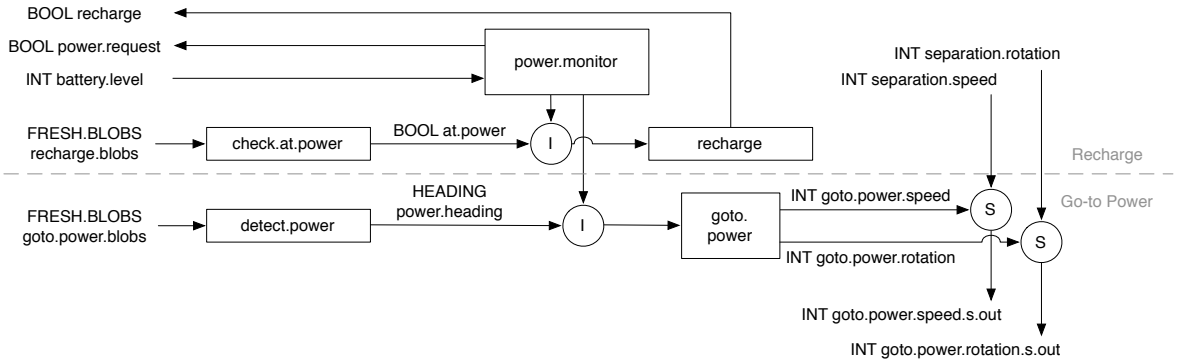

Figure 7. Process network for power recharge.

Figure 7 shows the process network for this behaviour. In general, the `power.monitor` process inhibits the recharging behaviours and so the robot will ignore any charging stations. The `power.monitor` process regularly examines the robot's battery level and stops inhibiting once the remaining power drops below a defined threshold. At this point, recharging becomes the priority for the robot and it will move to detected charging stations over any other attractor. Lower behaviours are still allowed to function while a charging station is being sought: if the robot comes across an item of rubbish and cannot see a charging station, it will move to collect the rubbish.

The `detect.power` process monitors the camera to identify charging stations, passing headings to `goto.power` to steer the robot. When the robot is docked at a charging station, the recharge module forces the robot to remain stationary until it has recharged. Once the battery level is replenished, `power.monitor` resumes its output of inhibiting messages and the robot is free to continue foraging.

2.4.5. Separation

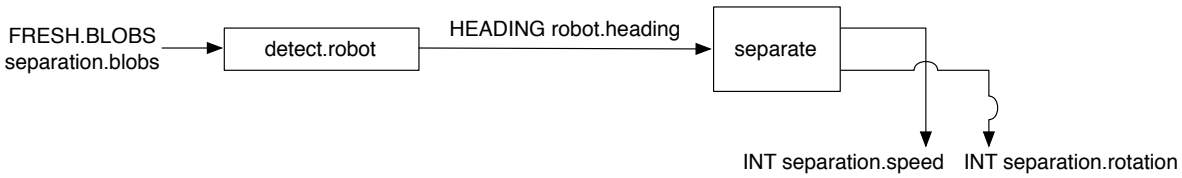

Figure 8. Process network for separation.

Figure 8 shows the process network for this behaviour. The Separation behaviour prevents head-on collisions with other agents by monitoring a 20° cone in the centre of the robot's field of vision. The `detect.robot` process examines the camera data for blue objects (robots) and passes the heading of the nearest detected robot to `separate`.

3. Performance Evaluation

We performed an experimental analysis of the system's completion of a set foraging task in order to quantitatively measure the performance of the controller.

3.1. The Simulated Environment

The control system was tested using the Stage simulation environment, which provides virtual robotic devices that can be controlled using Player. The RoboDeb software distribution provides a preconfigured occam-π, Player and Stage system that can be used inside a virtual machine. Simulation makes experimentation more convenient, and allows us to maintain precisely identical, controlled start conditions for multiple experiments. The use of Player also means that we could potentially use the control program unmodified on a real robotics platform.

Each simulated robot has a top-mounted PTZ camera with an 80° field-of-vision and a range of eight metres. To make it possible for Stage's simulated vision system to distinguish between objects in the environment, different types of objects are given different colours. Black objects are used to represent walls and fixed obstacles; they are not detected by the vision system, but can be detected by the robot's sonar sensors. The robot has two sensors, each of which faces 35° away from the forward normal of the robot and covers a 35° slice to a range of five metres.

The environment used was enclosed by an irregular boundary, 25m by 20m at its longest and widest points; the area of the search region was approximately 396m^2. This region includes non-uniform surfaces and a variety of obstructions. The task involved four robots, sixteen pieces of rubbish, a bin in the centre of the world and three arbitrarily positioned charging stations. Rubbish locations were picked using a random number generator with a uniform distribution.

The non-deterministic behaviour of the controller means that the robot's behaviour is not completely reproducible even given identical environmental conditions; it is necessary to average the results of multiple trials. The experiment was performed twenty times and spatial and temporal data was recorded for each trial. The completion criterion was defined as the successful deposition of all sixteen pieces of rubbish in the bin. Each experiment was subject to termination constraints, with a trial being halted if either:

- the trial took longer than twenty minutes to complete; or
- all four robots failed, either by unrecoverable collisions or behaviour stall.

3.2. Evaluation Metrics

The performance of our control system can be quantified in several ways. Full completion of the foraging task is defined as collecting all items of rubbish from the environment and successfully depositing them in the bin. Although largely qualitative, task completion provides a reasonable metric as to whether the implemented behaviours and coordination of the controller are effective.

Given the previously documented use of occam-π subsumption architectures [11,16], it can be reasonably expected that the implementation of a subsumptive foraging controller is possible. The proposed system coordinates many more behaviours than has previously been attempted with a process-oriented approach. The general level of complexity has been shown to be feasible in the subsumption architecture itself by the Herbert robot [17], albeit with some reliability problems. The following results detail the state of the system after the termination of each experiment, including both successful completions and halted trials.

4. Results

Figure 9 shows the average amount of rubbish collected and successfully returned to the bin, showing the mean number of items deposited and interquartile range. The mean success was fourteen returned items (rounded to the nearest integer). In five of the trials all of the rubbish was successfully deposited. The worst observed performance of ten returned items occurred once. In general, failure to return all of the items was due to collisions or behavioural stall, an intermittent fault that caused a robot to cease foraging. Agents affected by behavioural stall would continue to wander around the environment and avoid obstacles but would make no effort to pick up or deposit rubbish. We believe that this is caused by deadlock in part of the control network; this is discussed further below.

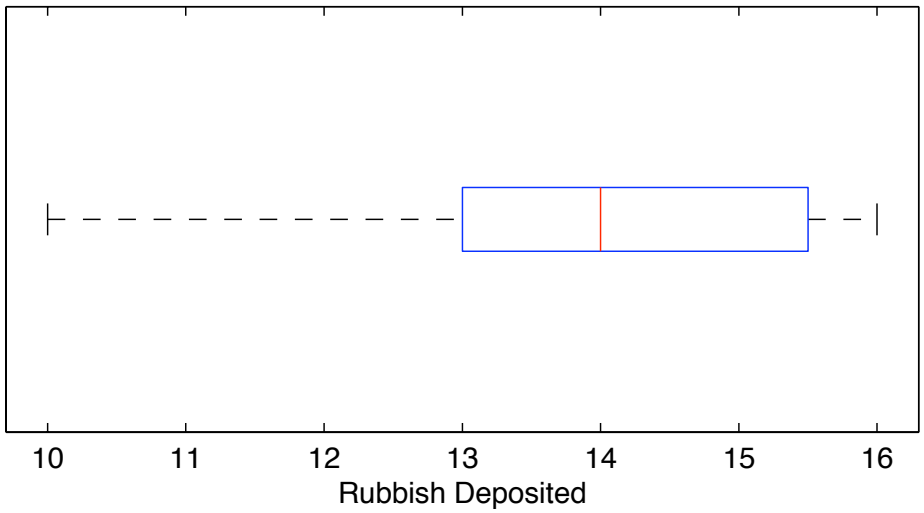

Figure 9. Box plot showing the amount of rubbish deposited in bin.

The controller itself is entirely capable of completing the task, as demonstrated by the observed "perfect" runs, but due to the discussed reliability issues complete success was not the usual case. In the majority of trials all of the rubbish in the environment was collected by the robots, but was not all deposited in the bin due to faults. Only four out of the twenty trials had uncollected rubbish in the environment at the point of termination.

The observed successful completion of the task provides evidence to support Hypothesis 1. It has been demonstrated that the developed controller is capable of collecting all items of rubbish in the experimental environment.

4.1. Reliability

During testing of the system we noticed some interesting behaviour. An intermittent fault emerged that affected the reliability of the system as a whole in terms of allowing the robots to complete their task. After a period of successful operation, usually after several collections and depositions of rubbish, the robot ceased to pick up rubbish or, if carrying an item, failed to drop it in the bin. The agent continued to function, wandering about the world and avoiding obstacles, but was unable to detect rubbish, the bin, charging stations or other robots. As both the wander and obstacle avoidance behaviours were unaffected by the presence of the fault it can be inferred that the failure occurs in one of the layers that makes reads from the camera. A likely cause for the problem is that one of the processes in the acquire rubbish, deposit rubbish, recharge or separate competencies locks; possibly due to an incomplete rendezvous. Such a failure will prevent a layer from reading input from the `blobfinder.cleaner`

process, blocking the process and preventing the transmission of camera data to all other dependent behaviours. We call this fault a *behavioural stall*, with affected robots known as *stalled agents*. The frequency of behavioural stall increased as more layers were added and so its presence was only discovered during extensive testing of the complete system. Retroactively stripping away behaviours made the failure progressively more uncommon; although it has not been observed without the recharging and separation layers a competency level at which behavioural stall conclusively does not occur cannot be established. Consequently, the fault could not be isolated to a single behaviour. It is possible that the error lies with the RoboDeb environment or experimentation programs, but this could similarly not be established. Due to behavioural stall the developed controller is somewhat unreliable. This is not uncommon with pure implementations of the subsumption architecture at this scale, as evidenced by the performance of similar controllers such as the Herbert robot [17]. Scaling issues with the subsumption architecture make it increasingly difficult to debug the behavioural network as its complexity increases. As the fault cannot be definitively isolated it is difficult to suggest possible solutions [18].) postulated a capability ceiling for the subsumption architecture as the level of complexity implemented by Herbert. The developed controller performs a very similar overall action and coordinates additional behaviours. Subsequently, the encountered reliability problems are potentially inherent to the subsumption architecture. Although only evaluated in simulation, the system has been shown to make several full completions of the foraging task.

4.2. Performance

Performance was measured as the average distance travelled by each robot and the time taken until the termination of the task. The results are somewhat distorted by agent failure; for example, the average completion time is biased by unsuccessful trials which were terminated at twenty minutes. To accommodate this, two sets of results are discussed: the average data for all trials, and the average data for the five trials which were fully completed. The average distance travelled by each robot is given in Table 1. As shown, the overall distance travelled was less during trials in which all of the rubbish was successfully returned. During these trials, agent failure was either minimal or non-existent, and so the work could be split evenly across many robots. If multiple agents failed, the remaining robots needed to cover more ground to collect the rubbish. There is a notable difference in the average distances travelled by each robot. This is likely a result of the interference caused by robot failures during measurement, as the distances have lower variation in the successful set. The average time taken until termination of the task is given in Table 2. Successful completion took an average of approximately ten minutes. As with distance travelled, the completion time averaged over all trials is longer than for successful tasks due to an increased proportion of agent failure.

Table 1. Average distance travelled by each robot (measured in meters).

Trials	R1	R2	R3	R4	Total
All	477.0	505.0	417.9	502.3	1872.2
Successful	405.2	421.4	368.2	400.0	1594.8

Table 2. Average time taken across all robots.

Trials	Time (seconds)
All	939.0
Successful	656.6

5. Reflection

We successfully implemented a standard real-world problem using a process-oriented subsumption architecture, and demonstrated its ability to successfully complete the evaluation task.

We found that the process-oriented programming model provides a solid foundation for the implementation of the interacting layers of a subsumption architecture. The implementation requirements of a subsumption architecture translate straightforwardly into networks of concurrent processes, yielding a relatively simple progression between design and implementation. The inherent modularity and composition of a process-oriented system, when combined with the design principles of subsumption, means that many of the components developed for this work could be reused in other subsumptive control systems. The use of composition to add and remove features of the control system was found to work effectively, even in more complex programs.

A fundamental difference between the two approaches in communication between components causes problems in systems of this size. Process-oriented design techniques, and the occam-π language in particular, focus on synchronous communication, where failure to receive a message from a process will prevent that process from executing any further. Subsumptive architectures assume asynchronous messaging: if a signal is not received, it is simply discarded without blocking the sender. Implementing subsumptive components in occam-π requires considerable effort to prevent unwanted blocking while minimising the number of synchronisations required. Process-oriented programmers usually rely on design patterns such as I/O-PAR or Client-Server to achieve safe communications [19], but few patterns have yet been identified for asynchronous programming within the process-oriented model. This is clearly an area for future work, since many process-oriented environments now provide facilities for asynchronous messaging. We need design patterns that let us construct safe subsumptive networks with minimal synchronisation overhead in the same way we presently construct Client-Server networks.

Debugging the system presented a number of significant problems. Besides behavioural stall – which we believe is caused by deadlock in part of the control system – the current version of the controller sometimes collides with objects or deposits rubbish just outside of the bin. It is difficult to tell whether these are deficiencies in the design of the subsumption architecture or faults in the implementation. Tracing the behaviour of individual processes with the Transterpreter is possible, if awkward (especially on embedded devices) – but these kinds of problems result from the complex interactions between processes, and we currently have no way of visualising and exploring these interactions. It seems likely that these behavioural problems are caused by the delay between sensor input being read and actuator output being applied, but we cannot easily measure this effect at the moment.

6. Conclusions

We have described our experience with the application of process-oriented programming and subsumptive robotic control to a relatively complex swarm robotics task. Motivated by the attraction of concurrency for implementing a system using an architecture which is naturally concurrent, we designed, implemented and tested a subsumptive control system for a standard robotics problem. This has been a qualified success. Design and implementation identified significant advantages of the modular and compositional concurrent style permitted by the process-oriented approach. The use of Player as an abstraction layer means that we can conduct experiments both in simulation and using real robots (“embodied simulation”) – or using a mix of the two.

Our implementation is capable of acceptable runtime performance – but suffers from hard-to-debug problems under specific conditions. We feel these problems result from deficiencies in the design approaches and tools currently used by process-oriented programmers, especially when applied to asynchronous and more complex systems. We believe that these problems were not identified in previous work involving the combination of process-oriented and subsumptive approaches due to the simplicity of previous case studies.

If these problems can be addressed, by the development of new design patterns and more effective debugging tools, we believe that process-oriented subsumptive techniques present considerable potential for application in swarm robotics. We would like to be able to construct subsumptive control systems that span multiple control systems – for example, having one robot be able to suppress behaviours within another robot, or make decisions based on a repository of information within the environment – without the need for a centralised control system. The process-oriented approach allows reasoning about communication between distributed systems with the same semantics as local communication, meaning that design patterns developed for single robots can be applied across swarms.

Acknowledgements

This work is part of the CoSMoS project, funded by EPSRC grants EP/E053505/1 and EP/E049419/1.

References

[1] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[2] Peter H. Welch and Frederick R.M. Barnes. Communicating Mobile Processes: Introducing occam-π. 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.

[3] Peter H. Welch. Process Oriented Design for Java: Concurrency for All. In P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, and A.G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 687–687. Springer-Verlag, April 2002.

[4] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. PyCSP Revisited. In Welch et al. [20], pages 263–276.

[5] Jonathan Simpson and Carl G. Ritson. Toward Process Architectures for Behavioural Robotics. In Welch et al. [20], pages 375–386.

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

[7] Christian L. Jacobsen. *A Portable Runtime for Concurrency Research and Application*. PhD thesis, University of Kent, Canterbury, Kent, England, December 2006.

[8] M. Dorigo and E. Sahin. Special issue on swarm robotics. *Autonomous Robots*, 17, 2004.

[9] E. Sahin and A. Winfield. Special issue on swarm robotics. *Swarm Intelligence*, 2((2-4)), 2008.

[10] Brian Gerkey, Richard T. Vaughan, and Andrew Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the 11th International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, Coimbra, Portugal, June 2003.

[11] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. Mobile robot control: The Subsumption Architecture and occam-π. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering*, pages 225–236, Amsterdam, The Netherlands, 2006. WoTUG, IOS Press.

[12] A. Winfield. *Foraging Robots*. Springer, 2009.

[13] M. Mataric. Designing emergent behaviors: From local interactions to collective intelligence. In *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, volume 2, pages 432–441, 1992.

[14] Damian J. Dimmich and Christan L. Jacobsen. A Foreign Function Interface Generator for occam-π. In Jan F. Broenink, Herman W. Roebbers, Johan P.E. Sunter, Peter H. Welch, and David C. Wood, editors,

Communicating Process Architectures 2005, volume 63 of *Concurrent Systems Engineering*, pages 235–248, Amsterdam, The Netherlands, 2005. WoTUG, IOS Press.

[15] Christian L. Jacobsen and Matthew C. Jadud. Concurrency, robotics and robodeb. In *AAAI Spring Symposium on Robots and Robot Venues: Resources for AI Education*, Stanford, Palo Alto, CA, 2007. Association for the Advancement of Artificial Intelligence.

[16] J Neeson. Occam-Pi for Multiple Robotic Systems. Master's thesis, University of York, 2008.

[17] Jonathan H. Connell. A colony architecture for an artificial creature. Technical report, Cambridge, MA, USA, 1989.

[18] E. Gat. Three-layer architectures. In R. Kortenkamp, R.P. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 340–366. AIII/MIT Press, 1998.

[19] Adam T. Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, University of Kent, October 2010.

[20] Peter H. Welch, Herman W. Roebbers, Jan F. Broenink, Frederick R.M. Barnes, Carl G. Ritson, Adam T. Sampson, Gardiner S. Stiles, and Brian Vinter, editors. *Communicating Process Architectures 2009*, volume 67 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2009. WoTUG, IOS Press.