High Level Paradigms for the Structuring of Concurrent Systems

a thesis submitted to
The University of Kent at Canterbury
in the subject of Computer Science
for the degree
of Doctor of Philosophy.

By
Jonathan Simpson
March 2012

Generated: 23/06/2014

Abstract

The need for software concurrency and the availability of hardware parallelism exist in symbiosis; the ability to exploit parallelism via concurrency is an increasingly important part of software design and development. Process-oriented design provides a model, of systems composed from networks of concurrently executing communicating processes, which eases the creation of concurrent programs that behave correctly. Robots are agents in the world and typically have many tasks to achieve simultaneously; writing robot control software requires coordination of these tasks. Given their interaction with the world, the need to complete concurrent tasks and the availability of parallel hardware, it is natural to consider designing and implementing robot control systems using a programming model specialised for concurrent systems.

This thesis presents the application of process-oriented design and programming to robotic control through the re-design and re-implementation of existing software architectures and hardware interfaces. The contributions made in this thesis demonstrate properties consistent with a *closeness of mapping* between the domain and programming model. Evidence is presented that application of a concurrent process-oriented programming model does not negatively effect the responsiveness of systems and facilitates more direct representation of the concurrency inherent to the task.

This thesis presents a visual program design tool, POPed, which is able to produce executable programs using process-oriented design techniques involving the structured composition of processes. Work reported on POPed demonstrates that process-oriented programs may be created via pure composition using high-level design techniques. This thesis also presents a methodology and tool for visualising the runtime state of process-oriented programs, further extending the applicability of the design visualisations from program creation to debugging. Evidence is presented of this tool allowing exposition of program properties which lead to common concurrency errors (livelock and deadlock) in process-oriented programs.

Acknowledgements

I wish to thank all of those with whom I have interacted personally and professionally throughout the Ph.D. process. Your number is greater than I could hope to list — you made the process memorable and enjoyable.

I would like to give thanks to my family: Ronald, June and Stephen. I cannot thank you enough for your unconditional support in all that I do.

To Christian, Damian and Matt: you piqued my interest and started this journey. I am proud to count you not only as collaborators but friends and owe you a debt of gratitude for support throughout. May our paths cross long into the future.

To Carl: for being the true origin of the bouncy medicine ball[1], an excellent office-mate and good friend. Your insight and advice has been to my aid on many occasions.

To Professor Peter Welch and Doctor Fred Barnes — for agreeing to supervise me and for your feedback and support throughout. The Concurrency Research Group provided a thought provoking and exciting environment in which to work, I thank you for allowing me to be a part of it.

To my examiners, Professor Alan Winfield and Professor Michael Kölling — it is a privilege to have been examined by yourselves and I am grateful for your time and thoughtful input, which has been to the benefit of my work.

Finally, my thanks to the School of Computing for funding and support, in particular the Course Administration Office for their aid in navigating the process throughout.

[1] Context may be found in the acknowledgements of Dimmich's Doctoral Thesis [Dim09].

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Process-oriented Programming is a model of concurrent programming based on Hoare's Communicating Sequential Processes [Hoa85]. In this thesis, the definition of process-oriented programming is influenced directly by Inmos' occam programming language [INM84]. Concurrent programming languages modelled on Hoare's CSP predate occam, a notable example being Brinch Hansen's Concurrent Pascal [BH75]. Despite this, the directness of the occam implementation of the CSP model makes it an exemplar for practical application of the model to programming.

In the process-oriented model problems are decomposed into networks of concurrently executing processes communicating with each other via message passing over channels. Processes may encapsulate serial program logic, parallel compositions of other processes or any combination of the two. Channels provide a mechanism for processes to communicate data, by reading or writing messages. Processes *block* waiting to read from or write to a channel, until the other party in the communication is ready to complete the action, producing a synchronisation event.

The work presented here applies these process-oriented programming concepts in the domain of robot control. At its simplest, a robot is a machine capable of autonomously completing a task. While it is possible for a robot to have no sensors, no processing capability or no actuators, all of these are typically found on a robot and creating programs which encompass them to sense, plan and act forms the basis of robot control. A robot uses its sensors to gather input data about conditions in the world (sense) and computes this input (plan) into activity of the robot via mechanical effectors (act), changing the robot's state and consequently the state of the world.

Robot control is an inherently concurrent problem. Robots interact with the world, where events are independent and can happen simultaneously — the world is inherently parallel. As humans, our experience of the world is also parallel; we see, hear, taste, smell and feel at the same time, our sensory stimuli being simultaneously processed by the brain into our comprehension of the world around us. Given this human relation to the world, it is natural to express the behaviour and decision making of a robot in terms of concurrent activity, rather than as a series of sequential or interleaved actions. While this may be obvious to state, we can only take it as a "given", when expressing control logic in a process-oriented programming language designed for the effective use of concurrency.

It is important to distinguish between *concurrency* and *parallelism*. *Concurrency* is a structuring tool, allowing the expression of tasks which should happen independently of one-another and the communication or synchronisation points between them. *Parallelism* is a performance tool, extracting program execution speedups from the concurrency of a program by executing code simultaneously instead of serially where more than one processing resource is available. Concurrency is valuable even in the absence of parallelism, as it allows more straightforward expression of programs which solve *naturally concurrent* problems.

The inherently concurrent nature of robot control means application of the process-oriented programming model reduces the level of abstraction between the behaviour of a robot control system and the expression of that system in software. When introducing and motivating the use of process-oriented programming, robotics presents an application area in which the challenges of the problem domain present opportunities to employ the strengths of the programming model.

Green and Petre define a cognitive dimension in the use of visual programming called *closeness of mapping*: given that programming requires mapping between a problem domain and a program world, the closer a program world is to the problem world, the easier solving the problem should be [GP96]. This thesis proposes that a closeness of mapping exists between process-oriented programming and robotic control; handling concurrency in a robot control program is eased by the availability of language primitives and a programming model inherently designed for concurrency.

Diagrams are an inherent and important part of the process-oriented model; the practice of process-oriented programmers is to represent program designs consisting of many concurrently executing processes using box and arrow diagrams. These diagrams do not deal with every small detail of the sequential logic of the program, rather they are a high level decomposition of the task. Process-oriented programmers often generate these diagrams

as part of their workflow; while designing the program, decomposing the problem into a set of communicating processes; and when debugging, annotating the program with state. The first interaction with the model of undergraduate students learning process-oriented programming at the University of Kent is labelling such a diagram based on a corresponding piece of program code.

Visual programming is the creation of programs through manipulation of graphical representations of program elements. Whilst process-oriented languages like occam are not visual, using textual syntax to represent program code, the role of diagrams in representing the high-level structure of process-oriented programs is a natural area in which to apply visual programming techniques. Visual representation draws, builds on and reinforces elements of the design process and mental model of process-oriented programs. It also gives programmers a different perspective on the structure of their program — the program code they write and the diagrams that correlate to it are two perspectives on the same construction.

Visual representations elucidate program structure for design, composing and connecting process networks via diagram, and for debugging at run-time, drawing process network diagrams annotated with program state. The use of these representations in software tools presented in thesis captures a practice of process-oriented programming for over twenty years — the use of process network diagrams as a model for designing and reasoning about process-oriented programs. This thesis proposes that *closeness of mapping* exists between process-oriented programming and visual representations of process-oriented programs; mapping between graph-like diagrams and connected networks of processes.

1.1 Origins

This work originates in the author's initial exposure to process-oriented programming as a first-year undergraduate student seven years ago, writing small robot programs in occam-pi on the Lego Mindstorms RCX via Jacobsen and Jadud's original exploratory work on a runtime environment [JJ04]. The port was experimental and had rough edges; with extremely limited space for program memory, hardware interfaces grew organically as new capabilities were required and errors would occur outside of the robot program, both in the runtime and hardware interface code. The ability to express robot control via communicating concurrent processes allows the structure of the software to directly reflect the multiple concurrent behaviours expected of the control system. Robots inherently require concurrent behaviours,

and having first class support for concurrency in the programming language brings the expression of the solution closer to problem domain.

Whilst later formal training in process-oriented programming as part of the course formed a solid foundation in the use of occam-pi and concurrent software design, the original motivation captured in the first few steps taken on the Mindstorms RCX was formative. The research direction of the author, through publications, and the content of this thesis flow directly from this first experience. Each element of this thesis addresses a part of that original experience and aims to extend the field, creating a richer set of knowledge, experience and principles for conducting process-oriented robotics.

An educational theme runs throughout this thesis and the work presented at a micro scale; a desire to provide motivation, tools and principles for the effective application of process-oriented concurrency. Growing availability of hardware parallelism heightens the importance to students of being capable and experienced in building concurrent systems. Principles and practices introduced via the process-oriented model of programming are generally applicable and aid in reasoning about design and implementation of concurrent systems.

The author has accumulated several hundred hours experience of teaching programming at the University of Kent with both object-oriented and process-oriented paradigms. The author's experience of teaching object-oriented programming is with the Java programming language in the BlueJ environment [KQPR03]. The use of graphical representations in BlueJ to convey the program structure and its facilities for inspecting running program components have been formative to the author in designing tools which better support the use of process-oriented programming.

1.2 Relation to the Author's Papers

This thesis presents contributions and material which have been previously published in peer-reviewed papers by the author, in collaboration with a number of co-authors. These consist of eight conference papers, six of which have primary authorship or contribution by the author of this thesis and two to which the author has made smaller contributions. The relation of these publications to this thesis is described below, specifying the contributions of and role of the author of this thesis in each.

This initial research interest of the author was driven by a desire to replicate existing robotic architectures and programming strategies in the process-oriented model, to replace ad-hoc

process decompositions with tried and tested techniques. *Mobile Robot Control: The Subsumption Architecture and occam-pi* by Simpson, Jacobsen and Jadud, [SJJ06], describes the implementation of Brooks' Subsumption Architecture in occam-pi and the subsequent implementation of a bump and wander program using the approach. The author co-conceived the idea with Jacobsen and Jadud, and subsequently completed the research and implementation of both the subsumption architecture components and case study program. This work is detailed in Section 3.3.1.

Work on this initial architecture paper inspired the author to improve platform support on the Mindstorms RCX, as the original port left little space resources for larger, more complex programs. The result, a minimal port of the Transterpreter occam-pi run-time with an emphasis on allowing user programs maximal use of the RCX's limited resources, along with a process-oriented hardware interface was reported in *A Native Transterpreter for the LEGO Mindstorms RCX* by Simpson, Jacobsen and Jadud [SJJ07]. This work is partially detailed in Section 3.1.5, covering the process-oriented API to the robot hardware, sensors and actuators. The author completed the port, interface design and implementation with advice and input from Jacobsen and Jadud based on their original port to the RCX.

Given this enhanced RCX port, an opportunity arose to make use of it in a workshop for professional embedded software developers presented in Vienna, Austria. This workshop was co-presented by Jadud, Jacobsen, Dimmich and the author, introducing process-oriented programming to these developers through a structured course, applying of patterns and principles from occam-pi programming to robot control the RCX. *Patterns for Programming in Parallel, Pedagogically* by Jadud, Simpson and Jacobsen [JSJ08] reviews this workshop. The author's contribution is to the teaching material and the subsequent documentation and analysis of the workshop as a paper. This work is detailed in Section 3.5, discussing the application of patterns which originated in process-oriented programming to robotics.

In both the subsumption architectures work and the Mindstorms RCX port, use of diagrams was identified as future work; subsumption architectures are designed using network diagrams showing the key interactions, while the Mindstorms RCX ships with a graphical language in its standard incarnation. Building a visual tool for manipulating these process-oriented robotics programs, at a compositional level, represents a logical intersection of these concepts.

The design and rationale of a visual environment for creating robot programs is presented in *Visual Process-oriented Programming for Robotics* by Simpson and Jacobsen [SJ08]. Advised by Jacobsen, due to his previous experiences in the area with POPExplorer [Jac06], the author

investigated a number of previous tools to establish a working feature set and initial design. This work forms the basis of Chapter 4.

Whilst working on a visual environment for creating programs the adaptation of such visualisations to represent the execution state of process-oriented programs became an additional research avenue. To facilitate this work, Ritson added a debugging infrastructure to the Transterpreter virtual machine and end-user tool, *TC1* to demonstrate its capabilities. *Virtual Machine-based Debugging for occam-pi* by Ritson and Simpson[RS08] presents this tool, *TC1*, and the debugging infrastructure created to enable run-time monitoring of programs. The author contributed initial concepts, design and literature review material for this work, which is presented in chapter 5.

Safe Parallelism for Robotic Control by Jadud, Jacobsen, Ritson and Simpson [JJRS08] presents details of a port of the Transterpreter Virtual Machine to the Surveyor SRV-1 and its application in a first course in robotics at Olin College. The author contributed API and firmware design for the Surveyor port, and this work is discussed in Chapter 3.

Expanding on the previous work done to implement subsumption architectures in occam-pi, *Toward Process Architectures for Behavioural Robotics* by Simpson and Ritson [SR09] explores the implementation of primitives and principles for use of a number of seminal behavioural control architectures. The author conceived this work, collaborating with Ritson on the primitive implementations, which are presented in Section 3.3.

Process-Oriented Subsumption Architectures in Swarm Robotic Systems by Posso, Sampson, Simpson and Timmis [PSST11] presents the implementation of a process-oriented subsumption architecture to perform a garbage collection task. The control program is designed to run as part of a foraging swarm, with many robots operating with the same program. The author of this thesis had a minor role in this publication, advising on implementation of the subsumption architecture on the robot and adding depth to the writeup. This work is also part of the process-oriented subsumption architecture research conducted by Neeson, Posso and Timmis at the University of York originating from the author's original publication and the work presented in this thesis, this line of research is discussed further in Section 1.3.

1.3 Impact

Published work detailed in Section 1.2 has demonstrated research impact, having been cited in other studies and reviews. This impact is both conceptual, contributing to the understanding

of the field and instrumental, documenting methods and principles for conducting similar works.

Patterns for Programming in Parallel, Pedagogically has provided a data point for discussion of models to use for teaching concurrency patterns in Computer Science. Process-oriented programs demonstrate a model without shared state focused on communication and, when written in occam-pi, a language designed solely to clarify the expression of such programs. This model can be presented as contrast to pragmatic threads and locks approaches and conventional programming with shared state [Fek09, Bun09]. The significance of simplified, process-oriented patterns in student's interaction with the message passing model is noted by Gross, designing a course across a broad set of Java concurrency libraries, specifically the use of process-oriented programming patterns in constructing distributed systems using MPI [Gro11].

The work and principles described in *Mobile Robot Control: The Subsumption Architecture and occam-pi* have been directly extended by two MSc. projects at the University of York. Neeson reimplemented an existing swarming subsumption architecture, previously implemented in C++, using occam-pi to evaluate the use of the language for robotics, concluding that the occam-pi approach provided greater flexibility, portability and a less abstract expression of the architecture [Nee08]. Posso examined the suitability of occam-pi subsumption architectures to solving complex robot control problems and effects of scaling the approach to larger problems, concluding that issues found when scaling seemed to be inherent to subsumption architectures and that occam-pi provides "an excellent platform for concurrent control" [Pos09].

The research direction of this thesis has a symbiosis with teaching of the "Concurrency Design and Practice" undergraduate module at the University of Kent. Having taught this course for the last five years, the author of this thesis has introduced robotics elements building on previous efforts by Jacobsen and Jadud. [JJ05]. This work takes the form of an additional assessment called *Life on Mars*, designed to allow students to practice using concurrency in the context of robot control. This assessment has run as part of the course since 2009 and is described fully in Section 2.6.3.

Jacobsen, Jadud and Sampson have also built on the principles established for introducing process-oriented programming, creating a software library and book introducing occam-pi and electronics via the Arduino embedded board [JJS10]. The Arduino is an inexpensive and widely available embedded system based on the Atmel AVR micro-controller, designed to provide an accessible platform for electronics projects.

1.4 Contributions

Research questions addressed in this thesis include the following. Can process-oriented design and implementation techniques reduce the level of abstraction, producing a closeness of mapping between the problem domain of robot control and its expression in control software? Can a compositional visual program design tool allow the creation of useful process-oriented systems? Can the dynamic evolution of system state in a running program be captured in visual representations of process-oriented systems which allow reasoning about program behaviour?

In investigating these research questions, this thesis makes the following contributions:

1. **Process-oriented robotics architectures.** This thesis presents process-oriented re-design and re-implementation of seminal behavioural control architectures and hardware interfaces for robotics. This thesis also presents fundamental patterns of process-oriented programming as applied to robot control. This work is presented with the aim of demonstrating a closeness of mapping between the problem of robot control and the use of process-oriented concurrency which eases the expression and implementation of robot control systems. Evidence is presented that shows these approaches are practical, maintaining response times when providing concurrent abstractions (Chapter 3).

2. **POPed— A compositional, visual process-oriented program design tool.** This thesis presents the design and implementation of a visual programming tool designed purely for process composition. Evidence is presented that such a tool can use the visual forms employed for program design in a purely compositional way along with a predefined set of components to generate programs (Chapter 4).

3. **A methodology and tool for state visualisation (introspection) of process-oriented programs.** This thesis presents the design and implementation of a tool for the visualisation of the run-time state of process-oriented programs in occam-pi. Evidence is presented for the application of this tool to the demonstration and explanation of designs which cause concurrency errors in process-oriented programs. This thesis demonstrates that the visualisations used to design programs may effectively be annotated with state in order to reason about run-time behaviour (Chapter 5).

1.5 Structure

Chapter 2 *Motivation and Background* sets the scene for this work, covering the background and other research which has informed the content of this thesis. Chapter 3 *Robotics* presents the contributions of this thesis relating to process-oriented robot architectures. This begins with a number of existing robot architectures implemented using process-oriented concurrency in occam-pi to form primitives and principles for process-oriented robotics. Chapter 4 *A Demonstrator Environment* presents the contributions of this thesis relating to the design and implementation of a demonstrator tool for visually composing process-oriented programs. Chapter 5 *Introspection* presents ideas for and the design of an introspection tool for examining running occam-pi programs, including a comparison to the eventually implemented tool. Chapter 6 *Conclusions and Further Work* contains conclusions over the contributions made and experimental work presented in this thesis. This chapter also details the potential for further explorations of the ideas and experiments contained within this thesis, based on the conclusions made.

Chapter 2

Motivation and Background

The work reported in this thesis originates from a desire to expand and explore process-oriented robotics from an educational context, as detailed in Section 1.1. The motivations for this work are numerous and overlap each other along the research direction; the growth of hardware parallelism, the sympathy between process-oriented programming and robotics, the pedagogy of robotics for computer science education and the pedagogy of process-oriented programming itself combine to provide a fertile area for research. This chapter builds the landscape; from advances in multi-core hardware through process-oriented programming and its embodiment in occam-pi to robotics and the pedagogy of the two both in isolation and combination.

2.1 Trends toward Multi-core and Many-core

The significance of any work involving the application of concurrent programming has a direct relation to widespread and growing hardware parallelism; given parallel systems, capable of executing tasks simultaneously, expressing the concurrency in a program is critical to making effective use of the available performance. Advances in computer technology have turned away from increasing CPU clock frequency speed and straight-line execution performance towards adding cores and increased hardware parallelism [Suto5]; programmers can no longer rely on the advance of technology to speed up the execution of sequential programs. The need for concurrency and programmers trained in its practice has never been greater, and it is essential that programmers are prepared with approaches, tools and experience for building and debugging highly concurrent programs. To this end, the ACM/IEEE Computer

Society Task Force on Computing Curricula's Curriculum 2013 identifies parallelism as a key knowledge area to be addressed in computer science curricula, referencing the "vastly increased importance of parallel and distributed computing" [oCC13].

Moore's law, often mischaracterised as offering a doubling of CPU clock frequency, predicts a doubling in transistor count every 18 months [Moo65]. Historically this rise in transistor count facilitated increasing straight-line execution speed and processor clock frequencies, meaning single threaded systems gained performance over time. From 2006 Intel moved from doubling clock frequency to adding cores to increase performance, citing the difficulties with transistor heat generation at ever decreasing size and increasing frequency [Int04]. As an indication of its future direction, Intel demonstrated the Tera-scale architecture, a processor with eighty cores, to emphasise the need for developers to embrace and develop models that scale with this growth to many-core.

The trend toward multi-core around this period is not limited to Intel and desktop computing; Sony, Toshiba and IBM formed a partnership which yielded the Cell Broadband Engine (Cell BE) in 2005. The Cell BE is a on-chip combination of a general purpose Power CPU core and eight specialised, highly vectorised, stream processing cores known as Synergistic Processing Elements (SPEs) able to communicate with the Power core. The Cell's heterogenous architecture makes it difficult to extract performance from, programmers must make effective use of the available parallelism as the main horsepower of the system is in the multiple SPEs. The Cell BE can be commonly found in homes today inside the Sony Playstation 3 console; its major competitor, the Microsoft Xbox 360, uses three 3.2GHz Power architecture CPU cores similar to the Cell's main core. In comparison to the Cell architecture in the Playstation 3, the Xbox 360's three general purpose core architecture provides high performance of sequential code, making it more suitable for a programmer to extract performance from using coarse-grained concurrency techniques [Arc11]. The seven specialised vector processing units of the PS3 require an approach which handles mass parallelism and the heterogeneity of the cores. Dimmich discusses and investigates allowing the exploitation of the architecture and performance of the Cell BE via process-oriented programming in [Dim09].

With the progression of this shift in processor performance scaling, multi-core architectures have become prevalent, increasingly moving into embedded systems. Embedded systems are particularly relevant for robotics applications, and these are also moving toward multi-core, with processors such as the ARM Cortex-A9 [ARM12] available up to quad-core in 2007 and the quad-core XMOS xCORE XS1 [XMO12] in 2008. More recently many-core embedded systems have started to become available such as the Parallela, a board containing a Dual-

core ARM A9 processor alongside a 16 or 64 RISC core Epiphany co-processor, designed to use just 5 watts of power [Par13]. These heterogeneous architectures encourage approaches which facilitate distribute processing; moving program components between processors and processor architectures, without requiring access to shared memory.

Robot platforms specifically present a challenge for the exploitation of parallelism as their architectures can feature multiple heterogeneous processors and the requirement for distributed processing between them. The ActivMedia Pioneer 3-DX (further discussed in section 3.1.4) presents an example of such an architecture. The Pioneer 3-DX, when equipped in its laser mapping and navigation specification, contains three co-existing systems: an analogue frame grabber connected to a video camera, an embedded "motion controller" board to interface with sensors and actuators, and an embedded PC104 board acting as an on-board PC host to which the two other systems are interfaced Similarly, the LEGO Mindstorms NXT contains both a main 32-bit ARM CPU and an 8-bit AVR co-processor which performs power management and handles reading from the sensors and buttons on the brick [Theo6]. Having multiple, distributed systems involved in control introduces problems of synchronisation, communication and concurrency; the control program must be able to wait for input from a specific subsystem without becoming unresponsive to input from other subsystems.

The ability to better express the concurrency in a program leads directly to an improvement in parallelism; if the user can express to the machine exactly which sections of the code may execute independently, this information can be used to inform parallel execution by the run-time system. As the availability of parallel execution hardware increases, the parallel speedup available is dictated by Amdahl's Law [Amd67]: the speedup available is limited by the number of sequential sections in the program.

The granularity of expression for concurrency in a model and programming language directly affects the potential for the runtime environment to execute the program in parallel. As hardware parallelism moves from multi-core toward many-core, effective use and exploitation of performance will require moving from fewer, more heavyweight concurrency primitives and locks on shared data to many lightweight primitives, avoiding shared data entirely.

Highly concurrent systems require approaches beyond threads and locks; parallel shared memory systems place the burden on the programmer to ensure synchronisation of access to data. This burden is the source of common concurrency errors, explained further in Section 5.2. Increases in parallelism mean this synchronisation must be finer-grained to extract full performance from the hardware. It is important to give programmers tools, approaches and mental models to build systems that fully exploit highly concurrent and

non-heterogenous architectures. Models which enable use of concurrency, including process-oriented programming, enable the creation of systems which make effective performance gains on many-core systems.

2.2 Process-oriented Programming

Process-oriented programs are composed of networks of concurrently executing processes, communicating over channels via synchronous message passing. The process-oriented paradigm is derived from Hoare's Communicating Sequential Processes (CSP) process algebra[Hoa85]. Process-oriented programming provides a model for writing concurrent programs using a small number of primitives: concurrently running processes, communication channels and synchronous communication over those channels. Process-oriented programs are typically *concurrent* and those that are concurrent are able to be *parallel* when executed on multi-core hardware with a language run-time supporting hardware parallelism.

2.2.1 Why occam-pi for Process-oriented Programming?

Use of the process-oriented paradigm does not imply the use of a specific programming language. The use of a language specifically created for its application, such as occam-pi, brings benefits in clarity of expression and execution performance. The work presented in this thesis uses occam-pi for building process-oriented systems, as it permits the most direct expression of process-oriented primitives; primitives for channel communication and parallel execution are first-class syntax elements. When teaching concurrency this direct expression of primitives allows students to focus on the properties of the programming model, abstracting their programs into concurrently executing, communicating components. Avoiding arcane library-based syntaxes or interactions between programming models when introducing concurrency provides a focus on the concepts, separating them from the implementation details and constraints of using message-passing concurrency as a library.

The Transterpreter virtual machine (TVM), an occam-pi run-time environment, is a significant factor in choosing the language for the application of process-oriented programming to robotics. The TVM, originally conceived for the purpose of allowing the application of occam-pi on a variety of platforms, is written in a highly portable subset of C and has been used to run occam-pi programs on systems from LEGO Mindstorms RCX (A H8 16MHz

CPU and 16K of RAM) all the way up to multi-processor desktop PC's (2.4GHz dual core and 4GB of RAM). The Transterpreter is discussed further in Section 2.2.5.

Erlang is a concurrent programming language designed by Ericsson, originally designed for building real-time control systems for telephone routing [AVWW93]. Erlang implements the Actor model of concurrency [HBS73] where actors (processes) communicate using messages but do not synchronise on the communication, unlike in CSP. Asynchrony of communication has implications for memory usage; messages between processes are placed in a mailbox which grows until the process reads the message. Erlang's runtime system is designed for larger embedded systems, requiring tens of megabytes of memory for a minimal instance — orders of magnitude larger than some of the target platforms for robotics work, as detailed in Section 3.1.

Google's Go programming language [Goo12] contains all of the necessary primitives for process-oriented programming, although as it is designed to support a number of models, its syntax is more complex and some process-oriented semantics (such as fork and join) must be manually specified.

Support for process-oriented primitives may be added to other programming languages via library, such libraries exist for half of the top ten most popular programming languages in use today [TIO12], including Java (JCSP [WB08]), Python (PyCSP [ABV07]) and C++ (C++CSP2 [Bro07]). However, as Boehm states in [Boe05], implementations of concurrency primitives as libraries offer significantly weaker guarantees of correctness. Indeed, in these languages processes must often be modelled as objects, leading to awkward syntaxes being needed and there are no checks on the usage of shared memory.

The occam-pi language has the ability to check parallel usage and prevent compilation of programs which use unsafe access patterns on shared memory, which would potentially introduce race hazards or data corruption. The channel communication model moves exclusive access to data between processes, resulting in data access patterns which protect data integrity (an issue discussed more fully in Section 5.2). The need for safe handling of shared data immediately arises in robotics from the need to share access to data from sensory input and the control of actuators. The use of process-oriented programming allows streams of input to be duplicated and output to be integrated and unified to the single streams expected at the hardware level. These techniques provide a foundation for building selection and behavioural control systems on top of, the subject of Section 3.3 of this thesis.

2.2.2 occam

occam was a parallel programming language developed by INMOS Ltd in symbiosis with the Transputer microprocessor [SGS95]. occam was designed to facilitate concise expressions of concurrent programs, with language primitives for parallel composition of processes, creation of channels and communication via message passing along channels. In the introduction to the initial specification of occam May states that it is "intended to be the smallest language which is adequate for its purpose" [May83]. Hence the name occam, from Occam's Razor, a principle of simplifying until further simplification sacrifices expression.

occam incorporates concepts from May's Experimental Programming Language [MTWS78] and Hoare's Communicating Sequential Processes [Hoa85]. occam differs from CSP in allowing a process to be composed from a sub-network of processes running in parallel. occam was designed to allow direct expression of both the concurrency of a component running on a single processor and the parallelism of a component running across a network of processors.

The Transputer microprocessor, for which occam was intended to be the programming language of choice, was designed to enable the construction of large parallel computers and features both high and low level elements in its design to support concurrency. At the low level, Transputers contain instructions to support channel communication (`in`, `out`) and managing parallel processes (`startp`, `endp`). At the high level, a set of four high speed serial links were included in the design, to allow Transputers to be networked to create much larger systems. occam programs could be written for and run on networks of Transputers, processes being distributed between Transputer cores and channels being multiplexed along serial links as specified by explicit placement instructions in the program code. Typical of occam's simplicity, syntax for the network distribution of processes is built directly into the language, the **PLACED** language keyword allowing the specification of where a particular process should be located.

As processes may be distributed across a network of Transputers, sharing data between processes is achieved via channel communications. All effects that processes in occam have are visible through their interface to the rest of the program; there is no global state beyond that of the hardware. This lack of global state means that occam-pi processes are compositional; processes encapsulate their state, have no side effects and behave as a 'black box' with a defined interface to the environment.

2.2.3 occam from Transputers to KRoC

Through a combination of commercial factors and the increasing speeds of traditional CPUs along with the failure of INMOS (then SGS Thompson) to ship improved Transputers, the architecture fell out of popular use in the early Nineties. The tight relationship between occam and the Transputer, which had led its uptake in the late Eighties made the continued use of occam difficult, as the available compilers were only able to generate programs in Transputer byte-code, to be run on Transputer systems. occam and its programming model was recognised as valuable by the Transputer community, and it was desired to be able to continue using occam to program parallel systems for commodity workstations.

The Southampton Portable occam Compiler (SPoC) was an early attempt at permitting the use of occam for writing programs on standard Unix workstations [DHWN94]. SPoC would translate a standard occam program into ANSI C code along with a set of functions and macros comprising a run-time system. A standard C compiler was then used to produce a binary for the target architecture. This approach meant that the generated C code was responsible for scheduling, there was no external scheduler. Each process offered a single method which would execute it, and these were executed in turn; the design of SPoC did not allow exploitation of hardware parallelism. Some optimisations were done at the occam source level, transforming the program before its translation into C, to ease the mapping between the two programming languages.

Subsequent to the development of SPoC, the occam For All (OFA) project was formed by the University of Kent and the University of Keele, motivated by a fear "that the engineering and commercial benefits previously enjoyed through the use of occam on transputer-based platforms will be denied to them in the future unless the language is 'opened' to the wider parallel computing community" [W+94]. The Kent Retargetable occam Compiler (KRoC) started life as a combination of deliverables from the OFA project; a native code translator for the Intel x86 architecture (`tranpc`) [Poo96], a run-time kernel (CCSP) to support concurrency in C and occam [Moo99], improvements to the INMOS occam compiler (`occ21`) and a wrapper script to drive the aforementioned components as a toolchain.

2.2.4 occam-pi

Welch and Barnes extended the occam language with concepts of mobility and network reconfiguration taken from Milner's pi-calculus to create the language known as occam-

pi [WB05]. The intention of occam-pi is to modernise the occam language, adapting it towards the modern environment of networked, powerful desktop machines with plentiful memory. occam-pi introduces more than just mobility and dynamism, including a number of features consistent with its goal of modernising occam for the desktop; a foreign function interface for calling C code, shared channels, mobile data and channel bundles. occam-pi as a language is a moving target, as additional language and run-time features are specified and implemented to support concurrency research at the University of Kent. A number of occam Enhancement Proposals (OEP's) have been created defining additional features that may be added to the language in future [Wel12].

2.2.5 The Transterpreter Virtual Machine

Motivated by a desire to use the occam-pi language for teaching on small robotics platforms, Jacobsen and Jadud encountered several impediments to its use in such an environment: the complexity of the KRoC backend, the need to port the CCSP runtime library to new architectures and the size of binaries generated by the toolchain [JJ04]. To enable these uses for occam-pi Jacobsen created the Transterpreter virtual machine, a *Trans*puter In*terpreter*, capable of running Transputer byte-code files generated by the existing KRoC `occ21` compiler. The Transterpreter is written in ANSI C so as to be highly portable [Jac06] and has been ported to many architectures: Intel x86, PowerPC, ARM, MIPS, Blackfin, Renesas H8/300 and more. This high degree of portability has been used to enable the use of occam-pi on a number of robotics platforms, among them the ActivMedia Pioneer 3-DX, LEGO Mindstorms RCX and Surveyor SRV-1. These platforms, their runtime ports and hardware interfaces are discussed further in Section 3.1.

2.3 Robotics

A robot is differentiated from a machine by autonomy, the ability to complete a task without operator control. A typical robot consists of a number of sensors, measuring properties of the world, some computational capability, and a number of effectors which can change properties of the world. Combining the processing of sensory inputs and control of effectors is called *robot control*. Robot control may be divided into three fundamental tasks: *sense*, *plan* and *act*. This is not to say that these elements are equally significant to a given robot control system; many modern approaches discard the notion of a planning stage, which attempts to build

a detailed world model for use in reasoning, in favour of more reactive or evolved systems using artificial intelligence to classify and react to the world.

2.3.1 Sense

Sensing provides data to the robot control program about the world. Robots do this via hardware sensors designed to detect and measure physical properties such as light, temperature or distance. Typically a robot will have sensors for a variety of physical properties, and may have different kinds of sensor for the same physical property to provide additional accuracy or redundancy. For example, a robot may be fitted with ultrasonic range finders around its circumference in addition to a high precision laser scanner, providing two different measures. There are cases where sensory input can be combined with particular mechanical actuation to form a local feedback loop, to better control or measure the actuation. For example, a stepper motor divides its rotation into equal steps and generates a tick signal on the completion of each step, the stream of ticks can be used to accurately determine how far the motor has rotated.

Handling this data input and making it available to the computation of the robot is a naturally concurrent task, each sensor represents an independent flow of input data into the program. Providing sensory data requires reading from the sensor hardware, capturing the data into a form usable by the rest of the program and updating it. Sensor hardware is varied, as are interface semantics; a particular sensor may provide regular inputs at a period, where another provides input only when the property it measures changes. Simply providing the values of the measured environmental properties as input to the program and keeping them updated in relation to the environment at a rate adequate for the program to behave properly can be a significant software engineering challenge.

More complex sensors, with local specialised processing capabilities, are becoming more common as the level of hardware sophistication rises in robotics. Sensors with independent processing effectively make robot platforms distributed systems, forcing consideration of the interaction between the two systems: synchronisation and communication.

2.3.2 Plan

Where used, the planner of a robot control system contains the computational logic which ensures that the robot can complete its intended task; the planning component of robot

control has been the subject of a wide number of approaches. Common to all approaches involving planning are the need to process the input data received from sensors and control the actuators to achieve the tasks desired of the robot. Planning may require *sensor fusion*, the collation and aggregation of sensor data to build a cohesive model of the environment. The robot may then use this model of the environment as a source of data from which the robot can evaluate the correct actions to take.

Using coarse grained concurrency mechanisms, such as threads or locks, to marshal sensor data within a program or its components immediately raises issues of synchronisation and introduces the potential for race conditions. Sensor reads from a planning component and updates, providing new sensory information, from the hardware interface will necessarily share data. These kinds of concurrency error are discussed further in Section 5.2.

2.3.3 Act

The physical hardware which allows a robot to produce a change in the environment is known as an effector. An effector can be as simple as an LED, or as complex as a robot arm. There are typically multiple effectors on a robot, a mobile robot will at minimum feature a number of motors or servos to create motion. As with sensors, different effectors provide different interfaces to the software. Some effectors may need to be part of a feedback loop to be controlled accurately; for example, a stepper motor provides a tick as input to the system for every unit of rotation it makes successfully, these ticks must be monitored to allow the program to control the amount of movement. Whilst interacting with the hardware to produce an accurate physical outcome from the software can present difficulties, resolving conflict between different demands in the control system of a robot is a far more significant issue in robotics. If two elements of the control system desire conflicting or opposite physical behaviour, the ability of the robot to complete either action depends on the ability to prioritise or co-operate between the demands on the physical hardware. A set of rules or principles for marshalling and co-ordinating behaviours is an essential part of a robotics architecture; the ability to clearly express concurrently executing components and reason about their interaction is a fundamental tool for composing these architectures.

2.3.4 Robotics Paradigms

There are two major paradigms in robotics:

- The Hierarchical paradigm, where sensing, planning and acting take place in a pipeline, as shown in Figure 2.1. Data about the environment from the sensors is passed into the planner, where a model of the environment is created, decisions are made based on this model and passed onto an actuation stage. The Hierarchical paradigm was first popularised by "Shakey the Robot" from SRI (then Stanford Robotics Institute) in the 1970s [Nil84].
- The Behavioural paradigm, shown in Figure 2.2, omits the explicit planning and building of a world model, using the world as its own model; the robot can be considered an agent program operating in the environment of the world. This is based on the concept that modelling the environment is a poor approximation of the actual environment, and that making best use, or responding to input more quickly is more useful. Planning is omitted in a behavioural control system, directly relating the behaviour of the robot to sensed conditions in the environment.

The two paradigms can be combined to produce hybrid Behavioural/Hierarchical systems, where a degree of planning is combined with a system which reacts based on sensory input. Behavioural systems are limited in that they can only act according to the stimuli they encounter, which can restrict the application for longer-term goal seeking and the system's memory of previous conditions. This hybrid approach can address these problems, adding a planning layer which can interact with the reactions to conditions to intervene as appropriate.

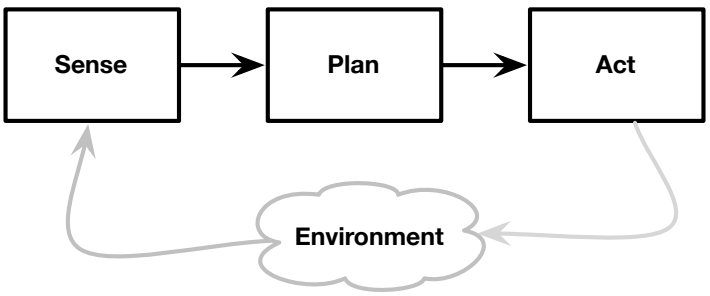


Figure 2.1: *The Hierarchical Control Paradigm, consisting of Sense, Plan, and Act primitives of robotic control*

The Hierarchical paradigm is rarely used in modern robotics as it depends on building accurate world models, enacting the generated plans exactly as specified and keeping the planner in sync with actual activity of the robot. These complexities delay reactions and produce brittle behaviour when faced with the environmental variability inherent to mobile robotics and the world as an environment. The Behavioural paradigm offers adaptability

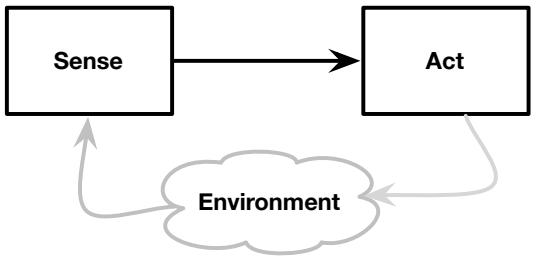


Figure 2.2: *The Behavioural Control Paradigm, consisting of Sense and Act primitives*

in changing environments, making use of environmental complexity in combination with simple behaviours to produce complex, emergent behaviour. This approach is influenced by biology and a more modern, cross-disciplinary approach to the design and control of robots [Win12]. A second important tenet of behavioural systems is the feedback loop between the environment and the action taken by the robot, shown in Figure 2.2; using the world as its own representation rather than building an outdated and potentially inaccurate model representation to plan with.

2.4 Robotics and Concurrency

Robot control is inherently concurrent; robots must interact with the world, where events are truly parallel and independent of each other. The relationship we have to the world as human beings lends a context in which to decompose problems of robot control; we experience the world through our senses simultaneously to processing and conscious or unconscious action. The use of a programming model in which tasks of sensory input, processing and actuation may be expressed as concurrent components, acting independently of each other and in synchronisation, yields a closeness of mapping between the problem decomposition and our human context.

The programmer is able to directly relate the streams of input and output to data paths through the control program. Expressing the concurrency of the inputs to and outputs from the robot directly in the structure of the program, even in the absence of concurrent physical hardware interactions, retains this mapping. Being able to express concurrency within the implementation language itself is a complement to the inherent concurrency of robot control problems. The data flow aspects of the process-oriented model fit naturally to the sense, plan and act primitives of robotic control.

```
void stop_on_light(){
    if (read_light_sensor(PORT_1) > 50){
        motor_speed(PORT_A, 0);
        motor_speed(PORT_C, 0);
    }
}

void beep_on_touch(){
    if (read_touch_sensor(PORT_2) == 1) {
        play_sound(BEEP);
    }
}

int main(){
    while(1){
        // Light sensor task
        stop_on_light();
        // Touch sensor task
        beep_on_touch();
    }
    return 0;
}
```

Listing 2.1: *An imperative robot program with two interleaved tasks, written in C*

A sequential robot program written in C for a hypothetical robot similar to the Mindstorms RCX is presented in Listing 2.1. This program has two independent tasks to achieve: if the light sensor reads a value over 50 the robot should stop its motors, and if the touch sensor is pressed, the robot should beep. The tasks are broken into functions, as would be typical of program structuring techniques in C. In the C implementation, the two tasks must be interleaved explicitly into a single thread of execution; the program first runs the light sensor task then runs the touch sensor task before looping. If the reading of the light sensor is a long task, or the `motor_stop` operations only complete when the motor has come to a complete stop, the touch behaviour will be affected; the entire program is one long sequence, even where functions are used to break up logically different tasks.

A concurrent, process-oriented implementation in occam-pi of the hypothetical program is shown in Listing 2.2. There are similarities between the two; both programs subdivide the two responsibilities of the programs into a functional unit each, using functions in the C version and processes in the occam-pi version. However, in the process-oriented model, the configuration and interface to hardware is drawn out explicitly as a top level component rather

than being distributed as parameters to various library calls throughout the program. The process-oriented version allows for the inherent concurrency of the program to be expressed; the occam-pi PAR keyword is used, meaning the two tasks and all hardware interface processes run concurrently with each other, each process having an individual thread of control. Given this explicit expression of intent, the occam-pi runtime environment handles the scheduling of and context switching between the processes; given parallel hardware, the two separate tasks operate concurrently without any additional work to the programmer. This pushes the need for reasoning about the interleaving of multiple tasks outside the robot program and the mental model of the programmer, unlike the C implementation which requires explicit interleaving of the two behaviours.

A process network diagram of the concurrent robot program is shown in Figure 2.3, showing the data-flow through the program and the construction of the hardware interface as part of the program. The occam-pi version of the program encapsulates the hardware interactions, reading from sensors and controlling motors, into processes at the edges of the network. This encapsulation means that the task processes themselves are abstracted from hardware interaction, handling only a single input value at a time and generating commands.

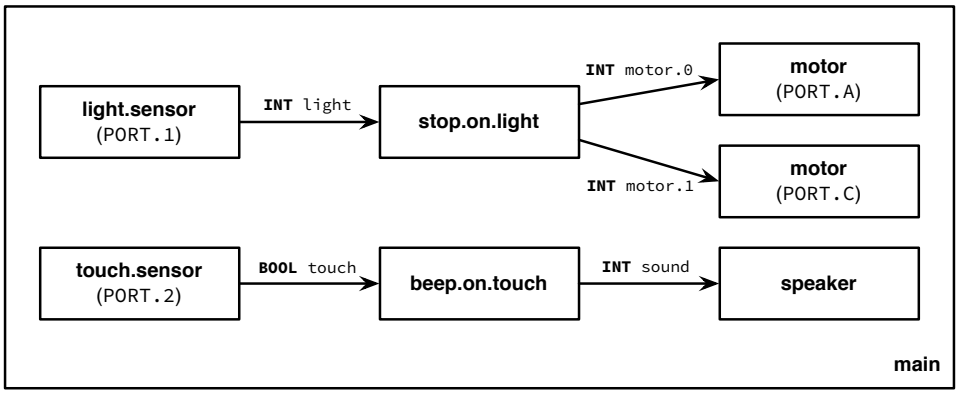


Figure 2.3: *Process network diagram for the process-oriented robot program shown in Listing 2.2*

The inherent concurrency of the process-oriented model, of processes being able to run independently of and in parallel to each other, makes it particularly suitable for expressing robot control logic. In the process-oriented programming model the flow of robotic control, from sensory input data to control of mechanical actuation, can be reflected directly in the composition of the program. Decomposing the program in this way, to a network of communicating processes which reflect the data flow of the program reduces the level of abstraction and improves the closeness of mapping between the problem and its solution.

```
PROC stop.on.light (CHAN INT light?, CHAN INT motor.0!, motor.1!)
  INT value:
  WHILE TRUE
    SEQ
      light ? value
      IF
        value > 50
          PAR
            motor.0 ! 0
            motor.1 ! 0
        TRUE
          SKIP
:

PROC beep.on.touch (CHAN BOOL touch?, CHAN INT sound!)
  BOOL touched:
  WHILE TRUE
    SEQ
      touch ? touched
      IF
        touched
          sound ! SOUND.BEEP
        TRUE
          SKIP
:

PROC main ()
  CHAN INT light, motor.0, motor.1, sound:
  CHAN BOOL touch:
  PAR
    -- Hardware configuration
    light.sensor(PORT.1, light!)
    touch.sensor(PORT.2, touch!)
    motor(PORT.A, motor.0?)
    motor(PORT.C, motor.1?)
    speaker(sound?)
    -- Tasks
    stop.on.light(light?, motor.0!, motor.1!)
    beep.on.touch(touch?, sound!)
:
```

Listing 2.2: *A process-oriented robot program with two tasks written in occam-pi*

In the process-oriented model, processes run in isolation of each other apart from communicating using the interfaces defined by their channel connections. These processes are compositional, meaning they may be run in parallel with each other and combined without side effects, communicating via message passing over channels. The independence of processes allows separation of concerns, with good practice in process oriented programming encouraging the creation of small, single purpose processes which are composed to produce higher level functionality. This composition facilitates the creation of larger and more complex systems by combining existing well specified and understood components, allowing the programmer to focus on achieving the desired functionality and behaviour of the system through composition. A process is essentially a 'black box' to the rest of the program, with no global state and only its channel interface available to interact with it. When composing processes, an outer process can be wrapped around the network, making the entire composition invisible to the outer program.

There is a symmetry between the physical makeup of the connection, by trace or wire, between discrete electronic components to form a robot's physical systems and the connection via channels of discrete software components to form a control system for that hardware. In reconfigurable robotics systems, the LEGO Mindstorms RCX and NXT being examples, the modular re-connectivity of the hardware can be mirrored in the software components used to construct programs. This symmetry provides a closeness of mapping; bringing the two processes closer conceptually reduces the cognitive difficulty in constructing a software configuration for a given hardware configuration. This effect applies bidirectionally, changing the software can drive changes to the hardware configuration where reconfiguration is possible.

Embedded systems found on small robotics platforms are typically memory constrained environments. Concurrency models such as Erlang's actor model, whilst having the benefits of separation between processes and a communication model raises the demand for memory due to the asynchronous nature of communications. In asynchronous communication, messages are buffered and stored whilst the process they are intended for is ready to receive them [AVWW93]. This buffering leads to increased memory usage and introduces nondeterminism to the run-time memory usage of the program; at best these buffers may be capped and messages discarded.

occam has been previously used for robotics, a result of the popularity of the Transputer for use in control applications during the mid to late 1980's [NWN88, JE88, KS84]. Early explorations are typical of occam and Transputer use in robotics, where the parallel software

and hardware provided a means to speed up complex calculations such robot dynamics and kinematics [HPR89]. Jacobsen et al. reintroduced the ability to use occam for robotics in the absence of Transputer hardware through the creation of the Transterpreter virtual machine runtime [JJ05].

2.4.1 A Demonstration of Process-oriented Robot Control

A robot competition run at the *AAAI Spring Consortium Symposium on Robots and Robot Venues: Resources for AI Education* provides a case in point example of the application of concurrency to robot control. The competition task was set out as follows: the robot should start in a corner and go as far as possible in the first 30 seconds, then return as close to home as possible in the next 30 seconds; each team had no more than 24 hours experience of their robot platform and an hour with the task definition to write the control problem. The author, Jacobsen and Jadud took part in this contest with the Surveyor SRV-1 robot (a platform discussed in detail in Section 3.1.6). A rudimentary port of the Transterpreter to the SRV-1 was completed in a few hours prior to the competition, providing a run-time environment on the robot for occam-pi programs. During the contest, in an hour, a subsumption architecture was designed and implemented on the robot to complete the challenge tasks, starting with a design session working entirely with a diagram to establish an architecture for the solution. The original chalkboard diagram in shown in Figure 2.4 and a redrawn figure of the diagram is shown in Figure 2.5. This program won the small contest, successfully traveling a distance of 39 robot diameters in 60 seconds. The methodology employed in going from problem statement to functioning robot control program exploited the visual representations and component isolation available due to the use of a process-oriented environment to program the robot.

A process network diagram aids in high-level reasoning about design and architecture of the program, decomposing the problem into a number of processes and deriving the communication relationships between them. Use of a diagram facilitated discussion and sharing of design ideas before a single line of code was written; the program's design was iterated several times before any code was written at all (as can be faintly observed in the erasures on the chalkboard in Figure 2.4.)

Clear understanding of the relationships between components is important in subsumption architectures, as components are able to interfere with the behaviour of each other to affect the behaviour of the robot. Section 3.3.1 explains subsumption architectures in detail and the

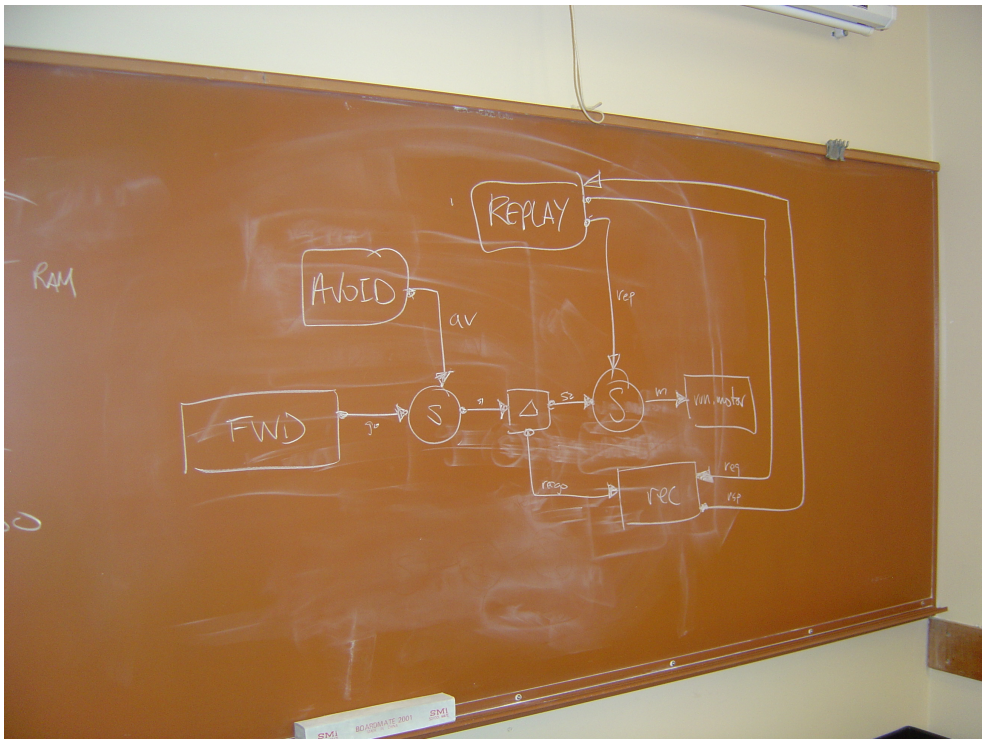


Figure 2.4: *Chalkboard design of a process-oriented subsumptive robot control system which negotiates an environment and follows the same path home*

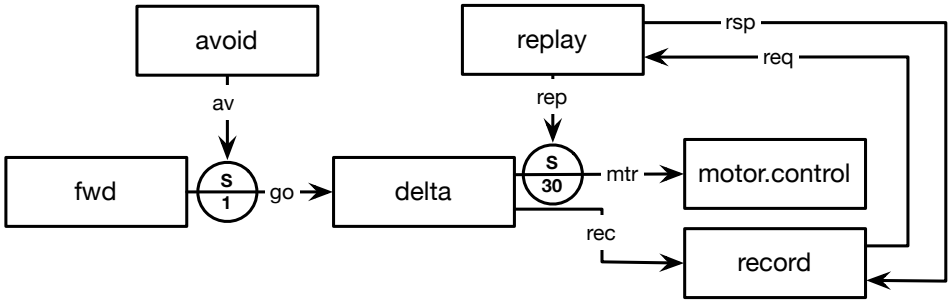


Figure 2.5: *Process-network diagram for a subsumptive robot control system to negotiate an environment and follow the same path home, as sketched in Figure 2.4*

application of this model using process-oriented concurrency.

Once the components and design of the program had been formalised, the design could be implemented as a composition of (as yet) unimplemented processes based on the diagram. Implementing the program fully is then a process of implementing each isolated process according to its interface to the rest of the program, rather than trying to structure the program on an ad-hoc basis by starting with the code.

2.5 Robotics in Computer Science Education

There are a number of problems and limitations inherent to the use of robots, and to a lesser extent robotics in Computer Science education. Use of physical robots in requires space and access; where for a typical programming assignment students may only need to sit at a computer, a robot will need both space to be programmed and an environment in which to be run. There are added costs; not only does a student need a computer (and potentially specialised software to program the robot), they also need access to a robot platform and any interfacing hardware to allow programs to be loaded.

Advocacy for robots as a teaching tool for Computer Science can be traced back to tools such as Pattis' Karel the Robot [Pat81]. Karel the Robot consists of a simulated world visualised as a grid of cells, representing avenues running east–west and north–south, a set of walls and a number of beepers contained inside the grid. Karel is capable of moving, turning, placing and picking up beepers, and switching itself off; the simple programming language used to control Karel contains a corresponding command for each ability.

There have been a number of environments designed to support robotics use in CS Education, a major advance being Pyro (*Py*thon *ro*botics). Pyro is a Python robotics framework which seeks to abstract across multiple robotics platforms and allow teaching of robotics at the undergraduate level, progressing from simple reactive control systems (those that directly connect input stimuli to output actuation) through to advanced robot architectures in a single environment [BKMY03]. Abstractions across robot hardware allow for different kinds of robots to be used as the control problems presented change throughout the course. Python supports a number of different programming methodologies, lending further flexibility to Pyro. Pyro, subsequently developed as Myro has become part of the Calico Project, a wider language agnostic and multi-context framework and environment for teaching Computer Science [BKM+12]. Robotics has also been used in Computer Science education as an application

area to increase student engagement, through the use of personal robots [KBB+08]. These small robotics platforms allow for a constructivist, hands on approach involving physical, tangible objects.

The first Computer Science course at MIT, 6.001, based on Scheme and the famed Structure and Interpretation of Computer Programs (SICP) text has been replaced by 6.01, a course taught in Python and focused on robotics, due to the engineering challenges inherent to them. The motivation behind MIT's move to robotics in the first course is captured by Martin's "Real Robots Don't Drive Straight" [Mar07], that the significance of feedback in a system to adapt to changing conditions or imperfection in actions is an essential part of the learning experience; Martin identifies the damaging effect of these low-level feedback loops being presented as closed loop higher-level primitives.

The use of small robots in the classroom as experienced by the author is fundamentally driven by Papert's theory of constructionism, that learning can happen best in a context which has hands on, tangible elements [Pap86]. Papert expanded on and demonstrated these concepts in a practical context in the book MindStorms with the LOGO Turtle, a computer controlled device which introductory students could use to draw pictures by following a series of commands [Pap80]. A LOGO Turtle may be either virtual, existing in simulation and drawn on screen, or physical, using motion and a physical pen which may be raised or lowered to produce a line on paper.

A collaboration between Papert's research group, the MIT Media Lab, and the LEGO Group resulted in a prototype Programmable Brick [RMSS96], a small embedded computer to be used in conjunction with LEGO, allowing children to design programs that interacted with the physical world. The Programmable Brick was designed to maximise its input/output capabilities to provide the most possible applications of the brick, able to read from six sensors and control four motors or lights simultaneously. Programs were loaded to the Brick via a cable connection to a computer, where programs were written in a dialect of LOGO. Visual environments were developed for this LOGO dialect, a number of which are discussed in Section 4.2.1.

The LEGO Group subsequently commercialised the Programmable Brick as the LEGO Mindstorms RCX. The Mindstorms RCX has been used widely in Computer Science Education, in a variety of contexts and in conjunction with a wide range of different programming paradigms [Bar02, Fag03, JCS03].

These efforts are based in the general principle that Robotics, specifically via small robot

platforms is an engaging area in which to motivate introductory programming.

2.6 Pedagogy of Process-oriented Robotics

occam, and subsequently occam-pi have been taught as part of the undergraduate Computer Science degree program at the University of Kent for the last 25 years [Wel99]. A set of standard exercises are used which place emphasis on the compositional aspects of the process-oriented paradigm. A toolbox of elementary processes are used throughout the course, known as the "Legoland" component set. Diagrams are the first representation of process-oriented parallel programs the students experience and work with in the course, as an exercise prompts them to write a parallel composition statement for a network diagram containing a number of processes and channels. Starting with exercises involving composition of parallel processes and channel connectivity emphasises the most significant differences between process-oriented programming and the object-oriented paradigm students are used to.

Efforts have been made by the author of this thesis and others at the University of Kent to introduce robotics to this module as an area in which to motivate introductory process-oriented programming. These efforts have focused on making available environments which present authentic challenges from robotics: the concurrency between sensory input and actuation, task selection along with mediating control of the hardware and the error and imprecision inherent to real-world robotic systems, sensing and actuation. The following subsections detail these efforts and their input to the pedagogy surrounding concurrency at the University of Kent and other academic institutions.

2.6.1 RoboDeb and Player/Stage

RoboDeb was a prepared virtual machine image for the free VMWare Player software that allowed students to virtualise a Debian Linux system with the tools required for developing robotics control software in occam-pi and simulating robots installed and pre-configured ready for use [JJ07]. The RoboDeb environment employed Player/Stage, a general purpose robot control and simulation toolkit; Player runs as a device server, offering control over hardware and Stage offers a simulation environment for Player devices [GVH03]. RoboDeb also included a library for communicating with Player from occam-pi, providing process-oriented abstractions over the underlying serial calls. Programs written for the Player device

server can be run both in simulation via Stage and on physical robotics hardware. In the case of RoboDeb's use at the University of Kent, this physical robotics platform is a single (due to its relative expensive and size) ActivRobots Pioneer 3-DX [Mob] robot.

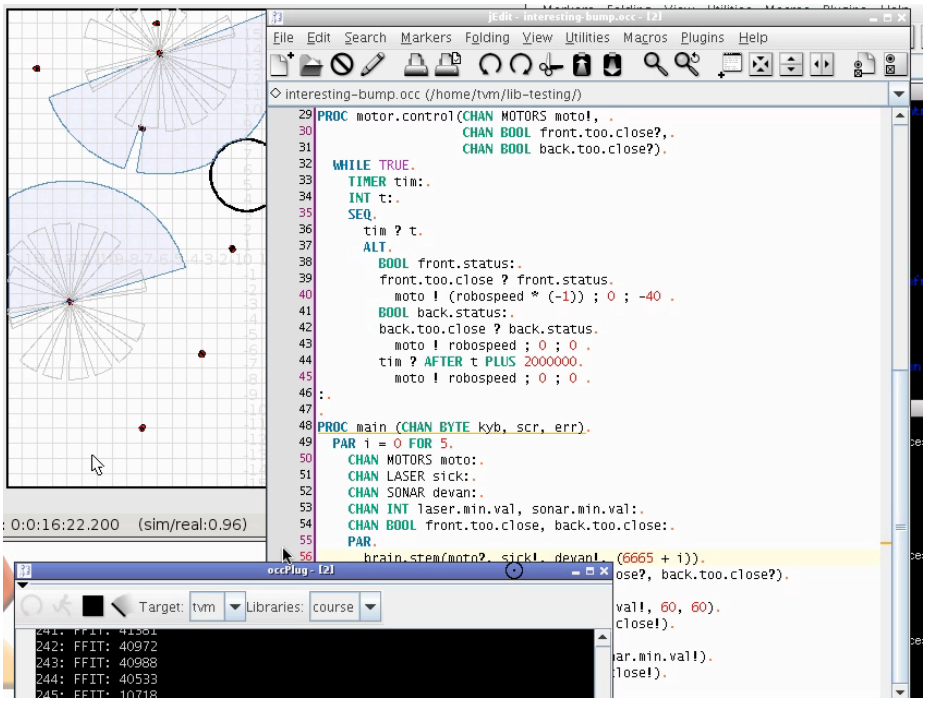


Figure 2.6: *RoboDeb in action, a typical RoboDeb session with Player simulator and jEdit occam-pi editing environment, from [JJ07].*

Establishing a direct link between the simulated environment and a physical robot by running student programs, designed for simulation, on the physical platform demonstrates the noise and inaccuracy issues of the real world and hardware versus purely logical, software problems. While the use of simulation disagrees with the constructivist agenda of hands-on making, the convenience to students of having the option of wider access to a simulated robot and environment at any time to work on their code was of benefit.

The author has previously used the RoboDeb environment to deliver a workshop entitled "occam-pi robotics: Complex Behaviours from Simple Systems" to Year 13 (17–18 year old) students as part of their A-level qualification at a local school. This workshop involved the students starting with material involving sequential programming, moving on to parallel process composition and finally building a working program in the RoboDeb environment, starting with a half-complete template and using process composition to add components to form a working solution. This session included a whiteboard exercise, asking the students to direct the connection and creation of components, reasoning about the solution to a problem

graphically prior to their solving a similar exercise in code. This graphical precursor to the practical, symbolic coding task gave the students existing experience of high-level process composition that allowed them to solve the task much more easily than expected.

2.6.2 Cylons

Cylons was an assessment designed by Peter Welch in 2007, based on an agent visualisation by David Wood. Students were to design a robot control program which could navigate a robot (represented by a circle) around a maze without becoming stuck. By default the robot control program written would be spawned hundreds of times, controlling many robots concurrently. It aimed to replicate some of the challenges of RoboDeb, but without the need to run an entire virtualised system. The programs were intended entirely to be run in a small simulated environment, which had more in common with an agent simulation than a robot simulator; actuator control and sensing were both without any degree of error. A key outcome of the task was for students to obtain emergent behaviours, such as swarming and queueing, from interactions between the robot, other robots and their environment.

2.6.3 Life on Mars

Life on Mars was an assignment set as part of the undergraduate concurrency course at the University of Kent and designed to be completed by students at the end of the course. The assignment has run four times since being designed by the author of this thesis in collaboration with Ritson in 2009. Students are presented with a 'martian' world, shown in Figure 2.7, featuring a sandy surface, a rocky outcrop (indicated by the lighter orange areas), a number of coloured beacons and a red dot indicating the position of their robot. The simulated robot is equipped with a front facing blob-finder and a number of ultrasonic hazard sensors placed around its circumference; the field of vision for the camera is indicated by a cone of red lines and the hazard sensors are indicated by a series of yellow lines projecting from the robot.

The robot is fitted with two motors powering a wheel on each side and a third castor wheel, allowing it to be rotated on the spot and moved forward or backward. A command/response interface is provided between the robot and the terminal the program is run from, allowing the robot to accept user commands and output responses according to a defined occam-pi protocol. An shared output channel on which BYTEs may be sent is provided as a convenience,

allowing students to output textual state from their robot programs to the terminal for debugging. This channel of debug output often serves as a lesson in the difficulties of using 'printf' style debugging with concurrent programs for the students, as use of this channel throughout the program leads to contention issues between components. These effects and their significance to students learning how to reason about the behaviour of their programs are discussed more fully in Section 5.2.4.

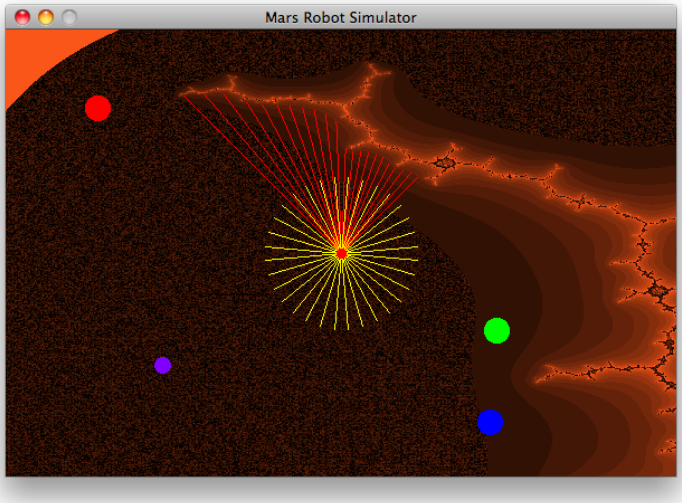


Figure 2.7: *A screenshot of the Mars Simulator from the Life on Mars assignment*

The software interface presented to this simulated robot is based that of a real robot, the ActivMedia Pioneer 3-DX, with which both authors of the assignment have practical experience. The interface is conceived in such a way that it is possible for programs designed by students to be run on the real robot in a physical environment outside of the simulator. The assignment aims to provide a realistic experience of programming for a robot control problem in occam-pi whilst pragmatically avoiding the need for students to have continued access to physical hardware and a lab environment to complete the assignment. The use of the virtual machine-based RoboDeb environment with Player/Stage by Jacobsen and Jadud [JJ07] notionally removed the requirement for hardware and a lab environment, but in implementation actually required the use of an additional laptop pre-configured for using the virtual machine for a number of students. Life on Mars has the advantage of being a standard occam-pi program, with no external dependencies on robotics hardware or simulation environments. This absence of dependencies means that Life on Mars is capable of running on any platform where students have used the occam-pi language successfully for

earlier exercises.

To complete this assignment, students are asked to write a control program for the simulated 'mars rover' from a blank process body, achieving a growing level of competency over the completion of four assessed tasks. The four tasks require an increasing amount of co-ordination between sensing and locomotion: moving around safely, locating coloured markers, and finally delivering a package to a particular location indicated by two markers. The first task is to get the robot to respond to commands to turn from the operator and report back the exact amount of turn the robot completed; the design of the robot interface is such that they must request a turn in the correct direction then sum ticks back from the motors to determine how far has been moved, to account for any slip that may occur. Even this most basic task requires co-ordination between two sources of input (operator command requests and motor ticks) and output (operator command responses and motor control commands). Students start with this simple movement (allowing operator commands to be translated into rotation and forward/backward motion), adding hazard detection from an array of ultrasonic sensors, a simulated vision problem and then a combination of all of the above behaviours into a routine which can deliver a package at an operator specified location.

Life on Mars' assignment support code presents a series of well defined process interfaces to the simulated robot hardware, and these were designed such that a library could be made to run these programs on a real robot platform. Some of these interfaces are more complex than a simple synchronous channel - channel bundles are present which require a client/server approach and the use of protocols to handle communication. For example, an operator channel bundle has both an `operator.request?` channel end, allowing messages to be received from the virtual operator, and an `operator.response!` channel end, allowing messages to be sent back to the operator console. The assignment is structured such that students have to get to grips with protocols immediately. At this point in the course students have completed an assignment requiring them to design and use protocols of their own definition in refactoring a simulation of the Dining Philosophers problem, as originally formulated by Djikstra [Dij87]. This progression is designed such that students are comfortable working with channels that carry variants with different parameters rather than single values or arrays of basic types. Fully solving this assignment means the students have to deal with four sources of input in parallel, and controlling both motors and a gripper actuator in response to the input — a challenging task.

2.7 Wider Pedagogy of Concurrent Systems

There is an increasing need for programmers who are able to make effective use of concurrency in a multi-core and many-core future of computing. Whilst considering the pedagogy of process-oriented programming, it is also important to include the wider pedagogy of concurrent systems. Teaching students to identify, avoid and solve common concurrency errors, such as race hazards and deadlocks has traditionally been difficult; a number of tools have been developed in different concurrency models and using different approaches to give programmers an understanding of the execution of their program. There are a number of tools that aim to simulate and visualise the concurrency within a program, whether by creating a visual representation of the concurrency primitive itself (thread, lock) or relating execution state of the system to the program. This section does not aim to be a comprehensive review of available tools and methods, rather identifying a representative set of approaches taken in the literature.

2.7.1 BACI

BACI, the Ben-Ari Concurrent Interpreter is a concurrency simulator originally designed by Ben-Ari [BAK99]. BACI consists of a compiler and interpreter combination; the compiler generates code which can be run in the interpreter, the interpreter allows the visualisation of machine state as the program runs. This visualisation and externalisation of execution state allows programmers to reason about and verify the behaviour of their code while it is being executed.

2.7.2 SPIN and FDR Model Checkers

The SPIN model checker, designed by Holzmann for verifying communications networks, is designed to allow the verification of interactions between asynchronous processes [Hol97]. Ben-Ari created a pedagogic IDE around SPIN, jSpin to simplify interacting with the Spin tool, along with a tool called SpinSpider, to draw state transition diagrams of the SPIN output [BA07].

The modelling language used by SPIN, PROMELA (*Pro*cess or *Pro*tocol *Me*ta *La*nguage), allows the definition of concurrently executing processes communicating synchronously over channels, meaning it would be possible to apply SPIN for verifying process-oriented

programs. A similar approach would be possible via process-oriented paradigm's relationship to CSP, whereby CSP models of programs can be written for process-oriented programs. The FDR [Foroo] model checker by Formal Systems is able to verify a wide range of assertions on the properties of CSP models, including freedom of the model from concurrency errors such as livelock and deadlock (discussed further in Chapter 5).

These approaches rely on using models as a target language, a layer removed from solving practical problems and analysing their use of concurrency — helpful to illustrate and explore concurrency issues, but not tied directly to the implementation of the program. Pedagogically, to use a model checker of this kind requires teaching the modelling language on top of the programming language in use for concurrent systems, doubling the burden and requiring students to map the logic and structure of their program into a program model in a completely different language before being able to check its properties. Errors made during this mapping can remove the benefit of model checking, the results gained from a model checker are only as good as the accuracy of the model in capturing the semantics of the program.

Maintaining a model of a program in parallel with its implementation is problematic, as the model and program are separate entities — errors can be introduced translating between the two, updating the program code to reflect a changed model, or updating the model to reflect a changed program.

2.7.3 Elucidate

Exton's Elucidate is another trace visualisation tool, which aims to avoid requiring modification of the users program to allow the visualisation [Extoo]. It makes use of debugging support built into the Java VM, running the user program in a second VM alongside the one running the Elucidate environment. The use of a virtual machine with support for tracing provides logging that is transparent to the user program. Not having to modify the code being debugged to receive traces of its behaviour is a significant advantage, as modification of the code gives rise to the possibility of errors being introduced and with concurrent programs, the potential for debugging code to change the temporal aspects of the program (the effects of modifying programs and temporal issues in concurrent debugging are discussed further in Sections 5.2 and 5.2.4).

2.7.4 ThreadMentor

ThreadMentor is a pedagogic concurrency visualisation tool designed to let students observe the actions of concurrent primitives within their programs [BCHS00, CMS03]. It consists of a class library, designed to resemble the standard API's for thread management and concurrency primitives, which wraps around the underlying calls. This library communicates with a visualisation system, updating the state of the primitives such that the user can see the parallel state in their program. There are disadvantages to this approach — as the visualisation is a separate component there can be a delay between the observed activity of the system and its visualised state.

2.7.5 ParaGraph

Heath and Etheridge's *ParaGraph* aims to make use of the large quantities of information generated when monitoring the behaviour of parallel programs, offering 25 different perspective views over input traces [HE91]. ParaGraph is designed to monitor message passing concurrent programs, and while not educational in nature, is motivated by aiding the user's understanding of their program's execution for performance gain. It is interesting to consider ParaGraph, as its approach of providing a very large range of visualisations of concurrent program behaviour, from space-time diagrams to hypercubes and gantt charts pushes the choice of appropriate visualisation to the end user. While Heath et al. set out to make ParaGraph easy to use, an understanding of the visualisation types, their appropriate applications, uses and comprehension of how the measured property can be affected by changes to the program are all required for its effective application.

2.7.6 PARADE, POLKA and XTANGO

The PARallel program Animation Development Environment (PARADE) system created by Stasko et al [SK93, Sta95] is another visualization system designed to allow users to observe the behaviour of their algorithms, although instead of visualizing concurrent primitives it allows the creation of customisable visualizations appropriate to the individual programs. The POLKA animation system, a component part of PARADE, is designed to allow the animation of algorithms without requiring any low-level graphics code to be written. A direct antecedent, XTANGO, has been used successfully in this role in the classroom for

visualising operating systems [Har94]. POLKA provides high-level shape drawing primitives to construct visualisations and a C library based interface which facilitates parallel updates of the diagram in sync with the program. With a system of this kind, the end-user would insert calls to the animation library to create and update the state of the visualisation in step with program state. The wider PARADE system includes `gthreads`, a set of macros designed to monitor the activity of the standard `pthreads` threading library, allowing for the automated generation of tracing code for programs built to use this library with minimal modification. However, all of the intended uses of these tools expect that there will be modification to the underlying program to provide traces for later visualisation.

Chapter 3

Process-oriented Robotics

Robot Control is a mixture of both engineering and artificial intelligence, presenting a unique set of challenges to the programmer. Robots sense and interact with their environment, transforming the input from sensors into output actuation and producing behaviour. A robot will typically have multiple sensors and multiple effectors, all of which must be controlled simultaneously. This inherent concurrency means that use of multiple control loops or complex interleaving of tasks is required even before considering the computation involved in decision making.

When writing programs for robots that run in the real world, as opposed to simulation, it is essential to consider control as a real-time problem. When writing real-time programs, run-time performance becomes clear — the delay between a robot sensing a condition and acting upon it is visible as hesitation in the robot's reaction to stimuli. The performance differences between desktop PC hardware and embedded platforms typically found in small robot platforms exacerbate this effect; on a desktop PC a badly written program may simply take longer to complete whereas on a robot badly written robot program may perform its task unsafely or completely fail to do so.

The correct use of interleaving or concurrency in a robot control program is critical in ensuring the expected behaviour of a robot. Even simple robots have multiple tasks to achieve simultaneously. If a robot is programmed to avoid walls whilst carrying out other tasks, the control program must co-operate between the tasks and the sensing, computation and actuation of the task which allows it to avoid walls. This need to handle concurrent tasks in the problem domain motivates the application of a concurrent programming model; the ability to consider a robot control problem in terms of data-flow further motivates a

process-oriented and synchronous message-passing based approach. There are a number of research areas covered in this chapter:

- Making process-oriented programming available on robot platforms suitable for the classroom, designing programming interfaces which make use of the robot's hardware consistent with the programming model.
- Combining existing behavioural robotics architectures and the process-oriented model, to establish the applicability of the model to established methodology and implications for structuring process-oriented robot programs beyond ad-hoc process networks.
- Applying existing process-oriented design, taught as part of an existing concurrency course, to robotics.
- Presentation of a case study process-oriented robot program against an equivalent C-based robot program to establish the viability of and identify properties of the process-oriented model in this context.

Developments in each of these areas have informed each other, the end result being a body of knowledge and established evidence of the viability and applicability of the principles established. This chapter lies at intersection of robot control and the process-oriented programming model, providing an answer to an overarching question; how can the process-oriented programming model be effectively applied to the problem of robot control? The case study which concludes this chapter both establishes the viability of a process-oriented approach using occam-pi for robotics and identifies a number of advantages this approach brings over an existing approach.

3.1 Process-oriented Programming on Robot Platforms

It is the author's belief that robot programming is most compelling when the robot is able to act independently of, and untethered from, the computer used to program it. Real world robot applications often require this ability, and in the classroom there are specific pedagogic benefits to presenting constructivist 'hands on' elements along with programming tasks (detailed in Section 2.5). This is not to say that tele-operation is without benefits – desktop toolchains may be used relatively unmodified (save the development of any necessary interface or communication libraries) and there are no issues with building program loaders or

custom firmwares. A number of robot ports, their operating software and hardware interfaces are discussed below, along with a single robot operated by desktop interface (the LynxMotion AH3-R Hexapod, as further discussed in Section 3.1.7).

3.1.1 Run-time Support

While process-oriented programs may be written in many programming languages, via library or otherwise, occam-pi provides both the clearest expression and the availability of a run-time environment that makes porting straightforward. The reasons for choosing occam-pi as a process-oriented implementation language are discussed in Section 2.2.1.

The Transterpreter (introduced in Section 2.2.5) has a rich history of being ported to new architectures and platforms, as an intellectual exercise [Jac06] demonstration of its design strength for portability and to facilitate research explorations using the process-oriented programming model [DJJ06]. This history has carried over into facilitating the use of robot platforms for process-oriented programming. As described in Section 1.3, a number of researchers at other institutions have used these ports for principles and design inspiration when designing their own process interfaces for robotics. The single threaded nature of the Transterpreter has not been a limitation to this work, as none of the robots used have hardware parallelism. Use of a multi or many-core embedded board for process-oriented robot control would be a fertile area of future work and is discussed in Chapter 6.

3.1.2 Process-oriented Hardware Interfaces

The design choices made when providing an interface to the robot hardware to programmers influence the design choices they will make when writing their programs. When seeking to provide an environment in which the process-oriented model may be taught and motivated, the provision of sequential, imperative interfaces to the underlying hardware is undesirable. The first steps for a programmer in writing a process-oriented robot program should not involve encapsulating calls to and from hardware functions to allow use of message passing and concurrency in the program, this should be a feature of the hardware interface provided. Presenting the hardware via a process-oriented interface to the programmer provides an appropriate starting point from which their programs may build, with sensor data provided as messages into the program and control of effectors facilitated via messages out of it. The author and a number of collaborators have worked on facilitating the use of process-oriented

programming on robot platforms, via additional runtime support and the creation of such concurrent hardware interfaces.

As discussed in Section 2.2.2, the origins of occam-pi mean it has built-in language features designed for low-level interfacing to hardware, placing variables at specific memory addresses and so on. These language features can be used to build hardware interfaces that avoid calling to libraries written in other programming languages and using underlying abstractions. providing a interface based around processes and communication from the hardware up.

Building interfaces comprised entirely of occam-pi processes executing within the virtual machine context allows reasoning about the behaviour of the hardware from the process-oriented run-time environment. When using external libraries, providing a process abstraction over underlying calls can introduce undesirable and inefficient polling to maintain the synchronisation between the underlying activity and the process' behaviour, interfering with the virtual machine's scheduling and execution performance. Interactions between the virtual machine's mechanisms for calling out to external functions and libraries and side effects in the external code can also produce unpredictable behaviour.

However interfacing completely in occam-pi is not always possible or practical. It is often desirable to re-use existing hardware interfaces and existing platform code, especially when bootstrapping a port to run the virtual machine on a new platform or where the underlying libraries provide acceptable performance and characteristics for the desired application. The following section addresses interfacing to external libraries and its existing applications to process-oriented robotics.

3.1.3 Calling into Existing Libraries

Having to wrap a process-oriented interface on top of high level abstractions can cause problems; as the run-time behaviour of a call made from the process-oriented environment into the underlying library is not entirely contained within the process-oriented run time the programmer can reason about. For expedience, the first port of call in making use of a robot platform, after providing a functioning run-time port of a process-oriented language, is often wrapping the existing, proven, libraries for controlling it.

The KRoC toolchain provides a Foreign Function Interface (FFI) for calling into C functions from occam, which appears as a function call in the occam source. This mechanism originally only supported non-blocking calls [Woo98] and support was added later for allowing pro-

cesses to make blocking calls (such as those to file or network operations) [Baroo]. These FFI calls require wrapper code to be written for each target C function that it is desirable to make available from occam. Efforts have been made to harness automated tools for analysing C libraries and generating these wrappers automatically; Dimmich added support for occam-pi to SWIG (the Simplified Wrapper Interface Generator), a general purpose piece of software for creating bindings between libraries and scripting language. [DJo5]. However, use of SWIG require extensive setup and configuration and it is often more efficient for a small number of functions for the wrapper processes and functions to be written by hand.

An example of using the FFI to access a C function wrapped via SWIG is shown in Figure 3.1, showing the layers of processes and functions in both the occam-pi and C runtimes. Two of the wrappers are auto-generated by SWIG: a C wrapper prefixed by an underscore which can accept arguments in the occam-pi style and an occam-pi function which accepts the arguments required by the underlying C function. The mapping from C brings with it the expectation of shared state, in the example wrapping there is a `self` structure passed into each function in C which is wrapped into the occam-pi function — a clean process-oriented implementation requires isolating this state into a process and creating a communication based interface, as seen in the figure. The depth of the stack and wrappings also highlights another issue, the unpredictability and effects of calling into a secondary runtime environment and functions in another language from occam-pi. These wrappings do not give the same guarantees of safety and correctness against concurrency errors with calls wrapped in this way, as state can be maintained in the C library outside of the occam-pi runtime environment; these calls also affect the scheduling of processes, as they must switch to executing a C function.

A more advanced method for interfacing to C called CIF is available in occam-pi via the KRoC toolchain [Baro5]. CIF allows for processes to be defined in C and run alongside the processes written in occam-pi, with the same scheduling and communication characteristics. As typically the need for interfacing to C is rooted in the need for access to third-party libraries inside occam itself, being able to interface by writing C processes is of less utility. Additionally, due to the use of separately compiled C as part of the program at run-time this method is not supported by the Transterpreter runtime used for running occam-pi on robot platforms.

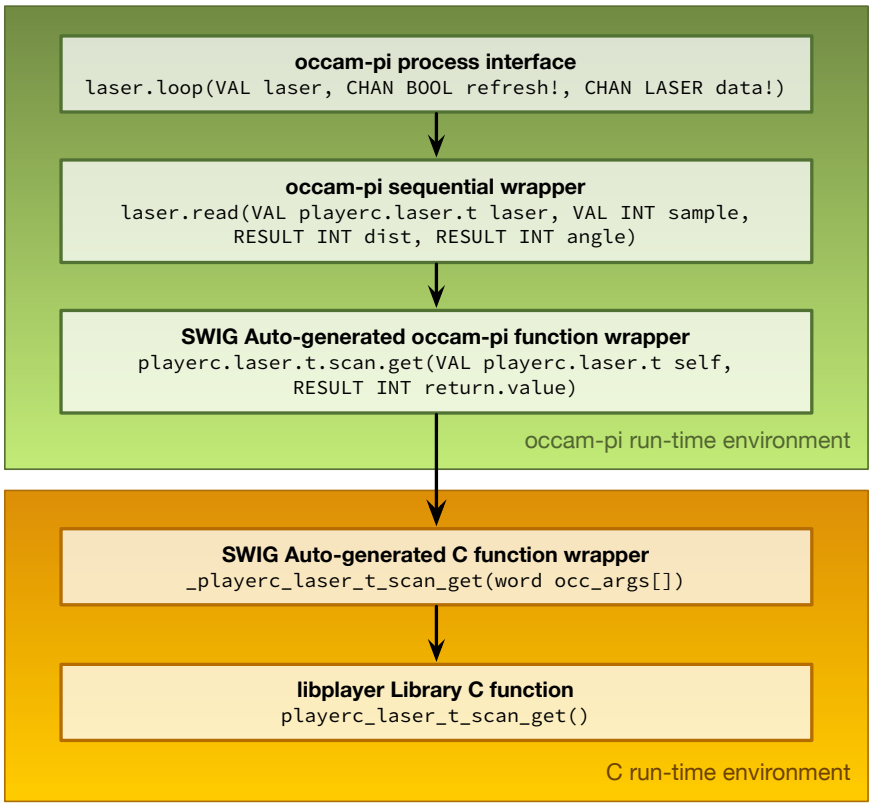


Figure 3.1: *Architectural diagram of the occam-pi process interface and C Foreign Function wrapping for reading from a laser in Player/Stage from occam-pi*

3.1.4 ActivMedia Pioneer 3-DX

The Pioneer 3-DX, produced by ActivMedia Robotics, has two wheel differential drive, sixteen ultrasonic range-finders around its circumference, and a high resolution laser range-finder, which provides centimetre resolution to an eight meter distance in a forward-facing, 180° arc. The particular robot used for these experiments contained a 700MHz PC104 board and was running Debian GNU/Linux; much faster boards are available for fitment in contemporary versions of this robot.

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, which provides a control interface for all of their robotics platforms [Ade12]. Third, and most interesting is the open-source Player API, a cross-platform robotics API written in C/C++ and compatible with the

Stage simulation environment [GVH03].

Player provides an abstracted driver interface for motors, sensors, and other devices typically found on a robot, allowing control logic to be ported between supported robot platforms with minimal modification to the program. Player is designed to be run as a client/server application, meaning code to generate control signals can written against the client library and run on a remote desktop PC, while the server (controlling the hardware) runs on a robot directly. The two may be connected via ethernet, Wi-Fi or a serial port.

This client/server separation also makes the use of a simulator significantly more straightforward; a simulator can present itself as a player server in an identical way to an actual player server controlling a physical robot platform. The difference between simulation and the real robot is hidden from the program interfacing to Player. There are two simulators in the Player software distribution: Stage, a 2D simulator capable of displaying dozens of robots simultaneously and Gazebo, a 3D simulator which provides a virtual world complete with accurate physics for more in-depth testing of control algorithms.

Platform support in the RoboDeb project by Jacobsen et al. was accomplished through use of Player/Stage, interfacing to the Player robot driver software via its C library interface [JJ07]. A number of wrapping processes were created around these FFI calls, effectively allowing direct usage of the underlying C functions to the user program with minimal translation of parameter types and other minor changes. A secondary layer of process interfaces were supplied as part of this support library, offering a channel interface to the motor and ultrasound ranging hardware rather than the strict sequential interfacing of the C API. A deconstruction of the architecture of this wrapping is shown in Figure 3.1.

The layers of support code and indirection between the original C code and the user space occam-pi processes are undesirable but do not prevent effective use of the Player/Stage environment for process-oriented robotics when running on a high powered desktop computer or on the Pioneer 3 robot (a 700MHz PC board in the model used at the time, but currently available with up to 2.26GHz Dual Core CPUs). The removal of these layers is a desirable goal, from a port maintenance, code path simplicity and execution efficiency point of view; these layers have a negative effect on the ability to reason about the behaviour of user programs, as there is potential for state and timing-related errors given the interplay between them.

3.1.5 LEGO Mindstorms RCX

The LEGO Mindstorms RCX is the first in the series of the LEGO Mindstorms robotics kits, based on the concepts of the original programmable brick developed at MIT. The programmable brick from which the set takes its name (shown in Figure 3.2) is known as the RCX; the RCX contains a small embedded system consisting of a 16MHz Hitachi H8 architecture CPU, 16KB of ROM containing low level routines to operate the hardware and 32KB of RAM for firmware and user programs. On the surface of the RCX are three input and three output ports, a LCD display for indicating the state of the system and four push buttons to control power and program execution. The RCX is fitted with an infra-red (IR) port which allows for communication, this IR port is also used for uploading firmware and programs to the brick from a host computer.

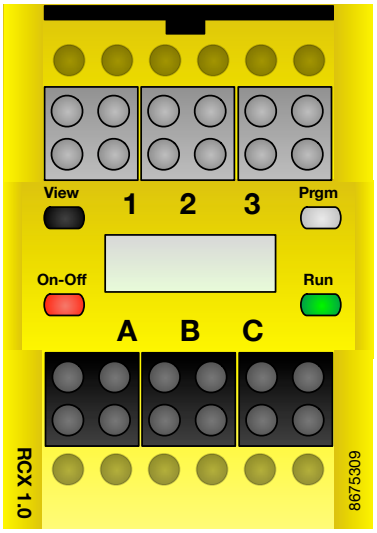


Figure 3.2: *An illustration of the RCX programmable brick supplied with the LEGO Mindstorms RCX invention kit*

The RCX is a constrained environment in which to run a virtual machine, custom designed firmware and a user program. When providing a user environment for programming the RCX the three elements are part of a constant trade-off: every kilobyte of RAM consumed by the runtime, firmware, or interfacing code is a kilobyte less available for user programs. The effect of these constraints was clear in Jadud and Jacobsen's original port of the Transterpreter virtual machine (TVM) to the RCX, where support was achieved by executing the virtual machine as a program running on top of an existing third-party OS for the RCX [JJ05]. In this

first port Noga's BrickOS [Nog04] handled running the RCX itself, while the TVM and user program were combined, uploaded and run as a single BrickOS program. This particular set of trade-offs meant there were serious constraints on the use of occam-pi on the RCX beyond the most elementary examples; either programs grew too big for the combined program and VM to be uploaded, or programs failed at run-time as there were no space in RAM for allocation of memory after the combination was present.

Given these practical constraints and the desire to investigate the possibility of a process-oriented firmware, the author implemented a replacement port of the TVM to the RCX, keeping as much of the code comprising the port inside the occam-pi run time environment as possible. Some small C functions were written to allow calls into the RCX's ROM and handle passing of addresses/values back and forth to those calls. All of the higher level interaction is composed in occam-pi inside the virtual machine. A minimal amount of C and H8 assembly is used to initialise the hardware and pass control to the TVM's scheduler. Putting the majority of the runtime code inside the occam-pi environment results in the scheduling of hardware processes and semantics of the hardware interface being consistent with the user program. Existing commonly used run-times for the RCX, LeJOS and BrickOS, use a time slicing model of concurrency presented via a threading interface to the programmer. This approach of providing a threading library has the potential for programmers to create many classes of concurrency error [Boe05].

Figure 3.3 shows the memory use of both the original the Transterpreter port when running as a program within BrickOS and the native port described here, removing BrickOS and replacing it with an occam-pi-based operating environment. Both the TVM and a replacement operating system for the RCX fit within 11KB of RAM, 1KB less than BrickOS running alone and providing an additional 9KB of space for program storage and memory usage when comparing each running the TVM and providing an occam-pi runtime. Additionally, a number of interface processes to provide hardware abstractions would be contained within the program bytecode itself - in the native port, fewer of these processes are required as a number are provided directly by the occam-pi-based operating environment. The combination of an occam-pi runtime and operating environment provide a safe, message passing based interface for using the underlying hardware from concurrent programs.

The reconfigurable nature of the RCX, with generic 2×2 stud LEGO brick input and output ports, is a challenge for software reconfigurability; users may connect any type of sensor to any of the three input ports (labeled 1–3) and any type of actuator to the three output ports (labeled A–C) at any time. The hardware itself is not capable of distinguishing between the

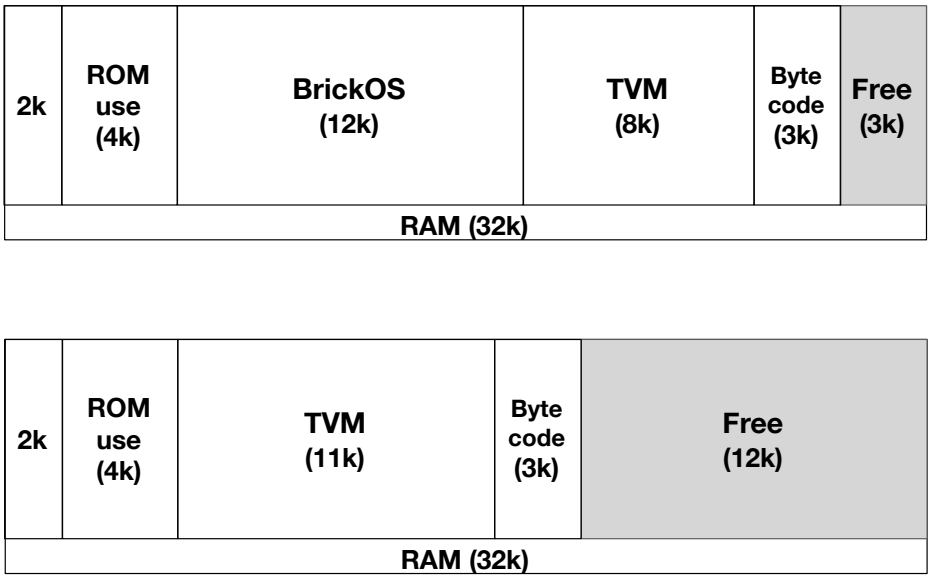


Figure 3.3: *Memory consumption for the Transterpreter VM running as a BrickOS program (top) and natively, without an underlying operating system (bottom)*

types of sensor or motor connected, meaning information about which sensors and motors are connected to which ports must be contained within the user program. This modular connectivity can be reflected in the architecture of software interfacing components; as the physical configuration of the robot changes, the topology of the process network can be reconfigured and altered in a way that directly reflects the physical changes.

A sensor can be represented by an individual process - `light.sensor`, which is parameterised with a constant which indicates the location of the sensor (e.g. `SENSOR.1`). This process wraps up all underlying calls to the hardware, in this case wrapping over a sensor process. The user is presented with an API which provides streams of sensor data over channels coming in from sensor processes and a set of actuator processes which accept commands over a channel interface. The user can specify the configuration of the hardware through using different kinds of processes parameterised to read from different ports on the hardware.

When creating a hardware interfaces to the RCX, a type based-system was used whereby channels carrying data to and from the processes interfacing to the hardware were given a type according to the kind of sensor or actuator. This permitted type enforcement when building components for robot programs — a process could express in its interface explicitly that it expected a light sensor input, a touch sensor input and a motor control output. This idea is ultimately flawed outside of a fully graphical environment, given the lack of generic

types in occam-pi, the simplest components for dealing with channels of integers (thresholds, deltas etc.) have to be rewritten for every type in use.

When writing programs textually this means one of the first language features that must be taught is the type system and typecasting to allow these individual types to be converted to and from standard language datatypes. This explicit conversion step places a barrier between communication between processes and the processing inside each process, requiring conversions to and from individual types per sensor or actuator. As individual processes inside the network away from the hardware interface may use standard data types for communication, there is also a lack of consistency in how this type conversion must be applied. Using standard datatypes for the hardware interface, allowing the re-use of predefined generic components like multiplexers or deltas and removing the need for type conversion reduces the complexity of applying process-oriented programming and makes a graphical environment more useful.

3.1.6 Surveyor SRV-1

The Surveyor SRV-1 is a small tracked robot platform designed with the goal of encouraging explorations of computer vision for mobile robotics [Goro8], shown in Figure 3.4. The only source of sensory input is a fixed forward-facing 2 Megapixel camera along with two forward-facing laser pointers. The robot uses a 500MHz Analog Devices Blackfin DSP processor along with 32MB of SDRAM and is equipped with Wi-Fi connectivity, provided as a serial IO interface to the hardware.

Wi-Fi connectivity on the SRV-1 is provided by a serial interface exposed over a TCP/IP port, meaning it is relatively trivial to communicate with from the desktop. This network capability allows the creation of programming environments for the SRV-1 which communicate with the robot directly to upload new programs, streamlining the process of working with a remote robot host separate to the development machine.

The SRV-1 is the same physical size and similar in motion capabilities to a basic Mindstorms RCX robot with tracks (in fact, the early production SRV-1 used in this work uses rubber tracks identical to those supplied with the RCX). These physical similarities provide for a significant contrast in capability between the SRV-1 and the RCX; the SRV-1's significantly faster CPU, camera and WiFi connectivity make it more computationally interesting as a platform for robotics but additional cost dramatically reduces its practicality for classroom teaching.

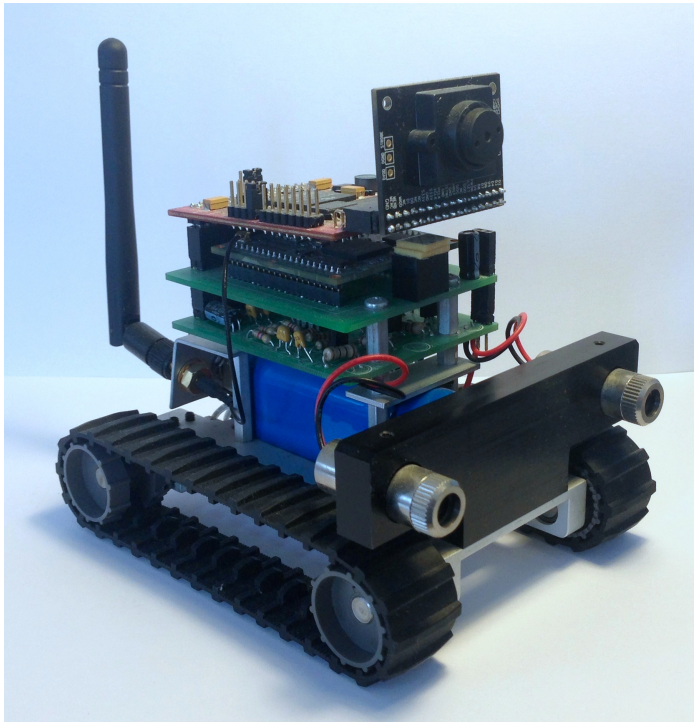


Figure 3.4: *The Surveyor SRV-1 Mobile Robot*

The SRV-1 introduced a problem not found in prior platforms - the development of a channel-based interface for streaming video from the 2 Mega-pixel camera on the board. Whilst the Pioneer is also fitted with a camera (in the case of the robot available to the author, a high end Sony EVI-D70 Pan/Tilt/Zoom video camera with independent serial control of the camera's motion) the Player device proxy controls the image data stream and converts it into a blob stream, significantly simplifying its use. The Pioneer is sensor rich, with a SICK LMS200 laser rangefinder and 16 ultrasound rangefinders mounted around its circumference, allowing a lot of use of the robots without using the camera feed. In contrast, the SRV-1 has only a front-facing camera, with ranging ability provided by two laser pointers also facing forward. These pointers provide ranging via measuring their size and intensity in frames from the camera. The lack of sensors makes effective use of camera data on the SRV-1 from occam-pi critical to meaningful use of the robot platform. Work with occam-pi in the classroom typically allows the the cost of communication along channels to be ignored, as channels are typically typed `INT`'s, `BYTE`'s, or `BOOL`'s — very small data types and typical communications consist of single values. Camera data, in the form of entire bitmapped frames, has very different properties and considerations required in the design of process-oriented programs. Communication patterns, the costs of message passing and process execution performance become important

when input messages take the form of 2MB frames from the camera being delivered at a constant rate.

The SRV-1 port of the Transterpreter was constructed with the same aim as that of the RCX port: to provide a process-oriented hardware interface running with a minimal amount of code running outside of the occam-pi run-time environment. Making the interface code and firmware behave with the same semantics as a user program allows the programmer to reason about the concurrent behaviour of the interface and its interactions with their program.

On the RCX, the ability of the run-time environment is limited by the capabilities of the hardware. Given the RCX's 16 MHz CPU and 32KB of RAM it is unrealistic to retain the hardware interfacing processes in memory and dynamically reconfigure the process network, or use channel bundles and dynamic memory. Hence the RCX uses an approach whereby a minimal occam-pi firmware is present to receive programs over IR and start/stop their execution, leaving the hardware interface to be compiled into and supplied as part of the uploaded user program. Compiling hardware interfaces into the program means they must be loaded individually with every user program, as the virtual machine considers the hardware interface part of the user program the processes are scheduled together, in a single scheduler. The downside of this approach is that a run-time error condition in the user program renders the entire user program (including hardware interfaces) unresponsive, as the occam-pi processes handling button presses to stop the program are also stopped due to the error. For serious application of process-oriented programming to robotics this kind of run-time behaviour is unacceptable.

There are significant safety concerns in the robot control program and its runtime becoming unresponsive; any protection built into the firmware to ensure actions are stopped will not take effect. This behaviour also poses problems for introductory, pedagogic use; introductory programmers are likely to create errors and having the entire platform lock up means no useful information can be gleaned about the error. Strategies for providing debugging capability to identify such errors are discussed further in Chapter 5.

As the SRV-1 has a significantly faster processor and much more RAM, a new method could be used; the virtual machine was extended to allow multiple *execution contexts*, meaning multiple occam-pi programs could be run simultaneously side-by-side in a single virtual machine instance. An interface for communicating between these contexts is provided, along with the ability to create new contexts and control their execution. This permits a design of the firmware as two separate occam-pi programs: a firmware program, offering a process-

oriented interface to the hardware, code loading and execution control; and the robot control program, which operates independently and may encounter run-time errors without halting the entire system.

The ability to run an occam-pi program from inside another occam-pi program without affecting its own execution allows a process-oriented program loader and firmware to be run which can accept subsequent programs over the Wi-Fi serial interface. The hardware interface and program loading code runs permanently as firmware in one half of the virtual machine; it provides a message passing interface which allows the user program to receive sensor input and control the hardware. Figure 3.5 shows the architecture of this virtual machine port supporting multiple contexts, with the hardware interface loaded and connected to an example user program.

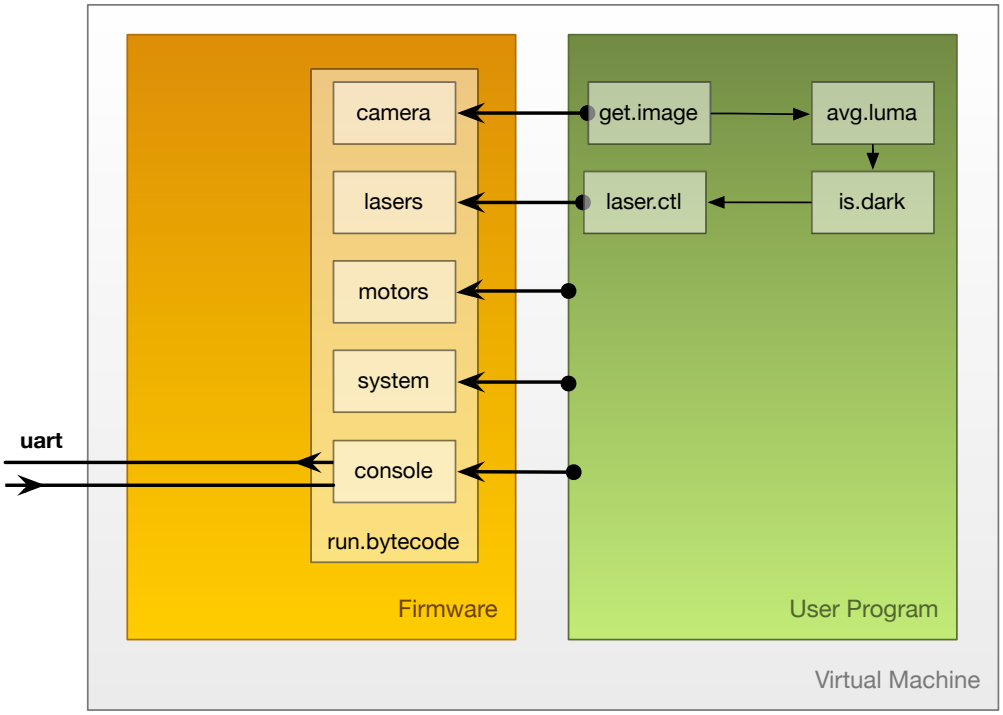


Figure 3.5: *Architectural diagram of the Surveyor SRV-1 port of the Transterpreter virtual machine, showing the firmware processes active and a user program loaded.*

Channel bundles are used in the SRV-1 interface, as their name would suggest, these are collections of channels defined as bundle types. This allows the provision of a client/server interface to pieces of the hardware – a programmer may request a frame from the camera on the request channel of the camera interface and get the response on a separate channel typed appropriately for mobile byte arrays. The client/server interface is a specific communication

pattern in occam-pi, designed to avoid deadlock when correctly implemented [MW97].

The use of channel bundles adds complexity to the interface between the program and the robot; students would typically only deal with separate channels carrying basic types. However, the result is a more powerful and flexible interface to the hardware which provides full parity with the manufacturer's supplied firmware whilst also providing access to lightweight concurrency primitives; the use of a threading model to permit the same degree of concurrency in C would increase the complexity of programs. A case study is presented in Section 3.6 which examines the properties and performance of this process-oriented firmware running an occam-pi program against the manufacturer supplied imperative firmware and an equivalent C program.

3.1.7 LynxMotion AH3-R

The LynxMotion AH3-R Hexapod is a six legged walking robot [Lyn]. Each leg is roughly 60 degrees apart and driven by three servos, giving it three degrees of freedom: swinging forward or backward, raising or lowering, and extending or contracting. The Hexapod is fitted with ultrasound range finders on a rotating turret covering 360 degrees around the robot, controlled by an Arduino based micro-controller. There are also two tilt sensors, mounted at 90 degrees on the robot's body and pressure sensors on each of the robot's feet. The robot is controlled via a serial link to a host PC. The Hexapod stands apart from the other ports discussed above, as the robot architecture work conducted on it consists of occam-pi programs written on the desktop for the KRoC toolchain, using a serial library to communicate with the robot itself.

3.2 Braitenberg Vehicles

Braitenberg Vehicles are a series of psychology thought experiments, designed to invoke reactions of human-like emotion with very simple, directly reactive systems created by Valentino Braitenberg [Bra86]. They also form a set of the simplest possible reactive control systems, expressed through direct wiring between input sensors and output actuators in their original form. Two of Braitenberg's original vehicles are shown in Figure 3.6. These vehicles contain two light sensors and two wheels, with a pair of each on either side of the robot.

In Vehicle 2a, stimulus to the light sensor on the left side is delivered to the left motor, and

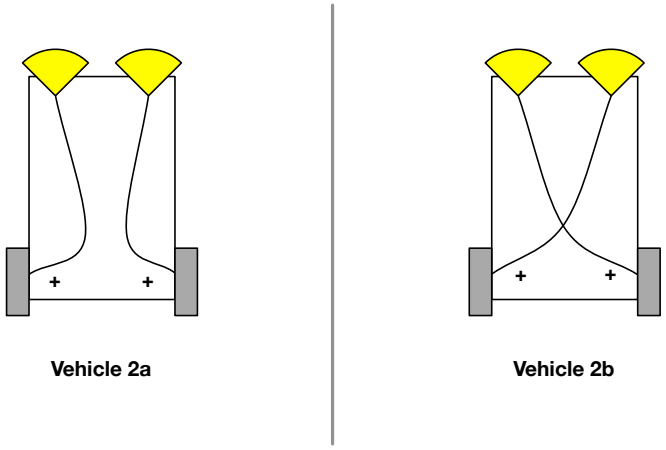


Figure 3.6: *Vehicle 2a, a Robot that 'avoids' light, and Vehicle 2b, a Robot that 'likes' light*

vice versa on the right; when vehicle 2a encounters more light on its left, the speed of its left motor is higher than the speed of its right motor, turning itself right, away from the light. In Vehicle 2b, the connections are inverted, meaning higher light levels on one side of the robot will speed up the opposite motor. This results in the robot turning toward light sources; a higher light level on the left of the robot increases the speed of the right motor, turning the robot left. The general light level of the environment will control the overall speed of movement for both vehicles.

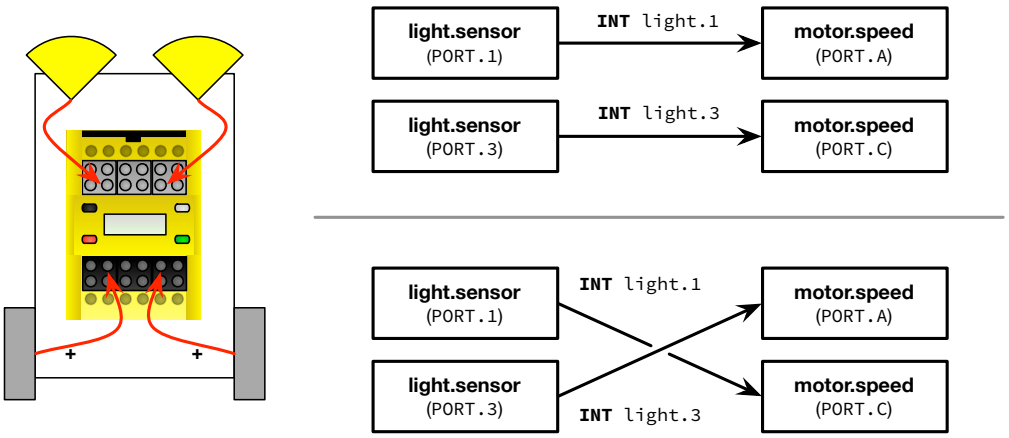


Figure 3.7: *A RCX controlled vehicle and occam-pi process networks for robot programs that 'avoid' light (top) and 'like' light (bottom)*

The experiments have value as an exercise in introductory process-oriented software engineering, mapping the direct connection to connectivity between processes, connecting the output

of sensor reading processes directly to the inputs of the actuators. The process networks required to implement these vehicles are straightforward, as shown in Figure 3.7, along with a diagram of a RCX-based vehicle platform for clarity. The RCX-based vehicle has the left and right light sensors connected to ports 1 and 3 respectively, and the two motors connected to ports A and C respectively. Given processes that scale their values appropriately, in this case to the range -100–0–100, they can be connected together directly to implement vehicles. The only configuration required of the processes is setting which hardware ports the sensors or actuators are connected to. No sequential code is required, as the solution is pure process composition and channel creation; this lack of programming and focus on wire-up makes the vehicles a suitable environment in which to motivate visual process network composition, discussed fully in Section 4.3.2.

3.3 Process-oriented Robot Architectures

Behavioural robotics, as introduced in Section 2.3.4, involves robot systems composed of independent behaviours. Given the concurrency inherent to managing multiple behaviours running on a single robot, the use of a programming model in which components may run concurrently and which provides facilities for data sharing and communication between concurrently running processes is an area of interest. The separation of robot control into individual, independent tasks found in behavioural robotics has a similarity to the decomposition of a program into individual, independently executing processes in the process-oriented model.

In Behavioural robotics, rather than building a single shared world view with sensory input, each component is designed to receive just the subset of sensor data required for its function and process it independently. In the absence of this planner-built unified world view, behavioural systems rely on the consistency of the world itself as measured by sensors to co-ordinate between the different components in the system.

The application of standard process-oriented decompositions of robot control programs results in programs with a large number of concurrently executing components. There are a number of existing robot architectures, which aim to provide structuring principles and patterns for interaction between components, specialised for robot control. Implementing these architectures in occam-pi, as a process-oriented language, serves two goals:

- To examine the suitability of occam-pi and by extension process-oriented programming

as a foundation for building robotic control programs following existing methodologies.

- To enable a specialised set of primitives and principles for constructing robot programs to be used in occam-pi.

Expressing these architectures in terms of a process-oriented programming language, such as occam-pi, allows us to distil design rules, structures and primitives for use in the development of process architectures for robot control.

3.3.1 Subsumption Architecture

Brooks' subsumption architecture [Bro86] was one of the first behavioural control systems, allowing robot programs to be expressed as a hierarchy of levels of competence which interact with one another to control the robot. Brooks' original Subsumption architecture was implemented used a network of finite state machines known as *modules*, along with asynchronous message passing between input and output *ports* on these modules. Brooks' terms the connections between these ports *wires* for reasoning purposes. Modules output values to a port and the most recent value output to that port is available for input to the receiver constantly essentially providing a memory cell located in front of the port into which values are written. These communications are asynchronous due to assumed unreliability of the message sending between components; one-place buffering provides resilience by allowing the process to execute using the latest successfully received data.

The subsumption architecture contains two unique primitives used for composing modules together; inhibition and suppression. Inputs to a module may be *suppressed* for a period, replacing them with a different source of input and discarding the original input. Outputs from a module may also be `inhibited` for a period, causing them to be discarded entirely. Modules are composed into layers, each adding an increasing *level of competence* to the robot's behaviour. These layers co-ordinate by using the suppression and inhibition mechanisms to reconfigure and control communication between modules in lower layers of the system and the robot's hardware interface.

In Brooks' original implementation, these modules are compiled along with a scheduler which supports executing them both concurrently on a single processor and in parallel across multiple processors. The ability to mix both concurrency and parallelism was exploited in

Brooks' six-legged walking robot "Genghis", which had a control system consisting of 57 finite state machines running on four processors [Bro89].

Process-oriented programming makes implementing robot control programs using the Subsumption Architecture straightforward, as a number of the primitives can be mapped directly onto the primitives of the process-oriented model. The concurrently executing, finite state machine modules become processes, the wires connecting them become channels, with input and output ports being channel ends.

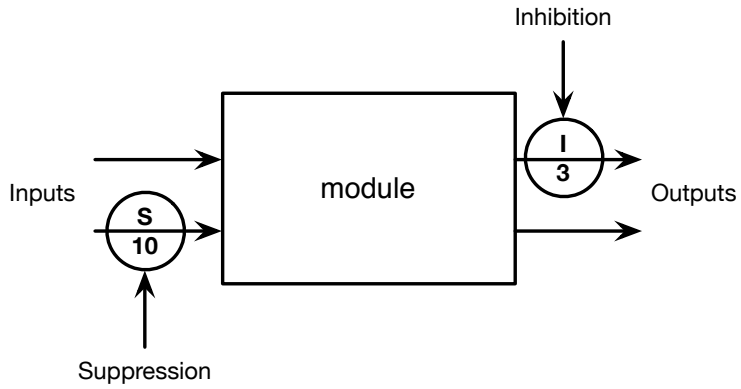


Figure 3.8: *A module for a Subsumption Architecture, with a suppressor on an input line and an inhibitor on an output line*

Figure 3.8 shows a module in the style of Brooks' diagrams with both suppression of an input wire and inhibition of an output wire. The circles are used to indicate inhibition or suppression, via the I or S at the top of the circle and the value at the bottom is the specified timeout period before usual input to or output from the module resumes. Diagrams of this style are used in Brooks' 1984 paper to show the networks of modules and their wiring; there is a considerable resemblance to the process network diagrams used in process-oriented programming due to the similarity of the two models.

As process-oriented programming provides synchronous channel communication, the one-place buffering behaviour of reads and writes in Brooks' original model can be removed; processes act only when there is new input rather than having a buffer to ensure a value is always available when they execute. Suppression and inhibition in the original subsumption architecture were specified in terms of wire behaviour, which cannot be mapped directly onto the process-oriented model, as channels cannot have programmed channel end behaviours. Suppression and inhibition can be modelled as processes with the original semantics from Brooks' wire ends; using processes for these primitives makes them more evident in the

design of the program at the network level. This increased visibility has a correlation to Brooks' diagrams, which show inhibition and suppression as individual components despite them not being separate components in the original model. The semantics of each primitive and an example process-oriented implementation are presented below, these two process templates combined with the language model allow the full expression of subsumption architectures.

Suppression

To suppress an input to a module an additional, suppressing, wire is connected to the input port. When this second wire becomes active the messages received along it are sent to the module as input, and any messages received on the normal input are ignored once this input occurs. The original input will be ignored for a set timeout period once a message is received on the second wire. The second wire effectively replaces any input on the ordinary input wire for the length of the timeout, *suppressing* the input.

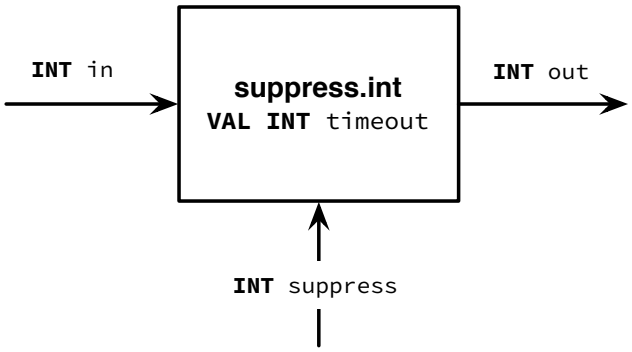


Figure 3.9: *Process diagram for* `suppress.int`*, an occam-pi process which acts as a suppressor on channels of integers*

The interface of a process that implements suppression is shown in Figure 3.9, and a sample implementation in occam-pi is shown in Listing 3.1. The sample implementation deals with channels of integers, but this can be used as a template to implement such a component for channels of any type. A boolean flag is stored which controls whether or not the suppressor is active, and the process continuously reads from both its usual input source (`in?`) and the suppression input (`suppress?`). When not suppressing, inputs on `in?` are passed through to `out!`, inputs on `suppress?` flip the boolean flag to activate suppression and then start the timeout period. Whilst suppressing, inputs on `in?` are discarded and inputs on `suppress?` are passed to `out!`, through to the process. The timeout is monitored whilst suppressing and

once it has expired the boolean flag is reset to disable suppression.

Inhibition

Inhibition of outputs is similar to suppression; to inhibit the output of a module a second, inhibitory, wire is connected to the output port. However, when any message is sent along this wire it prevents the usual output from the module, causing the messages to be discarded for a fixed timeout period. Unlike suppression, where the messages take the place of the usual input, messages on the inhibitory wire are also discarded - they do not take the place of the usual output. The second wire mutes the module, *inhibiting* its output.

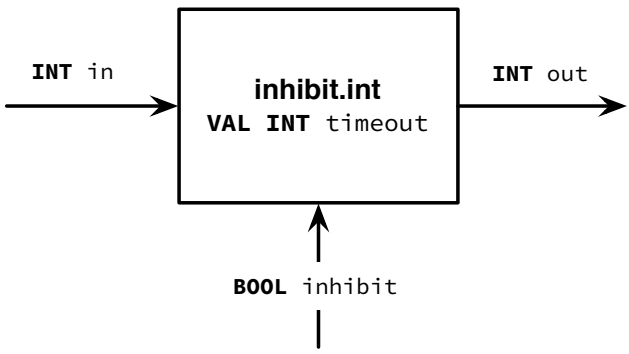


Figure 3.10: *Process diagram for* `inhibit.int`*, an occam-pi process which acts as an inhibitor on channels of integers*

The interface of a process that implements inhibition is shown in Figure 3.10, and a template implementation in occam-pi for channels of integers is shown in Listing 3.2. The implementation of the inhibitor shares much with the suppressor, although it can be simplified as data received on the inhibitory input channel can be discarded, whereas on the suppressing channel of a suppressor the data must be passed on. A boolean flag is used to track whether the component is currently active. If the inhibitor is not active, messages on the usual input, `in?`, are passed to the output, `out!`, and messages on the inhibitory line `inhibit?` set the timeout and toggle the boolean flag to set the component active. If the inhibitor is active, messages from from `in?` are discarded, and messages on the `inhibit?` channel reset the timeout afresh. Whilst active, the inhibitor also checks the current time to see whether the timeout period has expired, and the component should be reverted to inactive.

```
PROC suppress.int (VAL INT timeout,
                   CHAN INT suppress?
                   CHAN INT in?, out!)
  TIMER timer:
  INT time:
  INITIAL BOOL suppressing IS FALSE:
  WHILE TRUE
    INT value:
    PRI ALT
      -- When not active, activate on input from suppress.
      NOT suppressing & suppress ? value
        SEQ
          suppressing := TRUE
          timer ? time
          time := time PLUS timeout
          out ! value
      -- When not active, pass data from in to out.
      NOT suppressing & in ? value
        out ! value
      -- When active, deactivate after timeout.
      suppressing & timer ? AFTER time
        suppressing := FALSE
      -- When active, pass data from suppress to out.
      suppressing & suppress ? value
        out ! value
      -- When active, discard input on in.
      suppressing & in ? value
        SKIP
:
```

Listing 3.1: *An occam-pi implementation of* `suppress.int`*, a process which acts as a suppressor for channels of integers*

```
PROC inhibit.int (VAL INT timeout,
                  CHAN BOOL inhibit?,
                  CHAN INT in?, out!)
  TIMER timer:
  INT time:
  INITIAL BOOL inhibiting IS FALSE:
  WHILE TRUE
    PRI ALT
      -- Enable inhibiting until timeout on inhibit signal.
      BOOL flag:
      inhibit ? flag
        SEQ
          inhibiting := TRUE
          timer ? time
          time := time PLUS timeout
      -- Reset the inhibitor once the timeout expires.
      inhibiting & timer ? AFTER time
        inhibiting := FALSE
      -- Let data pass from in to out when not inhibiting.
      NOT inhibiting & in ? data
        out ! data
      -- Discard data from in when inhibiting.
      inhibiting & in ? data
        SKIP
:
```

Listing 3.2: *An occam-pi implementation of* `inhibit.int`*, a process which acts as an inhibitor on channels of integers*

Process-oriented Bump and Wander with Subsumption

To demonstrate the design and composition of a process-oriented subsumption architecture a *bump and wander* example program is presented. Bump and wander is an elementary mobile robotics example, requiring the robot to navigate safely within an enclosed space. One possible decomposition of this problem is as three tasks:

1. Move forward when there is clear space
2. Back away and turn if the path to move forward is obstructed
3. Move forward if an object behind obstructs the back away and turn behaviour

A process network for this simple bump and wander program using the subsumption architecture is shown in Figure 3.11, designed for use on the Pioneer 3-DX robot platform (described in Section 3.1.4) or a simulated Pioneer 3-DX in the RoboDeb environment (described in Section 2.6.1).

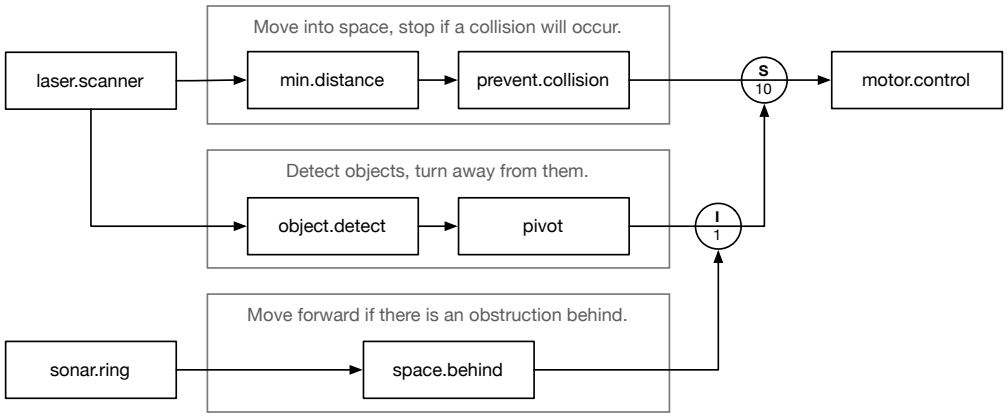


Figure 3.11: *A Subsumption Architecture-based bump and wander program for a robot with three levels of competence.*

The example program has three levels of competence, which correlate to our three task decomposition of the problem and build on each other, re-using elements of the underlying behaviours rather than duplicating logic.

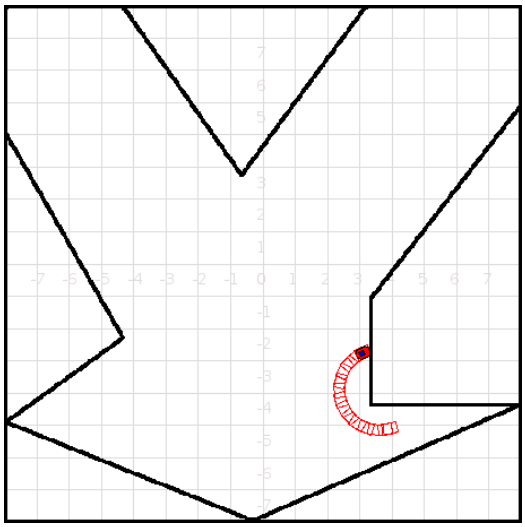
1. **Move into space and execute an emergency stop if too close to any object.** This is implemented using a pipeline of two processes: a process, `min.distance`, which takes

a 180° scan of laser ranges as an array and produces a single int corresponding to the lowest range as its output. A second process, `prevent.collision`, sends a stop motor command if this value is below a threshold (20cm in this case) or a motor forward command if the value is above. This level of competence means the robot will drive forward in its environment until it reaches an obstacle, and then stop. If the obstacle is removed, the robot will continue moving.

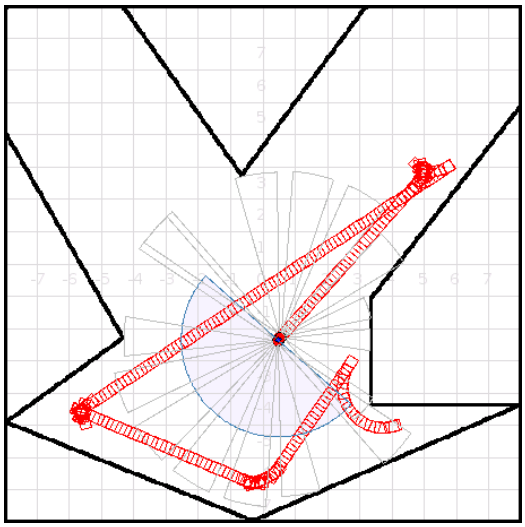
2. **Detect objects in the centre of the laser scan and back up to avoid them.** This layer is again implemented as two processes. `object.detect` looks at the central 90° slice of the laser ranges and sends a message if an object is below the detection threshold (75cm). `pivot`, on receiving a message sends a command to rotate the right motor on the robot backward, rotating it anti-clockwise and moving the robot backward. Importantly, this message suppresses the input to the motors from `prevent.collision` in the first level of competence, replacing any forward or stop messages to the motor with the back up message. The robot will back up for the timeout period of the suppressor (1 second), and the lower level behaviour will resume control. However, if turning for a second has not produced a clear path, the second level of competence will continue to generate messages and suppress the lower level competence until a path is available. This level upgrades the control system from simply driving into an area until it stops due to proximity to actually being able to navigate around the space. However, as the robot's emergency stop behaviour uses the laser range-finder at the front, the robot backs into walls while trying to avoid obstacles in tight spaces.

3. **Move forward if backing up is obstructed** Whilst the previous levels of competence used the laser scanner, it scans to the front of the robot and cannot be rotated. The robot has a number of sonar sensors around its circumference which can be used to find obstacles at closer proximity. The `space.behind` process monitors four sensors at the back of the robot and checks a minimum threshold on the distance; if this distance is too low, it inhibits the output from the second level of competence telling the robot to turn, meaning the first level behaviour of moving forward ceases to be inhibited, and the robot moves forward slightly during the inhibitory period.

This three competence network allows the robot to perform multi-point turns simply by engaging its *back up and turn* and lower level *go forward* behaviours alternately, with no explicit statement of the composite action.



(a) Demonstrating two levels of competence, the robot is able to turn and back away from the wall when it gets too close.



(b) Demonstrating all three levels of competence, the robot backs away from the wall and performs multi-point turns to navigate the environment.

Figure 3.12: *Simulation results when running two levels of competence and subsequently adding the third level of competence*

This program was designed for use on a physical Pioneer 3-DX and tested via the Stage simulator in RoboDeb; the program executing on the physical robot is able to successfully navigate an irregular space using laser ranging, performing multi-point turns when unable to continue forward motion. The Stage simulated environment is shown in Figure 3.12, showing the robot's motion trails and demonstrating the difference in robot behaviour as the additional level of competence which permits multi-point turns is added. Figure 3.12(a) shows the movement path of the robot when running with the first two levels of competence, rotating and backing away when it encounters a wall; Figure 3.12(b) shows the robot's behaviour when the third level of competence is added, making successful multi-point turns to negotiate a difficult environment. The value of the timeouts used to switch between going forward when there is no room behind the robot and the distance thresholds used in the program must be lower in physical environments with many tight corners. Where the thresholds or timeouts are set too high there is a possibility of the robot getting stuck, as it is too close to walls for both kinds of motion.

Limitations of Subsumption

Subsumption architectures have shown promise when used for robotics in occam-pi, and the adaptations introduced by Brooks' later work further enhance its suitability. A lack of modularity, identified by both the author and authors of other related control systems mean that systems using the Subsumption architecture tend to run into scaling problems, due to overly tight bindings established between behaviours inside the layers themselves.

Scaling the model of subsumption to larger systems is challenging due to the emergent way in which behaviours are achieved through the interaction between behaviours. These challenges are not specific to a process-oriented implementation of subsumption, rather to the architecture as defined. In simple programs, the small number of behaviours can be reasoned about simultaneously and there are a limited number of interaction points for subsumptive primitives, larger systems mean these values and the complexity scale beyond what can be maintained in the programmer's mental model of the system. Increasingly complex interdependencies form as higher levels of competence intercept values being passed between individual modules in lower levels, replacing their input or inhibiting output. The bump and wander program, presented earlier and shown in Figure 3.11, may be used to identify these scaling issues.

In the bump and wander program, reasoning about the motion of the robot requires understanding of the three levels of behaviour and the interaction of the three, as each alters the output of the previous. The actions taken by the robot reflect not only the combination of behaviours, but the specified timings in inhibitors and suppressors which control the interaction of behaviours. Figure 3.11 shows a larger, nine level of competence subsumption system developed by Posso which identified a number of difficulties in working with the architecture at scale in occam-pi [Poso9]. For nine levels of competence, there are twelve suppressors and four inhibitors, all of which contain timing information and produce conditional behaviour based on the sensory input to the behaviours triggering them. This level of complexity, combined with the complexity of the environment makes rationalising the robot's behaviour and current state extremely difficult from observation. The non-linear scaling and complexity of interactions mean that large subsumption architectures exhibit behaviour which is difficult to debug without the ability to inspect the state of the running system.

The interactions between behaviours mean that it is difficult to use the standard process-oriented approach to managing complexity, process composition, to abstract and present

sets of behaviours in isolation from the rest of the system. An abstraction containing a set of behaviours and their interactions may abstract and hide an input or output at which a further level of competence needs to inhibit or suppress control.

In his specification of the Colony Architecture, Connell identifies that Subsumption requires a holistic view and correct behavioural decomposition from the very beginning of design so as to offer the correct inputs and outputs for higher layers to interact with [Con89]. The Colony Architecture is discussed in more detail in Section 3.3.2.

In a later paper, Brooks revises the principles of inhibition and suppression, requiring continued communication over the inhibitory or suppressing channel and the use of short delay periods [Bro89]. The implementations of the subsumptive primitives presented here need no modification to be used in this way, as the use of a much shorter timeout period will also introduce the requirement for the channel to be consistently active to continue resetting the timeout.

Extensions of the Work

The author's work on process-oriented subsumption architectures has been extended by Neeson [Nee08], who reimplemented an existing subsumption architecture implemented in C++ using the process-oriented approach documented above to compare efficiency and ease of implementation. Neeson found occam-pi to be an "excellent environment" for the implementation of subsumption architectures, whilst expressing uncertainty about the scalability of the approach to more complex problems.

Posso implemented a complex subsumption architecture, shown in Figure 3.13, with nine levels of competence to determine the feasibility and ease of scaling process-oriented subsumption architectures [Pos09]. Posso concludes that occam-pi is capable of implementing complex subsumption architectures and is an "excellent environment for concurrent control" [Pos09].

Posso's complex subsumption architecture would enter behavioural stall conditions which were identified as fundamental to the subsumption approach. This work was conducted using the RoboDeb Player/Stage based simulation environment (detailed in Section 2.6.1), which interfaces to both simulated and hardware using foreign function wrapping of a C library. This interface approach and its introduction of conditions and potential problems outside of the process-oriented environment are documented in Section 3.1.3. A combination of

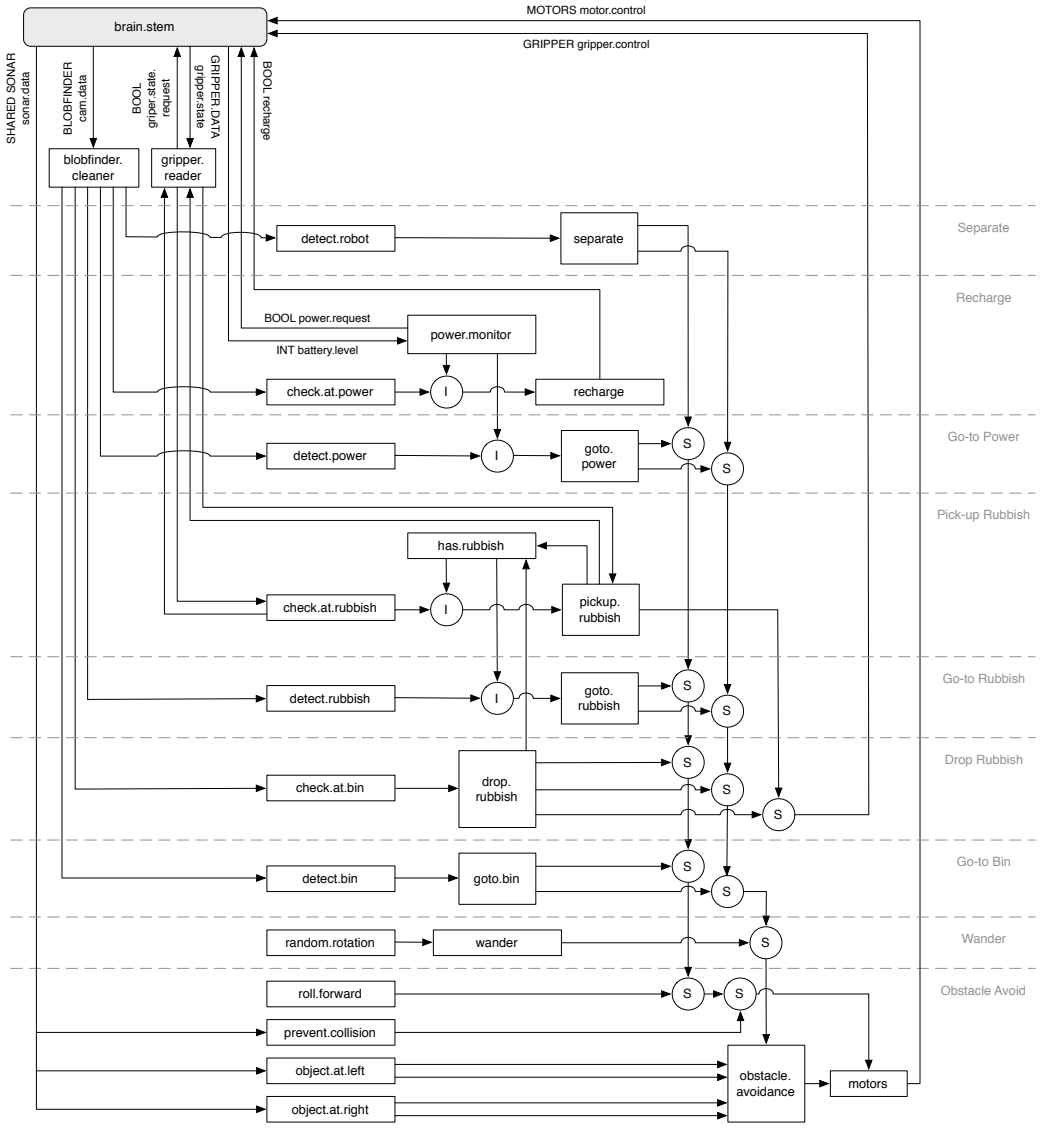


Figure 3.13: *A Subsumption Architecture-based bump and wander program for a robot with three levels of competence, from [PSST11]*

application of an introspection tool to examine program state (as presented in Chapter 5) and a robot platform with a more direct interface to the hardware would allow a full investigation of the behavioural stalls identified by Posso to be carried out and an understanding of the limitations of subsumption once platform deficiencies are isolated.

3.3.2 Colony Architecture

Connell's Colony Architecture [Con89] is a refinement on the early Subsumption Architecture which removes explicit inhibition allowing only suppression of outputs in lower level behaviours. The Colony Architecture uses a collection of modules which are related by the actuator they control, rather than a layered ordering across the system. Unlike Subsumption, additional layers do not necessarily increase in competence; modules may be introduced in a higher layer which provide more general solutions to lower-level control problems for cases where more specific lower-level modules cannot establish the correct action to take. The Colony Architecture removes the Subsumption's ability to "spy" on inputs and replace outputs of internal modules in lower layers, enhancing the system's modularity by only allowing the input and output of entire layers of behaviour to interact.

The Colony Architecture was used on a multi-processor robot which could also run programs written for the Subsumption Architecture. This multiprocessor robot used 24 loosely-coupled processors to run a system with over 41 behaviours, with software scheduling like that of Brooks' to run multiple behaviours per processor. Both Brooks and Connell identified that the explicit parallelism of behaviours allowed the control system to scale through the addition of more processors, maintaining reactivity and performance even with the introduction of more behaviours.

Implementation

A *retriggerable monostable* primitive is added, which is used when an event (defined as a single point of activity which does not persist in the environment) should trigger a behaviour. The detection of these events is performed using initiation and satisfaction predicates, which set the monostable true or false depending on the condition of the event. The monostable itself maintains a true value for a period of time, but eventually resets itself if not reset by the satisfaction predicate, acting much like a piece of memory with a watchdog timer. The monostable is used to persist the point state from the environment, allowing it to have time

to influence the system even if the state ceases to exist in the environment.

Modifications to the Subsumption primitives involve suppression requiring continued sending of messages over the control channels. Early Subsumption tended to use a single message and long delays, whereas the Colony Architecture and Brooks' later Subsumption both use short delays with regular messaging along the control channel.

3.3.3 Action Selection

Maes' Action-Selection model relies on a network of independent competence modules and the use of activation levels to control which modules are executed [Mae89b, Mae89a]. Activation levels in the network are propagated such that an executable module primes modules which can run after it in a task, while a non-executable module will prime those which run before it, causing activation to pool in the first behaviour in a task which is suitable for execution. Inhibition, or *conflictor* lines are connected between modules that oppose each other's behaviour, when a module with such a connection becomes active it inhibits the activation of the other modules which would impede completion of its task.

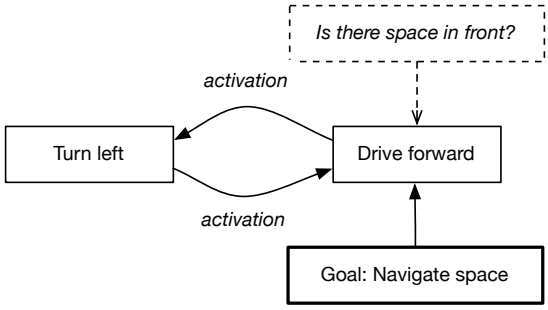


Figure 3.14: *A set of Action-Selection competence modules to move within a space. Activation spread is accomplished via bi-directional connections between modules, as shown.*

An example of a simple robot control program using Action-Selection is shown in Figure 3.14, a robot control program which has a goal to navigate into space. The competence modules for this program are *drive forward* and *turn left*, the drive forward module is preconditioned on there being space in front of the robot. The goal *navigate space* raises the activation of the drive forward module and it will become active, moving the robot forward. Once the robot runs out of space in front of it, the *drive forward* module will become inactive, as the *has space in front* precondition will become false. The drive forward module will then pass its activation on to the turn left module, causing a turn until the precondition for drive forward

is true again (i.e. there is free space in front of the robot).

Implementation

Use of an Action-Selection mechanism in occam-pi is most easily achieved through the creation of a second decision-making network consisting of a number of 'cells' which propagate the activation levels for each behaviour to calculate activation values for each cell. Cells which have an activation level over their threshold can be activated if they are *executable*. To be *executable* all of a cell's preconditions must be met, and the decision-making network will therefore capture all of the preconditions to allow it to arbitrate between behaviours and activate those that should be activated. These activated cells in the decision layer trigger the execution of behaviours in the robot control system itself, and free those control behaviours from having to incorporate the propagation of activation or management of preconditions. Goals are connected into the decision layer after the last module in a behaviour that completes them, and feed activation back through the modules responsible for completing the task until it pools in the first task, raising it above the activation threshold.

Partly due to its inspiration from neural networks, the process architectures that result from the implementation of Action-Selection are complex and require a second decision network to make their implementation relatively neutral to behaviours.

The Action-Selection model detailed here differs significantly from the Subsumption and Colony architectures in this formulation because it does not rely on message passing or finite state machines. The structure of the model in process-oriented programming defined here takes no advantage of the communicating process model for actual control code, making it difficult to integrate with systems using other architectures identified in this chapter, or with the use of process-oriented hardware interfaces. The need to implement a separate neural network based decision layer from the actual robot control processes and the highly connected nature of the decision layer adds implementation complexity that does not have any benefit in the process-oriented model. Arkin notes on Action-Selection in [Ark98] that "no evidence exists of how easily the current competence module formats would perform in real-world robotic tasks", due to lack of implementation on actual robot platforms.

3.3.4 Motor Schema

Arkin's Motor Schema approach to control uses multiple concurrent schemas (behaviours) active during the completion of a high level task [Ark87]. Two types of schema are employed in the architecture: motor and perceptual. Motor schemas are behaviours that control the motion or activity of the robot, such as *stay on path* or *avoid obstacles*. Perceptual schemas identify features and conditions in the environment that provide data necessary for a given motor schema to function, for example *find terrain* might supply a clear path vector to *stay on path*. In Arkin's system, multiple schemas may effect action at the same time, and these actions are merged through vector addition of potential fields.

Groupings of perceptual and motor schemas which achieve a given task are known as assemblages. Some assemblages may be present throughout the entire runtime of the control program, such as those that provide emergency stop or hazard avoidance facilities. Additionally, a planner module may load and unload different assemblages based on the input from the perceptual schemas connected to it. This mechanism provides effective re-use of components, as parameterised perceptual and motor schemas can be used across multiple assemblages.

To illustrate the Motor Schema approach, an example program is presented in Figure 3.15, written for the LynxMotion AH3-R Hexapod robot discussed in Section 3.1.7. The sample program shown in Figure 3.15 has two assemblages which are loaded constantly: one to detect body tilt (`detect.tilt`) and keep the robot level and another to detect obstacles in the path of the robot (`detect.obstacles`) and generate a vector away from them. The planner is able to load additional assemblages based on perceptual schemas which are connected to it. In this case two perceptual schemas are connected to the planner: one to detect lateral motion (`detect.lateral.motion`) which loads an assemblage to move towards (`investigate`) the source of the motion, and another which loads an assemblage which backs away from the source of the approach (`detect.approach`). The state machine for the planner is shown in Figure 3.16.

Motor Schemas provide the reactive component of Arkin's Autonomous Robot Architecture (AuRA), which combines the aforementioned planner with a spatial reasoner and other deliberative levels of function [AB97].

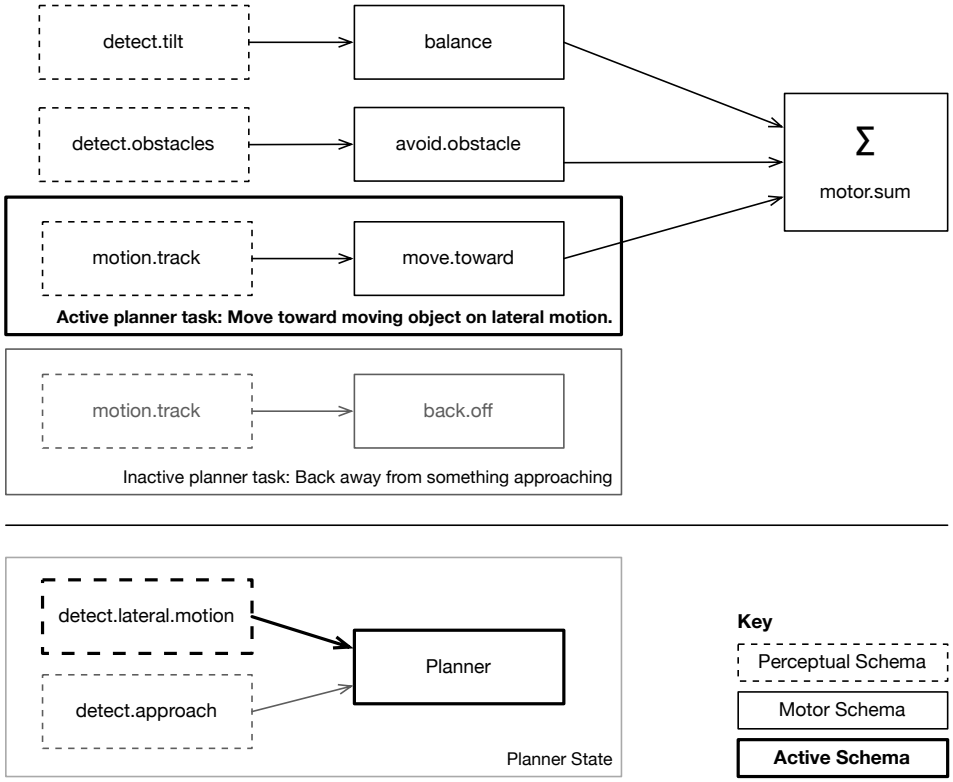


Figure 3.15: *A motor-schema based control program to navigate a robot to investigate motion and run away if approached.*

Implementation

The key primitive for the implementation of Motor Schemas is the vector sum, an implementation of which is shown in Listing 3.3. The example provided is suitable for controlling motion in a 2D plane, it is straightforward to add more components to allow control in a 3D space (x,y,z), allowing the system's behaviours to influence height or tilt. The planner in a Motor Schema based system is a custom-built state machine which uses perceptual schemas to determine when to change between states.

```
PROTOCOL VECTOR IS REAL32; REAL32:

PROC motors (VAL []REAL32 gain, CHAN []VECTOR in?)
  MOBILE []REAL32 x, y:
  SEQ
    x := MOBILE [SIZE in]REAL32
    y := MOBILE [SIZE in]REAL32
```

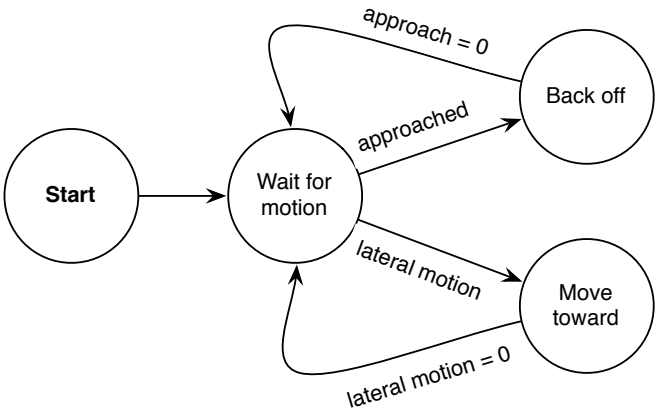


Figure 3.16: *State machine of the planner for an example control program using Motor Schemas which investigates motion and runs away if approached.*

```
      SEQ i = 0 FOR SIZE in
        x[i], y[i] := 0.0, 0.0

      WHILE TRUE
        INITIAL REAL32 x.v IS 0.0:
        INITIAL REAL32 y.v IS 0.0:
        SEQ
          ALT i = 0 FOR SIZE in
            in[i] ? x[i]; y[i]
              SEQ
                x[i] := x[i] * gain[i]
                y[i] := y[i] * gain[i]

          SEQ i = 0 FOR SIZE in
            SEQ
              x.v := x.v + x[i]
              y.v := y.v + y[i]

          -- drive motors ...
:
```

Listing 3.3: *An occam-pi implementation of the vector sum primitive which allows for the control of motion in a 2D plane.*

Arkin's Motor Schemas offer a method of command fusion which is simple and effective, matching with our typical use of process-oriented programming in robotic control and providing flexibility for the different kinds of motion possible on a variety of robot platforms. In the context of the wider AuRA architecture, the finite-state machine based planner and re-use of perceptual schemas make this approach to control modular and flexible, allowing both deliberative and reactive behaviours to be expressed in the same way.

3.3.5 Distributed Architecture for Mobile Navigation

Rosenblatt's Distributed Architecture for Mobile Navigation (DAMN) combines independent, asynchronous modules with arbiters performing command fusion via a voting mechanism [Ros95]. The overall goals of the system are prioritised via the weighting of votes placed by each module. Arbiters make a decision on the set of votes which have been received within their time step. This provides asynchronous operation of system components and allows behaviours to be a mix of deliberative and reactive modules, emitting decisions at different rates. Arbiters in DAMN offer a set of commands to behaviours; a steering arbiter might offer varying degrees of turn and the behaviour modules would then be able to vote on each possibility. Votes made by behaviours are normalised, and the choice which has the highest vote amount is chosen to occur.

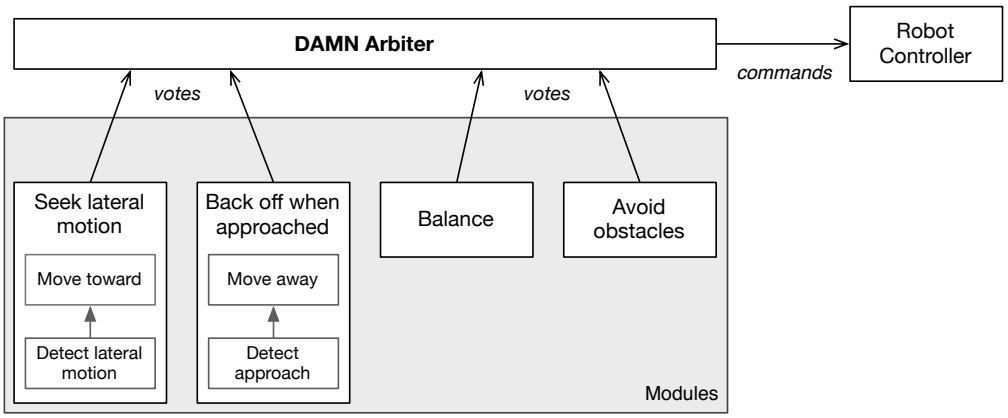


Figure 3.17: *A DAMN based control program to navigate a robot to investigate motion and run away if approached. The arbiter sends commands to the robot itself based on the votes made by behaviours.*

Arbiters may be connected to an adaptive mode manager, allowing the weighting of different behaviours to be changed while the system is running. A mode manager such as SAUSAGES altering the vote weightings allows for sequential action [Gow94]. For example, a robot might

have a primary stage of operation where it locates all target objects (soft-drink cans, red balls or similar) and a secondary stage where it retrieves all of those target objects. A mode manager would first weight highly all of those behaviours responsible for the target finding abilities, then reduce those weights and increase those to the behaviours responsible for retrieving the objects.

A process network decomposition for a DAMN architecture is shown in Figure 3.17. It performs the same task as the earlier example implemented using a Motor Schema approach, using the six-legged walker to approach moving objects and back away from objects that approach it.

Implementation

The development of process-oriented DAMN-based robot control systems requires the implementation of *arbiter* processes for each actuator to be controlled by the system. Each of these arbiters will offer appropriate choices for actions to take with the actuator that they control to the behaviour modules of the system.

Where Motor Schemas offer a blending approach to resolving the action of many behaviours into one coherent choice, Rosenblatt's DAMN allows the use of an arbitration-based approach. Behaviours vote on potential actions and decide on the correct action to take via a centralised arbiter, customised for the potential actions that can be taken for each effector. DAMN provides a framework which can exploit message passing for decision making without the implementation overhead and connection complexity of a neural network, whilst allowing complete freedom to the internal structure of the voting behaviours. The asynchronous nature of DAMN also allows a seamless combination of deliberative and reactive components each working at their own frequency, with the relative proportion of their inputs able to be counteracted through weighting. An AuRA-like planner or other mode manager could also easily be used to provide sequential or adaptive behaviour, providing a coherent framework.

3.4 Distributed Robotics Architectures

Modern advancements in robotics software architecture have moved toward providing middleware which facilitates the composition of systems composed from components co-ordinating via communication using well defined interfaces. These middleware systems

possess many of the advantages of process-oriented programming, although these advantages are present mostly at the coarse-grained component level rather than in the actual program components themselves.

The use of middleware architectures reflects the current state of software in general; often rather than building an entire system from scratch, programmers are required to integrate a number of established and well tested libraries designed for specific tasks along with custom logic into a coherent system. The use of a communication middleware is designed to promote the development of reusable best of breed components for a single task which may be used across robot systems. A variety of robotics middleware packages exist; the Robot Operating System (ROS)[QCG+09] has received significant attention from both academia and industry, while others such as Player [GVH03], YARP [FMN08] and Urbi [Gos08] offer similar co-ordination models combined with additional support for robotics or specific types of robots. ROS, in comparison, focuses primarily on providing only the tools needed to construct, run and debug programs, leaving robotics tasks to components within the program itself.

ROS provides a software framework for robotics based on distributed systems principles, permitting 'nodes' written in different programming languages and running on different physical hosts to communicate to form an overall system. This framework is designed to be lightweight, consisting only of a communications fabric to facilitate discovery of and communication between nodes. ROS contains two mechanisms by which nodes may communicate: a publish-subscribe asynchronous mechanism and a synchronous request-response model, allowing the programmer to choose the most appropriate form of communication for the transaction between two given nodes.

As both ROS and process-oriented programming involve individually executing components co-ordinating via communication they share the advantage of being able to build up programs compositionally, adding new and untested components alongside existing, well tested, components. However, as ROS is middleware, both the individual processes and communication are heavyweight compared to processes and message passing constructs in programming languages such as occam-pi or Erlang. Each process is a fully-fledged operating system process, networks of which are set-up via an XML configuration file which controls process instantiation and appropriate connection between processes.

The flexibility and power of message passing frameworks like ROS comes with a price; writing a software component requires the programmer to understand the communication model and distributed systems concepts surrounding its incorporation into a wider system. These communication and co-ordination elements introduce additional complexity when reasoning

about the entire system, asynchronous communication means that the system state is the combination of both the current activity of individual components and any messages left unprocessed in buffers. Due to the flexibility of these frameworks and the common transports used, a number of them are able to interoperate, allowing components written for a particular framework to be used under or interact with another. Player, for example, is able to connect to a YARP component to retrieve image data while YARP itself is able to interoperate with ROS, allowing the use of a mixture of YARP and ROS components in a single system.

In the light of these modern architectures, process-oriented programming enables the application of fine grained-concurrency at the component level, extracting performance benefits of parallel hardware and provide a consistent environment between the high level program architecture and component implementation. However, while process-oriented programming provides a message passing environment, interfacing into third party libraries typically requires technically complex and carefully designed interface code to avoid causing concurrency errors at the interface boundary (as discussed in Section 3.1.3).

3.5 Concurrency Patterns in Robotics

Process-oriented robot control programs in occam-pi prior to the work published in this thesis primarily consist of freeform process decompositions according to the techniques used for process-oriented programming. This thesis has presented the application of existing design principles for behavioural robotics architectures to process-oriented robotics. Inversely, there is significant value in applying design patterns and principles for process-oriented concurrent systems to robotics. These principles are captured in the introductory concurrency course at the University of Kent where students begin by working with Welch's elementary 'Legoland' processes, a set of fundamental building blocks for building data flow systems which can be used to teach basic design patterns for process-oriented programming. There are a number of established patterns for designing process-oriented systems that have arisen out of the years of experience at the University of Kent and elsewhere; a complete set of these patterns is defined and specified in Sampson's Doctoral Thesis [Samo8].

The author along with collaborators presented a workshop called "Patterns for Concurrency", introducing ten professional embedded systems developers to the process-oriented model of concurrency using these design patterns [JSJo8]. This nine-hour course set all of the educational material in the environment of writing programs for the LEGO Mindstorms

RCX, starting with the fundamental principles of process-oriented programming: channel declaration, communication over channels and creating parallel processes. This course concluded with a review questionnaire in which the participants were asked about their opinion of the workshop and the effect of the workshop on their practice. Of ten developers attending the course, all said the workshop was a good use of time; eight out of ten said they left with new perspectives that would influence their practice and planned to experiment further with the tools and materials provided.

3.6 Process-oriented Robotics: A Comparative Case Study

To evaluate the application of process-oriented programming for robotics a case study robotics task was designed for the Surveyor SRV-1 mobile robot. The SRV-1 is a useful platform for comparative work; the manufacturer supplied firmware and hardware interface for the robot is written in C and a fully process-oriented firmware and hardware interface has been created for the SRV-1 by the author and collaborators (as presented in section 3.1.6). The robot control program was first implemented in the imperative model using C, with the program compiled into and using the hardware interface of the manufacturer robot firmware. A subsequent, process-oriented implementation was completed using the hardware interfaces provided by the replacement Transterpreter and occam-pi-based firmware with the program loaded at runtime over-the-air.

3.6.1 Problem Definition and Experimental Setup

An environment was created in which the robot faces a light coloured wall with a black rectangle target placed directly in front of the robot, shown in figure 3.18. As the robot approaches the wall and target, the dark target area fills more of the camera's field of view, reducing the luminescence values in the frame; once the luminescence values reach or drop below a pre-calibrated threshold the robot should stop.

The robot program is calibrated at an initial fixed distance away from the wall and target, taking eight samples of the light level and using the maximum value as the threshold. Once this value has been set, the robot is placed at a 'start point', a fixed distance further away from the target, shown on figure 3.18 as the start point. The robot program waits until the light level rises from its initial calibration (i.e. the robot has been moved from the calibration/stopping

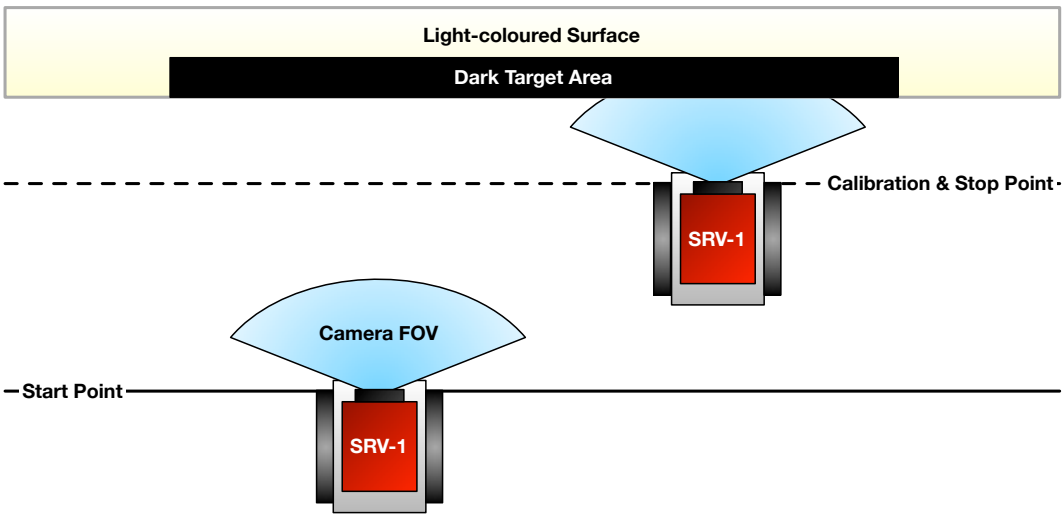


Figure 3.18: *Diagram showing experimental test using camera light level detection to establish the reaction time of the control system*

point to the start point) before triggering a delay period, allowing time for the robot to be accurately positioned. Given the field of vision of the camera and the SRV-1's speed of motion, a stopping and calibration distance 15 centimetres from the target was used, with the starting point a further 20 centimetres away. Once the delay has elapsed the robot drives forward at a specified speed, stopping the motors once the light level reaches the pre-calibrated value; the robot's reaction time and the deceleration of the robot will be exhibited as the distance past the calibration point that it travels before actually stopping.

The task is specifically designed to test the performance of the virtual-machine based approach, tying conditions sensed via camera data to motor control and observing a reaction. As the two firmwares share no common code, a physical calibration is used to avoid differences in hardware initialisation or hardware interface producing different behaviours given a constant numeric value. The light level is established from the camera by taking the brightness values of 64 pixels evenly spaced across the frame. Before the initial threshold setting phase of the program, the camera's auto adjustment for white balance and exposure run for several seconds and then disable the automatic adjustments. Disabling automatic adjustments is necessary to be able to use the camera to sense brightness, otherwise the camera would adjust for changes in lighting and the values sent to the program would be inconsistent.

3.6.2 Implementation Properties

As the C implementation of the robot program must replace the robot's firmware entirely, the entry point of the program must initialise the robot hardware; the occam-pi implementation can avoid all of the generic setup and is provided a high-level channel communication interface to perform program specific hardware configuration. The implementation of an occam-pi hardware interface as set out in Section 3.1.2 drives this high level approach, modeling the hardware as a communicating component rather. An example of this difference in abstraction is the camera on the SRV-1. The Surveyor's C firmware directly accesses memory storing the camera frame, reading image data directly from the memory it is written into by the physical hardware, whereas the occam-pi firmware passes a message once an entire frame is ready, along with a reference to the frame data. The latter approach provides a high level abstraction under which the camera can be reasoned about as a series of frames, rather than a memory buffer being continuously written to.

The main method of the C implementation is shown in Listing 3.4. The control logic consists of an initial calibration phase, then a loop performing the main task indefinitely. This main task loop relies on two further loops encapsulated within the `wait_for_dark` and `wait_for_light` functions, which encapsulate responsibility for detecting these conditions in the environment, looping and only returning when the conditions are found. The top-level process (TLP) of the process-oriented implementation, the occam-pi equivalent of a `main` method in C, is shown in Listing 3.5. In comparison to the C, which relies on a main infinite loop and individual loops wrapped in functions to block program execution, the occam-pi program contains an infinite loop inside each of the processes set up in the TLP; blocking in the process-oriented model is achieved via communication. The concern of co-ordinating the execution of the two tasks is mixed with the concerns of the actual robot behaviour, adding complexity to the program and making it more difficult to extend to support further behaviours.

The process-oriented implementation of the robot program consists of four processes executing in parallel:

- `get.image` configures the camera's automatic adjustment settings, then sends entire frames from the camera as byte arrays of pixel data on its output.
- `avg.luma` receives camera frames and gets the brightness values of the 64 point pixels used in the image and outputs a single brightness value for each frame it receives.

```
static void wait_for_dark(int level) {
    int luma;
    do { luma = get_luma(); } while (luma > level);
}

static void wait_for_light(int level) {
    int luma;
    do { luma = get_luma(); } while (luma <= level);
}

int main() {
    initialise_hardware();
    int threshold = 0;
    int i;
    for (i = 0; i < 8; ++i) {
        int luma = get_luma();
        threshold = luma > threshold ? luma : threshold;
        delayMS(200);
    }
    for (;;) {
        uart0SendString((unsigned char *)"delay"); uart0SendChar('\n');
        delayMS(8000);
        uart0SendString((unsigned char *)"start"); uart0SendChar('\n');
        wait_for_light(threshold);

        uart0SendString((unsigned char *)"delay"); uart0SendChar('\n');
        delayMS(8000);

        setPWM(100, 100);
        wait_for_dark(threshold);
        setPWM(0, 0);

        uart0SendString((unsigned char *)"stop"); uart0SendChar('\n');
    }
}
```

Listing 3.4: *The* `main` *method of the imperative robot program implementation, containing the control logic, and the two functions used to detect environmental conditions:* `wait_for_dark` *and* `wait_for_light`.

```
PROC stop.when.dark (
    CAMERA! camera, CONSOLE! console,
    LASER! lasers, MOTOR! motors
    SYSTEM! system
)
  CHAN MOBILE []BYTE frames:
  CHAN BOOL dark:
  CHAN INT luma, ctl.luma:
  PAR
    get.image (camera, frames!)
    avg.luma (frames?, luma!)
    is.dark (luma?, dark!)
    control.motors (dark?, motors, console)
:
```

Listing 3.5: *Top-level process definition for the occam-pi process-oriented implementation of the case study program*

- `is.dark` receives brightness values, using the first eight values it receives to set the calibrated threshold, then checking the brightness value against the threshold to generate a boolean as to whether the value exceeds the threshold.
- `control.motors` receives boolean values specifying whether the threshold has been reached, and stops the motors if it has. This process is also responsible for writing state to the console.

A process network diagram of the implementation is shown in figure 3.19; the program implements a 'pipeline' pattern between its components, refining from camera frame to brightness value to boolean threshold value. Three hardware interface channels are used by the program: `camera`, `console` and `motors`; these channels expose the same functionality as the calls to firmware functions in the C, but have the advantage of grouping together relevant functionality via protocols rather than having to rely on naming convention like the `uart0Send` prefixed console functions in the C.

3.6.3 Evaluation

Each test run of the program first required the robot to calibrate a brightness threshold value at the desired stopping point. The program was then allowed to loop ten times over the main control logic, driving forward and attempting to stop at the original point identified

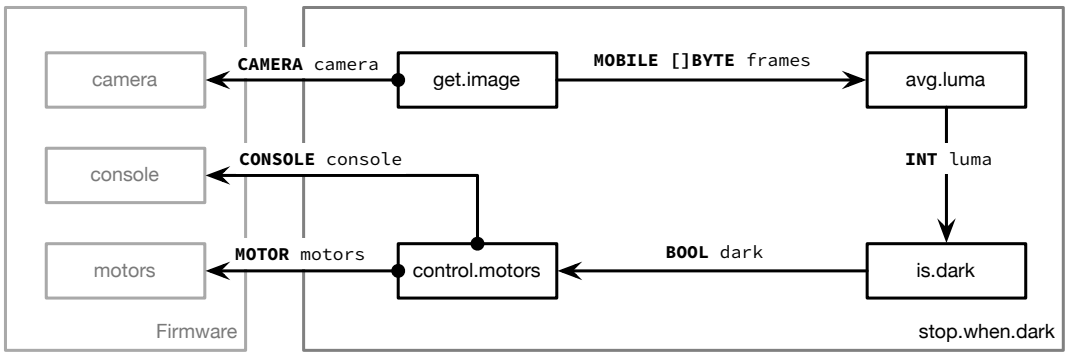


Figure 3.19: *Process network of the occam-pi case study robot program including its interaction with firmware processes.*

by brightness; the distance the robot travelled past the original point was recorded with each attempt. This distance encompasses both the robot's reaction time, to identify the condition in the environment and command the robot to stop, and the time taken by the physical deceleration of the robot; the physical deceleration should be constant between both implementations as the motors were driven at the same speed. The SRV-1 is rated at a speed of 20-40cm per second depending on the surface it is travelling on; in the case of the experiment a speed of around 20cm per second was observed.

Implementation	**Distance (cm)**
Process-oriented (occam-pi)	6.67 ±0.362
Imperative (C)	8.46 ±0.395

Table 3.1: *Average reaction and stopping distance of the robot (to 3 s.f., with 95% confidence interval)*

Three test runs were conducted with each implementation of the program the results of which are shown in table 3.1. Unexpectedly, the process-oriented implementation of the control program managed to stop more quickly (i.e. in less distance) than the imperative C implementation. The belief of the author is that this is due to more efficient implementation of the underlying hardware interface functions in the process-oriented firmware. These results show that the virtual machine runtime and interpreted hardware interface are comparable in performance to typical native code. In this case study the benefits of being able to use concurrency to structure the robot program were able to be employed, even in the absence of parallelism, to no significant detriment in the measured performance of the robot.

3.7 Conclusions

Further exploration of hybrid approaches, using multiple architectures in the homogenous process-oriented environment may yield useful combinations for solving particular problems. A library of components for building programs using different architectures would be greatly beneficial in the context of a visual robotics programming environment (such as that discussed in Chapter 4). Further extension work is presented in Section 6.1.

Multiple, parallel control paths through a behavioural robotics systems mean that individual components need less information about the environment and can be simpler — re-use of components is possible, as data can be routed appropriately through the network from sensor processes. Ideally it should be possible to add new behaviours to a robot without affecting the existing ones, whilst still re-using the elements of those behaviours that are relevant to the new behaviour. Achieving this using architectures like subsumption requires careful thought when decomposing behaviours into individual modules that carry out specific functions, as only modules that are sufficiently generic can be re-used. These decomposition skills are also practiced when writing process-oriented programs, the clearest expression being of a large number of individual components that each do only a single thing, composed to perform complex tasks.

Chapter 4

A Demonstrator Environment

This chapter presents a visual programming tool for process-oriented robot programming with occam-pi, designed to allow early diagram-only explorations of concurrent programming, and demonstrating the principles of process composition.

Diagrams are an important tool in the methodology of designing and implementing process-oriented programs. When teaching concurrency using process-oriented programming at the University of Kent students start with exercises where they reason conceptually with diagrams about connected networks of processes and communications between them. This approach encourages high-level thinking and reasoning about the problem decomposition and program architecture over language specific matters of syntax and run-time environment. Emphasising the continued use of diagrams to prototype and design concurrent programs is difficult, they are isolated from the program code they represent. Current practice means students either draw their diagrams on paper and discard them, or use a diagramming tool and update the diagram and program code in relation to each other.

Experienced process-oriented programmers use diagrams in both program design and refactoring, identifying and reasoning about the parallel decomposition and communication patterns of the system. The isolated environment of a process, with no side-effecting operations or global state, makes implementing the body straightforward once an interface to the rest of the program and purpose are defined (Section 2.4.1 further discusses this methodology).

In addition to application for process-oriented programming, diagrams have a history of being applied to the expression of robot control programs. Flowcharts in particular have been popular, as they are a commonly understood formalism for specifying human processes;

it is natural to mirror this onto an autonomous entity following processes to complete tasks. In educational robotics the LEGO Mindstorms RCX and NXT series of robots have gained significant popularity and both use visual programming languages to write control programs.

These established applications of visual programming to both process-oriented programming and robotics motivate the creation of a tool for building process-oriented robot control programs visually. While diagrams cannot capture the entire semantics of process-oriented programs, meaning textual programming is still required alongside diagram manipulation for many tasks, they are able to capture process instance creation, process composition, and channel creation. This subset of abilities is enough for explorations of building networks of predefined processes, such as the Braitenberg Vehicles previously introduced in Section 3.2. This process exercises the fundamental concepts of process-oriented design: process creation, channel creation and network definition.

The relationship between diagrams and textual program code is an important one; tools which do not have an equivalence between the visual environment and textual form of the program either require the programmer to consult both elements of the program or work entirely graphically (in the case of purely visual environments). The text-based programming model has knowledge, tool support, and flexibility built up over decades; it is reasonable to expect that all programmers will spend time working in and gain knowledge of a text-based environment. The strength of textual programming is in expressing sequential logic as a series of statements, conversely a weakness in visual programming models. Using textual programming for sequential code while moving high level component isolation and design to a visual diagram-based approach plays to the strengths of both models. The equivalence between the program structure in a process network diagram and the textual code to connect the network allows bidirectional reasoning about the model.

4.1 Visual Expression of Process-oriented Programs

Box and arrow diagrams are a lowest common denominator form in expressing the relationship between components, and use a very small and widely used set of visual elements. These diagrams can be found throughout Computer Science, from technical publications and office whiteboards to rough sketches on the back of napkins. Their informal, unstructured nature makes them a lightweight and popular way of exchanging ideas about software design. Slight specialisation of the box and arrow diagram yields the flowchart, a structured diagram for

expressing processes and algorithms that has its roots in mechanical engineering.

Goldstine and von Neumann pioneered the use of flowcharts for expressing the design of computer programs [GvN63], of which an example is shown in Figure 4.1. More advanced, domain specific diagramming languages such as UML have emerged which encapsulate more information about program code, to the point where they can be used as models for code generation and program analysis. Diagrams produced using these languages contain a significant amount of extra information but require understanding of additional visual formalisms — a disadvantage when considering their approachability and application in introductory contexts.

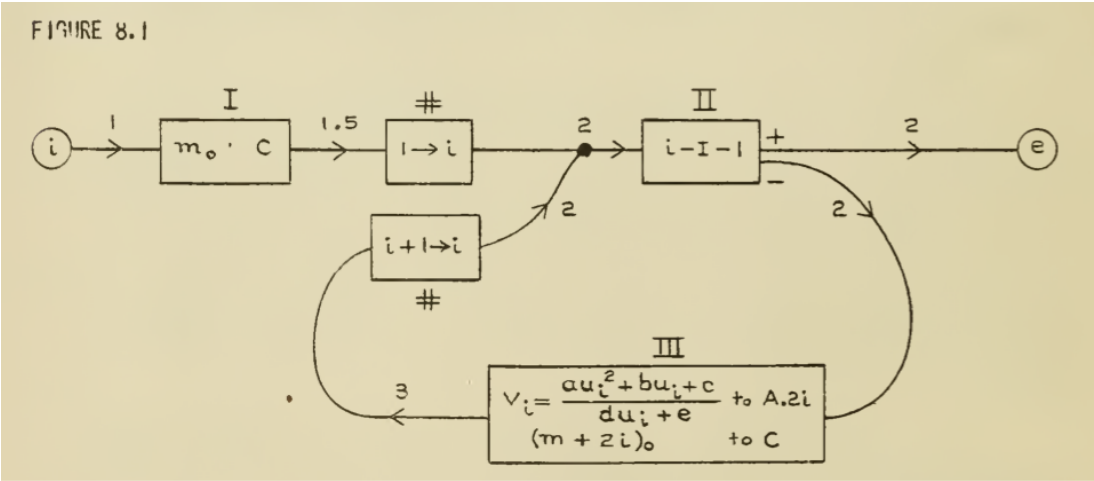


Figure 4.1: *An early flowchart expressing a computer program, from Goldstine and von Neumann [GvN63]*

Dataflow programming models, like process-oriented programming, are particularly suitable for the use of box and arrow diagrams as a design tool due to the isolation of components and the explicit flows of information between component boundaries. Boxes map directly to components or processes and the arrows to the directional exchange of data between those components. Data flow diagrams also have a harmony with the fundamental model of a robot, sensing the environment (input), using the input in computation (processing), and effecting state change in the world (output).

Process network diagrams add a minimal amount of extra annotation and semantics to the box and arrow diagrams to provide extra value in allowing clear communication of the design of a parallel program composed of processes and channels. A process network diagram shares properties of a data flow diagram, showing the communication path of data between components throughout the system. There are a number of extensions to the basic box and

arrow diagram format to express additional, advanced features beyond process and channel communications in process-oriented programs. There is no formal specification for process network diagrams, likely due to their close relationship to unformalised and unstructured box and arrow diagrams. Any commonality between diagrams has occurred organically, through the need for common communication and understanding. The visual representations for a relatively standard component like a barrier synchronisation is far more consistent than less commonly used occam-pi features like channel mobility, and process mobility. The subset of diagram representations used by introductory programmers are typically limited to processes and channels, the visual representations which are most mature.

4.1.1 Drawing Process-oriented Programs

There is no formal standard or publication defining the representations used in process network diagrams. Documenting community practice and identifying key features of each primitive's representation forms a basis from which to establish visual language design principles. Process network diagrams throughout the literature and even within a single research group of process-oriented programmers at the University of Kent exhibit a wide variety of diagram styles. These styles show influence by the diagramming tools (or lack thereof) used, personal preferences in representing specific components or design patterns and some degree of common practice. Sampson collects a number of styles encountered at the University of Kent and elsewhere in [Sam08], reproduced in Figure 4.2.

Patterns and trends in the use of diagram elements used to represent process-oriented primitives have emerged, which can be used to establish a generalised visual form for these diagrams. The variation between symbols used to represent a primitive increases as the usage of that primitive becomes less common — newer or less popular language features have less established consensus on their representation.

Choosing to implement a visual language based on common practices of diagram use is pragmatic. By building on the commonalities of existing diagrams, the visual language of the tool is immediately informed by several decades of visual communication of such systems in the literature. An alternate approach would involve creating a visual language from scratch, based on mental models of process-oriented programs and principles of cognition for diagrams. Blackwell and Engelhardt propose a taxonomy of diagram research which shows the breadth of research involved in such a task [BE02]. This approach would be an interesting area of future research and complimentary to a design approach based on user

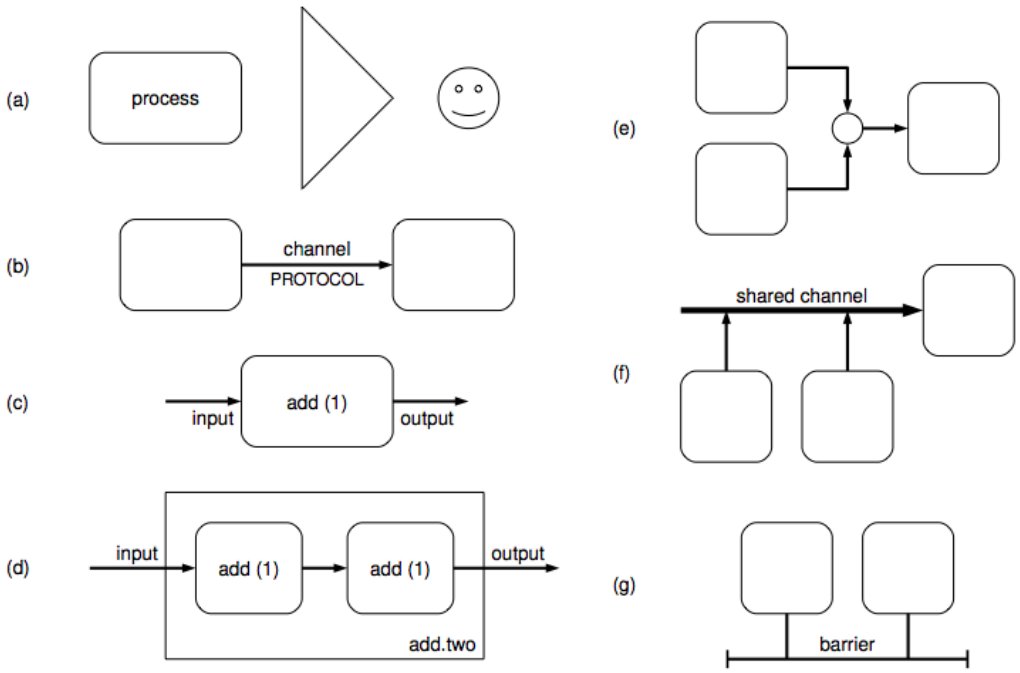


Figure 4.2: *A variety of process network diagram styles used in the Concurrency Research Group at the University of Kent and in the wider process-oriented programming community, from [Samo8]*

studies, both of which are detailed as potential extensions to this work, in Section 6.2.1.

Processes

Processes are typically represented by rectangular boxes, with a label stating the name of the process. Other shapes and symbols are sometimes used where conventions have been established. occam-pi course material used on the introductory course at the University of Kent typically uses triangles for `delta` (duplicate one input channel onto two outputs) and `plex` (multiplex two input channels onto a single output channel). The diagrams presented when discussing the subsumption architecture earlier in this thesis (Section 3.3.1) use circles for suppression and inhibition processes to echo the original diagrams the components are based on.

The amount of information captured about a process inside the box varies depending on the purpose of the diagram. Some high level diagrams may just contain the name of the process, although it is usual to include any parameter values supplied to processes for diagrams representing the design of programs. Where a process is a composition of other processes it

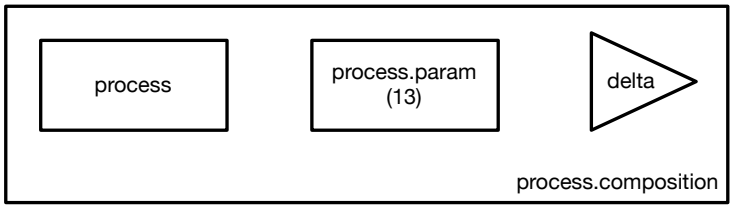


Figure 4.3: *Visual representations of processes*

is drawn as a containing box around the composed processes with its own name offset, as shown in Figure 4.3. This is also observed around entire programs when diagrams include the named 'top level' process and the interfaces to it provided by the run-time (equivalent to a `main` method in C or Java).

Channels

Channels are represented by directional arrows, drawn from writing end (where a process is connected to output to the channel) to reading end (where a process is connected to input from the channel) and may be labeled with the channel name, expected data type to be carried or both. There are two kinds of channels which have special visual representations: shared channels and channel bundles. Figure 4.4 shows a range of visual representations for channels.

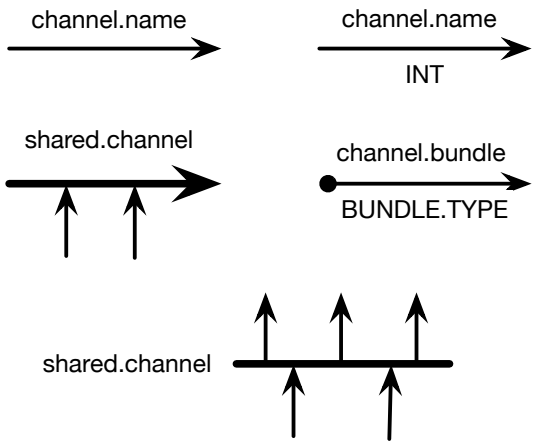


Figure 4.4: *Visual representations for channels*

Shared channels may have reading ends, writing ends or both shared; for shared channels where a single direction is shared the representation is typically a number of arrows in the

shared direction adjoining one larger, thicker arrow representing the single end. Where both ends are shared, a single thick line is used and both sets of arrows connect to the line.

Channel bundles are predefined collections of channels which can be instantiated as a group using a single type. Channels inside a channel bundle may go in both directions, and the bundle itself has a positive/negative end to define which way round the entire bundle connects. Channel bundles are typically drawn with the positive end as an arrowhead and the negative end as a filled sphere.

4.2 Existing Visual Programming Environments

As stated in the introduction to this chapter, both process-oriented programming and robotics have been the subject of previous work and interest in use of visual programming. Surveying the existing and historical environments available for robotics and process-oriented programming allows the identification of strengths and weaknesses of existing tools, such as visualisation methods and any particular features which may be desirable in a demonstrator tool.

4.2.1 Visual Programming for Robotics

The application of visual languages to robotic control was given significant attention by academia during the 1996 and 1997 competitions at the *Symposium on Visual Languages*, promoting the use of visual languages in robotic control. Efforts leading directly from the contest led to the development of experimental visual languages for robotics. Commercial factors have also driven the development of visual programming environments. The LEGO Mindstorms RCX and NXT series of robots has had a significant influence in promoting visual programming languages for robot control, having included several visual languages as the only supplied environments. The Mindstorms RCX was supplied to education with two different programming environments, RCX Code (based on the Logo Blocks model, discussed in Section 4.2.1) and RoboLab (based on National Instruments' LabVIEW, discussed in 4.2.1). Microsoft have entered the robotics environment market with a visual language based Robotics Developer Studio (RDS) which was introduced to provide a control solution for the growing number of mobile robots.

Whilst some of the tools covered in this section are outdated, specific to robots other than

the ones used in this work or unavailable for current use their design and feature set serve to inform the design of the demonstrator tool.

LEGOsheets

An early, rule-based environment called LEGOsheets allowed users to build a representation of a LEGO robot and configure its behaviour based on simulated "cables" connected to virtual sensors with user-supplied values [GIL+95]. LEGOsheets was developed on top of the general-purpose Agentsheets framework designed to simplify the development of grid-based agent environments, with components written in Common Lisp [RS95]. The use of external connections into the development environment to receive values from virtualised sensors and simulate actuation of outputs at development time allows for a single environment for program development and testing, potentially very useful in the classroom with a simulated robot environment.

Visual Behaviour-based Language (VBBL)

Cox et al. [CRS98] made use of a visual object-oriented data-flow language called ProGraph to develop Visual Behaviour-based Language (VBBL), a rule-based visual language making use of finite state machines (FSMs). These FSMs are defined with each state being a set of behaviours; the system uses conditions and message flows as events which control the transition between states. The underlying ProGraph language facilitates the use of message passing between components, but has a disadvantage in that all components are programmed entirely graphically. There are two different visual languages used: a flowchart based language used for sequential logic in method implementation and a graph-style diagram used for expressing data flows between components.

Improving on VBBL, Cox et al. proposed a visual programming environment based on representations of the actual robot itself, similar to LEGOsheets, noting that VBBL could be improved by leveraging "the obvious visual representations" of objects instead of focusing on the visualisation of abstract control concepts [CS98].

A second, improved environment allowed for the creation of robotic control programs through direct manipulation of a user-specified simulated robot within a defined environment, prompting for the specification of behaviour when unknown combinations of sensor input were encountered at runtime. This approach reduced complexity and avoided the manual

user creation of finite state machines. Having separate behaviours allowed for the concurrent execution of each within a subsumption architecture, albeit one missing the ability to prioritise behaviours over one another. Requiring the full specification of the robot's environment limits the utility of the model for problems outside simulation, as the physical world represents an inherently unknown environment. The hardware definition module (HDM) model defines a robotics platform as a programming target by composing classes of objects into a graphical and functional representation of the robot and its abilities.

The concept of hardware definition modules maps directly into our approaches with occam-pi robotics, using processes to represent hardware on the robotics platform which can be connected into process networks as required, providing a coherent model for specifying interfaces to and configuration of hardware from within the program. This model is particularly useful on reconfigurable robot platforms such as the LEGO Mindstorms RCX or NXT and its application is discussed in Section 3.1.5.

Logo Blocks

Logo Blocks [Beg96] was a visual language based on Logo and designed for use with the antecedent to the Mindstorms RCX; the Programmable Brick, designed by Resnick et al. [RMSS96]. A lineage of visual programming tools can be traced back to Logo Blocks' visual model of sequential program construction through interlocking puzzle pieces; these puzzle pieces fit into each other to build up statements from individual pieces of program grammar. Figure 4.5 shows Logo Blocks with a program being constructed.

The LEGO Mindstorms RCX was supplied at retail with the *RCX Code* environment, which used a visualisation based on that of Logo Blocks, constructing sequential code using components which slotted together like puzzle pieces. The history of the Mindstorms RCX and the programmable brick are described further in Section 2.5.

Scratch [MBK+04] and StarLogo TNG [BK07] use virtually identical visual programming models to Logo Blocks. Scratch is an environment focused on enabling children to create programs which use media, networking and rich interaction, building "animated stories, games and interactive art" [MBK+04]. StarLogo TNG is a graphical programming environment which lowers the learning curve to creating 3D games, allowing this area to be used as motivation with schoolchildren learning to program [BK07]. The principles behind the Logo Blocks visual model have been wrapped into a Java library called Open Blocks allowing their wider use. Open Blocks wraps the Logo Blocks visual programming model into a

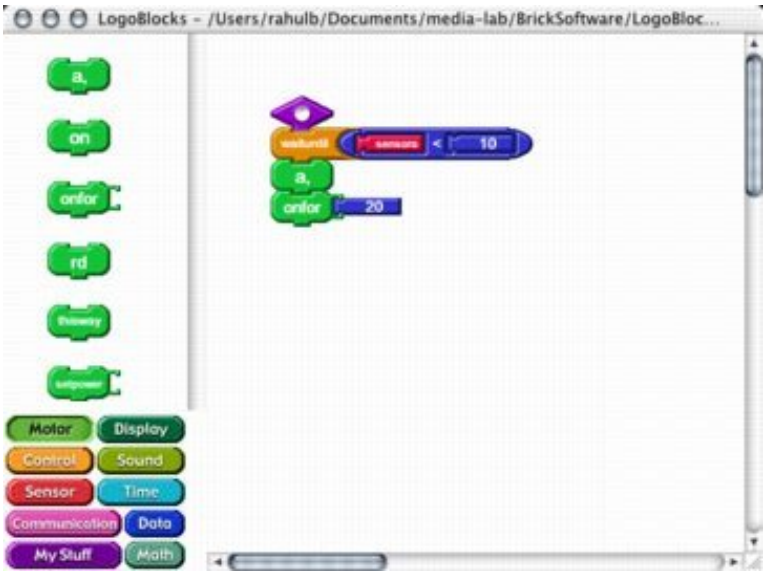


Figure 4.5: *The Logo Blocks environment with a program constructed, from [MIT02]*

general purpose Java library for use by application developers to provide other programming language implementations [Roq07]. This library has been employed in Google's App Inventor for Android, now maintained as the MIT App Inventor, a tool for visually constructing applications for mobile platforms designed to engage students [Abe09].

The Logo Blocks visual environment, while relevant due to its applications with the RCX and in Pedagogy focuses on a different problem to the one the demonstrator tool seeks to solve. Logo Blocks style visualisations address the creation of sequential code, while the problem area of the demonstrator tool is high level program structuring and purely program composition. The structuring techniques presented for sequential code are of tangential interest, as in combination with high level program structuring they would permit an entirely visual approach to process-oriented programming; this potential for further extension of the work is discussed in Section 6.2.3.

Mindstorms and LabVIEW

The LEGO Mindstorms RCX was supplied to educators with a piece of visual programming software called *RoboLab*. RoboLab is a package of library routines and customised interfaces for National Instruments' LabVIEW [ECR00]. LabVIEW is a visual programming environment originally designed for instrument control and data collection in laboratories. RoboLab presents a palette of possible actions for the robot to perform, which are connected

together by the user on a canvas to indicate their execution sequence. A number of additional components allow the modification of execution flow, including looping structures and conditionals.

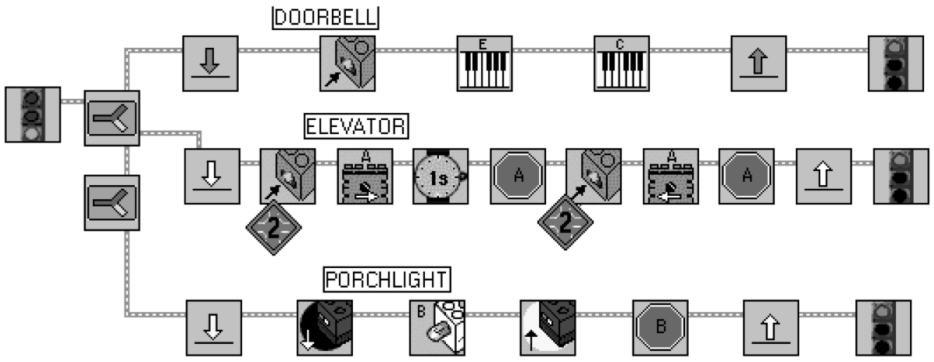


Figure 4.6: *A control program designed in the LabView-based RoboLab environment, from [ECR00]*

Barnes has created a set of materials designed for UK Key Stage 3 (11–14 year old) schoolchildren to pick up programming the Mindstorms RCX using RoboLab's visual paradigm and effective use of the variety of sensing, actuation, program control structures and conditionals blocks within a few hours [Bar05]. These materials are a significant inspiration for the development of a visual environment for process-oriented programming, as the same graphical compositions and methodology could be applied to process networks given an appropriate tool and visual toolbox of processes.

The LEGO Group continue to supply a visual language with the newer Mindstorms NXT, again based on LabVIEW and called NXT-G. The NXT-G language borrows metaphors from LEGO blocks in terms of visual layout, and is changed from RoboLab in significantly emphasising data-flow over wires between components. The full LabVIEW product is not specifically a robotics environment but can it be used for robotics, and sets of components are supplied to allow its use with the Mindstorms NXT.

LabVIEW's graphical control language, *G* has shown potential for the design of robotic control systems outside of small, educational robotics platforms, having been used by the Virginia Tech Team in the 2007 DARPA Urban Challenge to claim third prize [RAA+07]. LabVIEW's underlying runtime supports concurrency and the language is inherently concurrent, allowing the construction of programs with multiple flows of control. These environments are restricted to purely visual representations of programs; removing the speed and flexibility of expression that comes with textual programming. There is no underlying model

to LabVIEW's concurrency and control over it is taken away from the programmer — the runtime may run individual components in different threads at its choice, but the semantics of concurrency are not defined in the visual language.

Microsoft Robotics Studio

Another recent development in commercial visual programming environments for robotics is the Microsoft Visual Programming Language (MSVPL), included only with Microsoft Robotics Studio (MSRS) [Mico8]. This visual language is much like NXT-G in being a robotics focused data-flow language consisting of both primitives for looping and variable creation, alongside more complex components acting as services.

MSRS allows users to connect the data flow path between a number of concurrently executing, potentially network distributed services. These distributed services run via the Decentralised Software Service (DSS) and are co-ordinated within the Concurrency and Co-ordination Runtime (CCR). The CCR uses asynchronous communication between components to allow the design of parallel systems. Programming DSS components is a challenging task and requires knowledge of distributed service-oriented programming — components communicate using HTTP and a SOAP-based protocol — making the creation of custom systems difficult. These components are defined in C#, outside the scope of the visual language, although the need to program DSS components is avoided for common educational robotics platforms, as components are built into the environment.

The relevant component of MSRS, the MSVPL reduces the programming task to connecting combinations of these predefined C# components together visually. MSVPL can be used in this way to program several of the robots used in Chapter 3, including the Surveyor SRV-1 and Pioneer 3-DX.

The toolbox and component canvas model used in the MSVPL is very similar to that proposed for POPed, especially the provisioning of sets of robotics platform-specific components. It should be noted that there is no underlying textual representation for programs written using this visual environment, a constraint which MSVPL shares with LabVIEW. Lacking the ability to write sequential code textually, the visual paradigm in these tools must contain all primitive actions which can be completed. As a result of this constraint, MVPL has *Data* blocks which input values specified by the programmer to named *Variable* blocks, a clunky workaround for assignment, as shown in Figure 4.7. The entire MSRS environment is restricted to use on Microsoft Windows, a limitation in a multiple operating system environment.

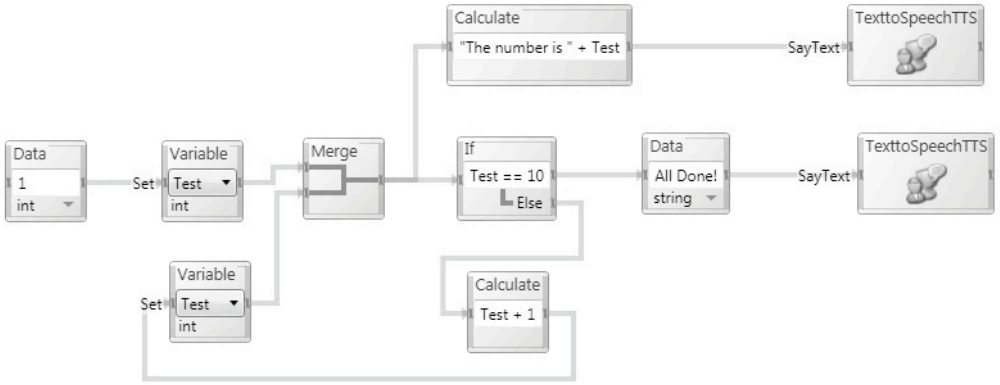


Figure 4.7: *A sample program in the Microsoft Visual Programming language, showing its visual representation for variable assignment, conditional and hardware interface, from [Mico8]*

4.2.2 Visual Process-oriented Programming

Historically a number of visual occam programming and program visualisation tools have been created; where the former is used for creating new programs visually, and the latter for visualising the execution of programs written textually. It is also possible to find a number of other tools for other varieties of process-oriented programming, including CSP-based approaches. Of the occam and occam-pi tools in the literature, with the exception of Sampson's LOVE, very few are in a state in which they can be viably used today. In the case of the occam-based tools, the majority of development and activity on visual environments and programming for occam occurred during the height of the success of the Transputer, predating the modern occam-pi-based desktop era applications of the language. The historical relationship of occam to the Transputer is discussed in Section 2.2. These environments depend on features built into the physical hardware of the Transputer or capabilities of commercial software developed by Inmos to support programming of Transputer hardware from the commodity workstations of the period. Tools designed for use with networks of Transputers commonly have features present to support the implementation and optimisation of occam programs on this specialised parallel hardware which have no equivalent in the desktop toolchains used for occam-pi today. Elements such as visual layout of software processes across processors in a Transputer network and the instrumentation of programs to provide processor and inter-processor link utilisation data provide suggestions on tooling which could be created to enhance the state of process-oriented programming in occam-pi. These opportunities are discussed further as extension work in Section 6.2.5.

GRAIL

Stepney's GRAIL offered a visual representation of the parallel and sequential structure of occam programs, using the hierarchical structure of the language to manage large programs by "folding" (hiding) sections of the program [Ste87, Ste89]. GRAIL allowed for the additional display of channel information, but the representation used bears little resemblance to those used for process networks and is tied to the hierarchical display of parallel and sequential occam code. No editing capability was provided within GRAIL, but its representation of process internals could be developed into a visual language for the creation of sequential occam code.

Visputer and Millipede

A number of graphical tools have focused on the design of distributed programs for execution across multiple Transputers, using the visual elements to allow programmers to specify the distribution of the processes comprising the program across the processors.

Visputer by Zhang and Marwaha provided a complete set of graphical tools for program composition, processor allocation, performance monitoring and debugging [ZM95]. Visputer provided the option for nested processes to be expressed in a visual language, with low-level logic provided textually. Millipede by Aspnäs et al. provided visual process layout integrated along with performance monitoring of processes and communication links [ABL92].

Millipede used processes and channel ends as its primitives in a *palette* of graphical components, providing a way to place channels between processes, configuring the network and specifying process interfaces. As with Visputer, Millipede made use of a text-based editor for specifying process logic and allowed the graphical expression of *compound* process networks containing nested sub-networks of processes.

TRAPPER

TRAPPER by Scheidler et al. offered a graphical programming environment comprising four separate component tools: Designtool, Configtool, Vistool and Perftool [SSKF95]. The entire TRAPPER environment covers a similar scope to Visputer, but its separation allows us to focus on Designtool, the most relevant component of the four to visual programming. TRAPPER allowed for the reduction of large process networks into a hierarchy of sub-

networks which can be collapsed, an essential feature for managing complexity. The use of connection points at the edge of process network diagrams to represent connections to the outside world in Designtool is novel and serves well to highlight the interface between program and hardware interfaces, as shown in Figure 4.8. The remaining three components of the TRAPPER environment relate to management of tasks involving the configuration, instrumentation and optimisation of programs running on actual Transputers, and as such are of limited relevance to our objectives at this point.

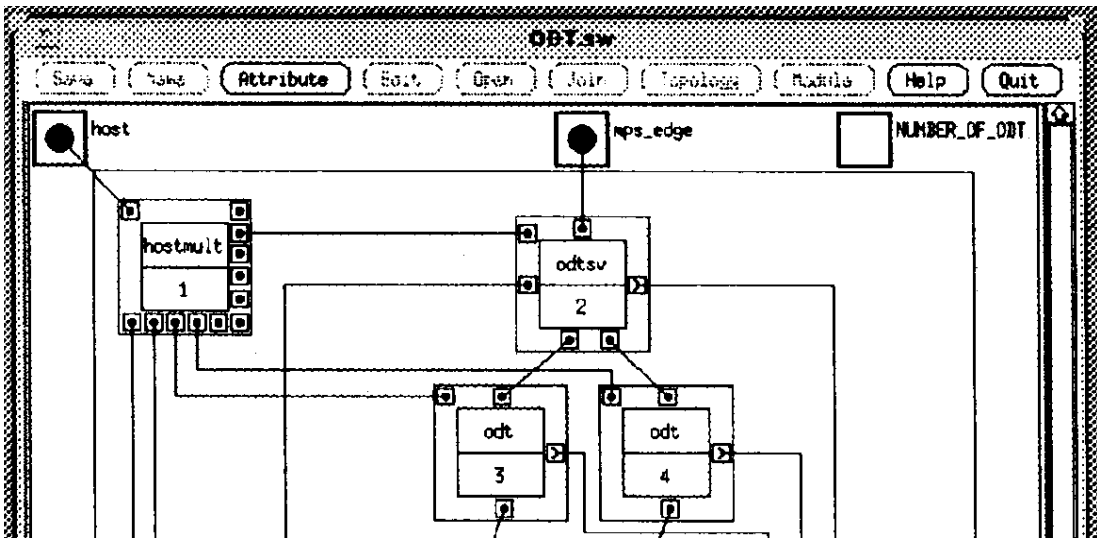


Figure 4.8: *TRAPPER's Designtool showing its connection points outside of the canvas for interfacing to external components, from [SSKF95]*

The Strict occam Design Tool

The Strict occam Design Tool (SODT) by Beckett and Welch allowed for process networks to be designed graphically using common paradigms of occam programming [BW96]. The tool was specifically aimed at the creation of deadlock-free systems and made use of *interfaces*, which defined roles for processes which specified their communication behaviour. These interfaces were used in SODT to enforce the Client/Server, IO-SEQ and IO-PAR parallel design patterns [WJW93], ensuring that networks designed with SODT were deadlock-free and that components were composed in these patterns which have been proven correct. Visual representations for various complex capabilities of the occam programming language were described as future expansions to SODT. These expansions are of interest when designing a visual environment capable of manipulating more complex process architectures, especially

those using features such as replication to create pipelines, rings and grids of processes.

gCSP

The gCSP tool by Hilderink and Broenink et. al. [BJ04] offers a visual environment for designing concurrent programs using a graphical modelling language based on CSP [Hil02]. gCSP is written in Java, allowing it to run across major operating system platforms. Both data-flow and the concurrency of the program along with a hierarchical view of the program's structure are presented to the user, more thoroughly discussed for the design of user programs in [VHB+00]. The ability to provide an information-rich outlined structure of a process-oriented program, whilst being beyond the scale of our current aims for an introductory tool, is a potentially desirable feature for developing more complex programs.

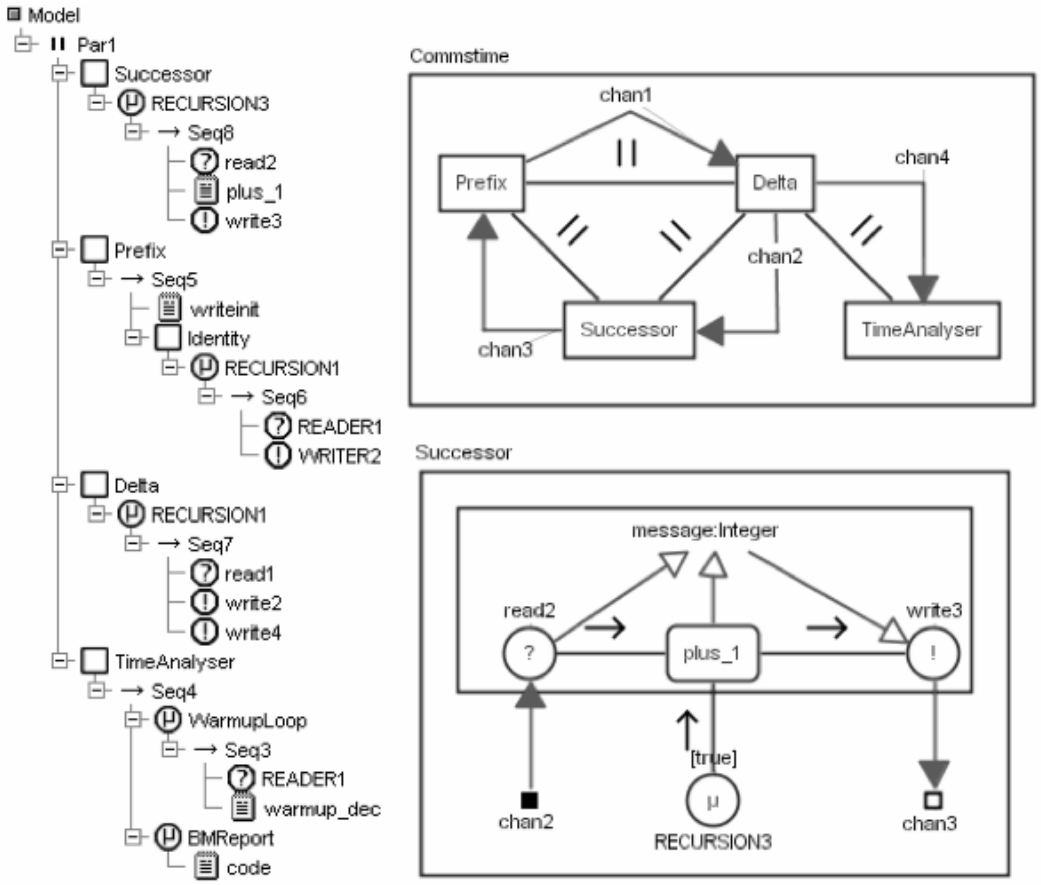


Figure 4.9: *A gCSP session showing sequential and parallel composition of processes, along with the hierarchical browser, from [BGL05]*

gCSP has multiple code generators which allow it to output C++, occam or CSPm from the user-facing graphical representation of the program, essentially making the graphical representation an intermediary between many different process-oriented languages [BGL05]. The model of code generation from connected components is similar to the desired functionality of POPed, removing the need for of syntax. The visual language used in gCSP is designed to fully express programs graphically and has stronger ties to CSP than occam.

However, gCSP's primary representation is its diagram format — the code generation is merely provided as a way to run these programs under different process-oriented runtime environments. gCSP is intended for building CSP systems and generating implementations, rather than allowing construction of programs in the implementation languages. The generality of gCSP is problematic for its use as an introductory tool for occam-pi programmers, as its syntax is fairly obtuse and there is a significant learning curve.

GATOR

GATOR (Graphical Analysis Tool for occam Resources) by Slowe and Tanner aimed to provide a debugging aid for occam programs and to progress towards a graphical environment for the program creation [ST04]. The user interface of GATOR is shown in Figure 4.10. GATOR was able to provide extremely rudimentary visualisation of connections between processes and data communicated by pre-processing source files. This processes parses the source for names of processes, names of channels and their connectivity; the source file is then transformed to include special tapped channels which report on their activity back the GUI tool via a TCP/IP port at run-time.

The visualisation of the process network is done via a spring-embedded graph, a primitive approach designed purely to ensure that the labels on the processes were readable and channels were clearly visible; this was only moderately successful as can be seen in Figure 4.10. GATOR highlights a significant problem in this regard: when visualising executing process-oriented programs, how to lay out the processes created in a way that reflects their relationship to each other and makes the program intelligible. Programmers are used to process network diagrams, where a lot of these layout decisions are made aesthetically and informally — delivering a similarly pleasant result automatically is very difficult.

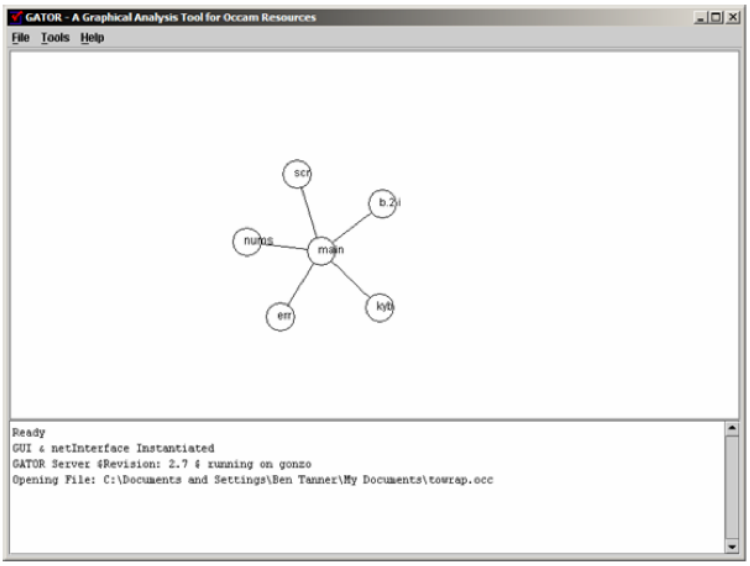


Figure 4.10: *The main window of GATOR, from [ST04]*

LOVE

The Live occam Visual Environment or LOVE by Sampson [Sam06] is a graphical environment for creating networks of synthesiser components for audio creation. LOVE's combination of process-oriented programming and audio builds on an existing relationship between visual dataflow tools and synthesis. The open source PureData graphical language [Puc96] and its progenitor, the MAX graphical language by Puckette et. al. [Puc91] build networks of *objects* which receive input, generate output or both. As its name suggests, LOVE focuses on a live approach to programming, allowing synthesiser components to be plugged and unplugged while the program is running; LOVE uses standard occam-pi processes and channels, with a small wrapper around the processes to allow them to be plugged and unplugged from networks.

LOVE shares scope with our desired tool, in presenting a predefined set of components for the user to connect together on a canvas on which they can be arranged. LOVE is a process-oriented program written in occam-pi; it draws a custom, vector based user interface; components in LOVE have representations based on their function which allow them to be interacted with, as shown in Figure 4.11. Due to this custom representation and user interface; LOVE cannot be repurposed as a tool for ordinary composition of processes.

LOVE uses graphical selection of channel ends to aid in making connections, with visual cues for correct input connections when an output is selected. This is a desirable feature in a

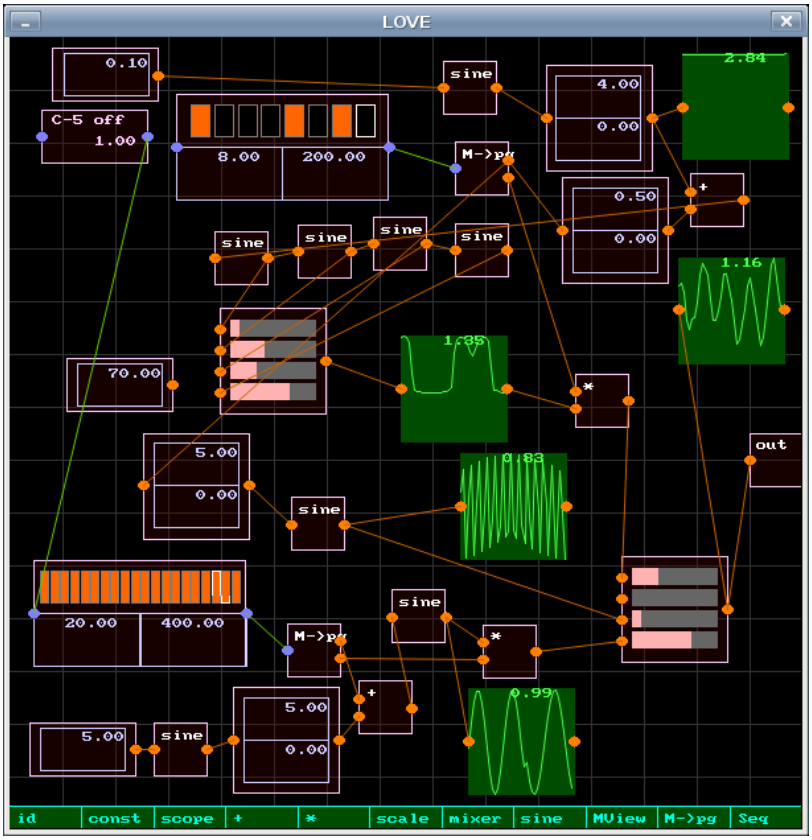


Figure 4.11: *The Live occam Visual Environment, LOVE, from [Samo6]*

visual editor, simultaneously enforcing type rules and easing the process of creating networks for the user. It has no structuring tools for larger networks, allowing the users to see the entire network at once, but restricting the size of networks that can be created; Sampson notes in future work for the tool that it would be ideal to allow the user to indicate groups of components to be re-used as a larger component.

POPExplorer

POPExplorer by Jacobsen takes a different approach to manipulating process networks as it interfaces with the Transterpreter virtual machine runtime directly, modifying internal runtime state to manipulate process networks [Jaco6]. Processes are selected from a list in the tool, and their source code is shown. When the user clicks a 'create' button, the code is compiled and the resulting bytecode loaded into the VM. The execution state of individual processes can be controlled independently Channels can be connected and disconnected whilst processes are running, and the communication state and type of a given channel end is displayed. The extent of POPExplorer's modification of runtime state introduces conditions that occam-pi programs would not expect to encounter; the communication model of occam-pi channels assumes that messages will not disappear 'in flight'.

4.2.3 Summary of Features

A number of desirable features for a combined robotics and visual process-oriented programming environment can be identified from the environments surveyed.

The use of a toolbox of components and a drag and drop canvas in which the user manually places processes is found across several tools: LegoBlocks, RoboLab, MSRS and LOVE. Of the robotics environments, RoboLab groups components by control structure, with separate 'toolboxes' for waiting for events and looping while MSRS separates out basic network routing and control structures 'Basic Activities' from a filterable list of all other components. The process-oriented programming environments surveyed do not generally contain the same segregation, with LOVE not distinguishing between generic and domain-specific processes and POPExplorer using a flat list of available processes. Grouping of components by function may be applied to provide a separation between the processes relevant to a specific hardware platform or architectural approach.

Most of the tools surveyed have an area of the interface where the components that are part

of the program appear when part of the active program. In tools where the visual language reflects textual program code, such as LogoBlocks and Scratch, the layout of components on the screen is tightly defined by where the component is syntactically valid. In these tools the location of a specific component has semantic meaning about what order the component will execute in, or which operation it will be a part of.

When working with network graphs, as in a number of the process-oriented tools, the location of specific components in the visualisation has no semantic meaning; the programmer may commonly choose to arrange the components for aesthetic routing of channel connections or to reflect functional groupings of processes. GATOR is unusual in this regard, as it applies an auto-layout algorithm to automatically layout the currently visible set of processes; this results in process network diagrams that do not resemble those typically drawn by hand. Tools which build data or control flow networks, including RoboLab and MSRS tend to result in programs which spread horizontally, with the convention being for input to arrive at the left of a component and output to depart from the right.

Techniques for the management of complexity in visual programs are not present in all tools surveyed; this limits their applicability to generating larger programs – as the user runs out of space for components, portions of the program become hidden from view.

Process-oriented programming allows for subnetworks of components to be encapsulated inside a process. Two of the process-oriented programming tools surveyed, Grail and Visputer, allowed these encapsulation processes to be expanded and contracted, facilitating working with larger scale programs. The ability to group subnetworks of processes into compound networks on the fly is desirable, as often a single behaviour or function is composed of many processes. The visual representation chosen for processes also has an effect; the larger the visual representation of a process is, the less of them may appear in the diagram at any given time if all processes are visible.

The provision of a user interface to examine and set the parameters to a component is common to many of the visual tools surveyed. Some, like MSRS and LOVE, allow this manipulation directly on the component body itself, clearly showing the current parameter value on the representation. This has advantages of clarity – the functionality of a network of processes depends on both its topology and the parameter values given to the components therein and both are clearly visible. The disadvantage to this direct interaction is in increased size of components, requiring additional space for the interface to set parameter values above and beyond channel connection points and the name of the process. This increased size of representation ties to issues discussed earlier around managing complexity as networks grow

larger.

The provision of a visual syntax for expressing sequential logic divides a number of the tools examined. RoboLab, MSRS and LegoSheets allow values and operators to be expressed as visual elements, while Scratch is designed around replicating the structure of textual code with visual components. Of the process-oriented tools, GRAIL and gCSP both employ two views of the program, a graph structure for high level design and a separate visual syntax for expressing sequential logic. Expressing sequential logic is a strength of textual program code. Removing the need for the programmer to write sequential logic through the provision of a comprehensive set of existing components tailored to the task at hand is possible given a limited scope; LOVE and RoboLab achieve this due to specific focus on a specific task and platform respectively. Aiming to avoid the need for the programmer to write sequential logic and designing a purely compositional tool makes provision of flexible, task and platform specific components critical.

4.3 Design

Formulating the design of the demonstration tool in isolation of its implementation separates practical concerns from the expression of an ideal environment. The demonstrator environment should support the creation of basic data flow example programs using pure process composition, using Welch's *LegoLand* components. The LegoLand components are a set of processes designed for use in introducing students to process-oriented programming. Students use the components to create programs which pass a stream of integers, providing an introduction to process network composition and the communication patterns required in process-oriented program design.

The environment should also facilitate the creation of behavioural robotics programs for robot platforms. To this end, the environment contains groups of hardware interface processes specifically for use with the LEGO Mindstorms RCX (presented in Section 3.1.5) and Surveyor SRV-1 robotics platforms (presented in Section 3.1.6).

Based on the elements surveyed in Section 4.2.3, the design of the visual environment focuses around three elements: a Toolbox, a Canvas and an Information Panel. A mock-up containing these elements is shown in Figure 4.12. To compose programs in the tool, users choose processes to add to the program from the toolbox list on the left side of the screen and double-click or drag the entry onto the canvas to the right to create a new instance of

the process. When browsing the toolbox list, metadata about the process is shown in the information panel to the bottom right including the purpose of the process and expected input and output. The toolbox list has some hierarchy, allowing for groups of processes appropriate to a task or given hardware platform to be visible and those not relevant to the task at hand to be hidden.

Channel connections are made using channel connection points on the processes themselves; convention determines that inputs are on the left side of the process and outputs are on the right side of the process. To connect a channel, the user selects two channel connection points in succession; if the selection contains both input and output connections with the same type, the channel is created. If the selection contains two connection points of the same direction, or of mismatched types a message is presented in the information panel explaining the problem with the selection. The user is aided in selecting the correct connection points as once the first selection is made, the unconnected points left in the network of the correct type and direction are highlighted.

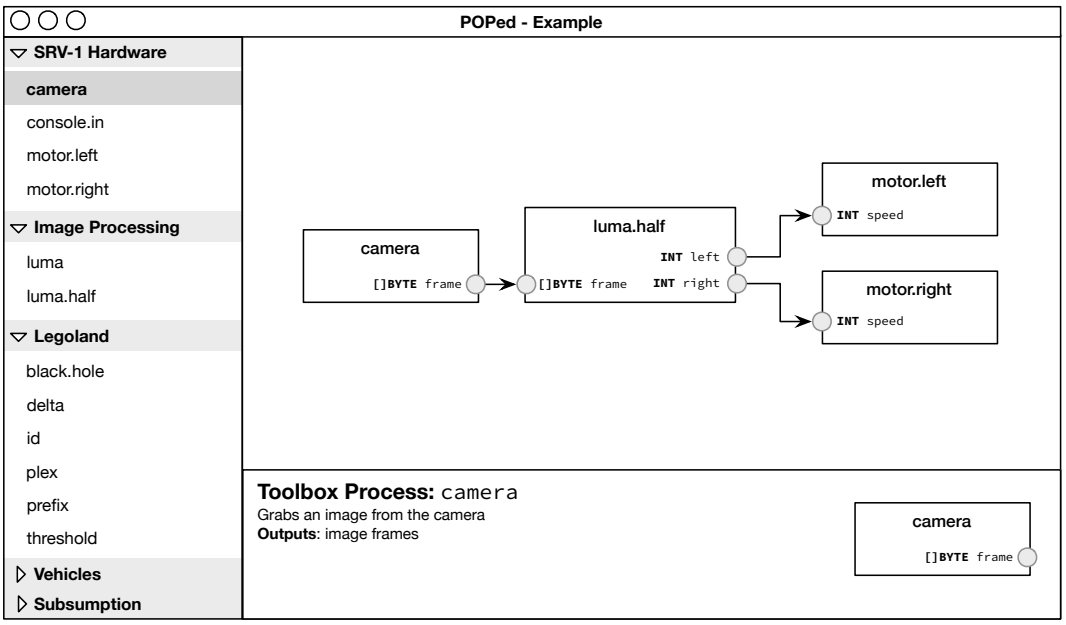


Figure 4.12: *A mock-up of the POPed user interface*

When the user wishes to run the program the demonstrator tool checks that the process network is fully connected, to verify that it forms a valid program, and writes an occam-pi source file containing the processes and topology of the network. The source file is created by combining the code for each of the processes used and generating a top level process

containing all of the processes and channel definitions specified graphically by the user. This generation avoids an entire category of potential programming errors which can occur in the wire-up of process networks, leaving channel ends unconnected or forgetting to create specific channels. Once the source file has been generated, execution of the program requires invocation of the occam-pi toolchain to compile the source to a bytecode file and in the case of robot platforms, the use of a tool to transfer the bytecode from the host computer running the environment to the robot itself.

Toolbox

The *toolbox* is located on the left-hand side of the window and contains a grouped list of the processes available in the environment. When a process is selected in the toolbox list, information about its inputs, outputs and parameters is displayed in the information panel at the bottom of the screen along with a description to aid the user in understanding the purpose of the process. The information panel is fully discussed in Section 4.3. Processes within the toolbox are logically grouped together to allow the programmer to hide sets of processes which are not currently relevant to the task at hand. To make use of components to build a program, the programmer can drag from the toolbox onto the canvas or double click the entry in the list, at which point a instance of the process in the toolbox appears and can be freely positioned on the canvas.

Toolbox processes can be specified in generic terms, despite the lack of generics in occam-pi, allowing processes to be parameterised by a type. POPed uses a template with placeholders for the generic types to generate code, substituting concrete types into the templating and ensuring that occam-pi's strict typing rules are met. This templating feature is essential to avoid the toolbox being filled with variants of basic network routing components, such as `delta` and `plex`, which vary only in the types declared on their interface and temporary local variables. The use of templating to emulate generics is fully detailed, along with an example of the process, in Section 4.4.6.

The ability to use generics means that processes such as `delta` can be specified , and the tool can generate a specialisation of the delta for basic types such as `INT`, or `BOOL` depending on the connections made. It is necessary to place a constraint such that once one of the channel ends on a process using generics is connected to, that the generic type is set for all channel ends. For example, if a `delta` process were placed on the canvas and had its input connected to a channel of `INT` from another process, its outputs would at that point be become of type

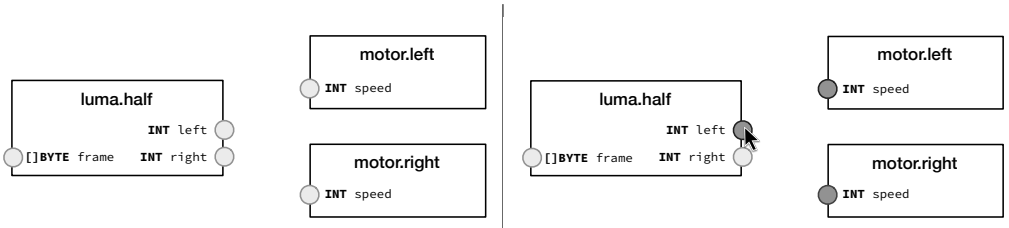
`CHAN INT`. This approach to generating code instead of real generics is limited in terms of its use as channels using `PROTOCOL`s cannot be handled as a standard datatype; `PROTOCOL`s require special handling via `CASE` statements to handle the variant present in the message. As hardware interfaces on the Surveyor SRV-1 make extensive use of `PROTOCOL`s, this will need to be improved to avoid the need for additional interface processes to simplify the protocols to basic types.

The processes visible in the toolbox reside in the filesystem in a hierarchical structure matching the toolbox. The set of processes present in the tool may be extended at any time by adding additional directories (for new groups) and process 'block' files inside those directories, as specified in Section 4.4.3.

Process Canvas

The process canvas is the central focus of the tool, being the area in which the user builds their program. Channel ends are represented by *connection points* on the process, allowing the user to easily connect the processes together. The user selects a connection point and the potential points to which a connection can be made are highlighted. This highlight refines the user's choice to just the valid set of connections, giving a visual representation of both type-compatibility and direction of the unconnected connection points in the network. This selection mechanism is shown in Figures 4.13(a) and 4.13(b); once a connection has been made, a directed arrow is placed between the two connection points. On attempting to compile a program which is not fully connected with channels between all connection points on the canvas, an error is displayed in the information panel informing the user of the processes which are not properly connected, and the offending channel connection points are highlighted.

Layout of the process network is managed entirely by the user, capturing the way paper or diagramming tools are typically used in the first few weeks of parallel program design. Processes on the canvas are able to have their program code inspected, allowing the user to gain an insight into the code behind the diagram. Ensuring the underlying code is not hidden is important as allowing the programmer to reason about the relationship between the diagrams being manipulated and the underlying occam-pi program code composing the components and network is a significant pedagogic element. There is a constructivist element at work here as the user creates the program from the high level components and is able to gain knowledge and reason about the internals of processes in the context of the



(a) Possible connection points, no selection made

(b) Connection point selected, possible connections highlighted

Figure 4.13: *Connection points and their type highlighting mechanism*

overall program and its function.

This functionality provides a starting point from which to include editing of existing processes, although editing is outside the scope of the prototype. By assuming that all processes present in the environment are syntactically valid and that all composition code is generated the tool is able to avoid a significant amount of work in designing appropriate error handling and feedback to let programmers debug and identify syntax errors in a combination of generated and user created code. As noted in Section 4.3.1, while this avoidance simplifies the handling of program code dramatically it also limits the applicability of the environment to general purpose process-oriented programming.

Given that POPed is a demonstrator tool, programmers using it will move on to create programs textually in the occam-pi language — the aim of the environment is to allow mental models to form and investigation of the programming model based on initial visual experiences of composition and network design.

Information Panel

The information panel is located at the bottom of the screen and allows contextual information to be provided to the student about processes selected from the toolbox and canvas. An example of the information displayed when a toolbox process is selected is shown in Figure 4.12, while an example of a selected canvas process is shown in 4.14.

This presentation of additional textual information is intended to allow the programmer to fully understand in isolation the components that make up the program, along with the relationship between the connected processes. Information may be inferred from the connections between processes to present the user with textual descriptions of the inputs

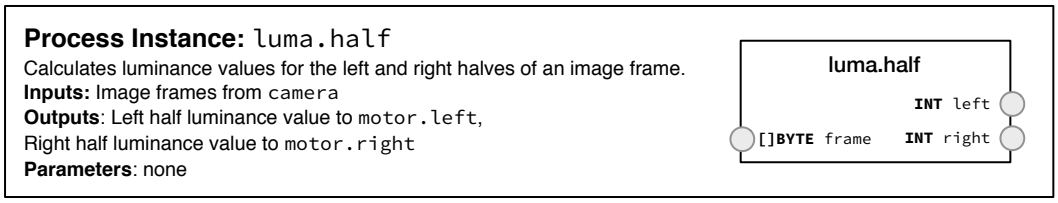


Figure 4.14: *The information panel with a process instance selected on the canvas*

and outputs from a process instance. These descriptions provide a simplified explanation of the component's operation within the system, given well named processes and described types. An example of these explanations for a `luma.half` process is shown in 4.14, where a `camera` process has been connected to input and two motor control processes (`motor.left` and `motor.right`) are connected to the outputs.

4.3.1 Limitations of the Visual Environment

By restricting the visual environment to a pre-constructed toolkit of processes, its utility is restricted purely to process network composition. This reduction in scope removes a significant issue in the development of a visual programming environment — interaction between environment generated and user written program code. In situations where syntax or run time errors can be introduced into the generated program code, the programmer is forced to reason not only about their own program but also the generated code. This restriction also facilitates the use of templating to emulate generics, as described in Section 4.4.6; requiring a programmer to reason about the effects of source-level transformation on their code adds an additional layer of complexity above the occam-pi language.

By omitting the ability to edit processes and making the assumption that blocks present in the environment are valid, the environment is able to ensure that only syntactically correct and compilable programs are constructed. While additional blocks may be created by following the block format specification in Section 4.4.3, adding a metadata header to the occam-pi process source code and placing the block in the toolbox directory, this is not an intended extension route for programmers.

The inability to write sequential code also limits the application for use of the tool for introductory process-oriented programming with students. Examining the tool in the context of the introductory concurrency course at the University of Kent, its current scope is capable

of being used for introductions to process-oriented designs over the first week of the course. However almost immediately after this introduction the exercises completed by students contain both high level tasks and the requirement to complete the implementation of skeleton processes (e.g. compose certain sets of pre-existing or pre-defined processes where one or more processes do not contain an implementation body). A number of tasks also request that students draw diagrams to show the process networks present in their compositions. The tool would continue to be relevant to the high level design portions of these tasks, and act to remove the need for a separate diagramming step; students would be working with the visual representations of their programs continuously. In the absence of a method to introduce process bodies or new process implementations to the visual environment the tool cannot be applied beyond the first week of the course.

POPed is restricted in the size of process-oriented programs it can be applied effectively to for two reasons: the inability to abstract sub-networks of processes into named compositional processes, and maintaining visual representations which resemble diagrams drawn using more flexible methods whilst including all state required for design.

In a typical process-oriented program some processes will be constructed via the parallel composition of other processes - this abstraction of internal functionality facilitates the creation of larger process-oriented programs. In its current state POPed does not accommodate any facilities for encapsulating sub-networks of processes or visualising any internal sub-networks of the top level processes in the network. Permitting sub-networks to be defined and selectively hidden would allow more complex programs to be designed in the visual environment, permitting the definition of sub-networks which may be re-used elsewhere in the program or the collation of a unit of functional behaviour into a single, named process.

As POPed does not employ any automatic layout algorithms when adding processes to the canvas, the organisation of connected or related process groups is left to the programmer. In this respect POPed requires the same effort as in drawing a process network by hand or in a diagramming tool, as the most legible diagram will be the result of choice in positioning of the processes.

A constraint in the use of POPed for larger scale programs is also the representation size of processes. The representations are relatively large due to the requirement of presenting usable click targets on the channel connection points and annotating the process with the name and type of all parameters and channel ends. The large size of process representations restricts the number that may appear on screen concurrently and makes it difficult to get an overview of the program. The environment having the ability to encapsulate sub-networks

of processes inside a process and conditionally display these sub-networks would allow functional groupings which resolve the problems of overviewing large programs with large visual representations.

4.3.2 Robotics Support

Two specific robotics control applications have been considered in the construction of the default process toolbox in POPed. A number of hardware interface processes and general purpose components have been provided to allow the creation of simple pipelines, but specific components have been included to facilitate the creation of programs based on Braitenberg Vehicles (described in Section 3.2) and subsumption architectures (described in Section 3.3.1). These two paradigms for designing robotic programs from communicating components have been applied successfully to process-oriented robotic control and lend themselves well to being constructed with fixed sets of components.

Vehicles

When targeting first explorations in concurrent robotics, Braitenberg's *Vehicles* offer a useful introduction to what can be accomplished with small numbers of processes connected together [Bra86]. By connecting sources of sensory input directly (or almost directly) to outputs, very simple programs can be created that behave in interesting and easily anthropomorphised ways.

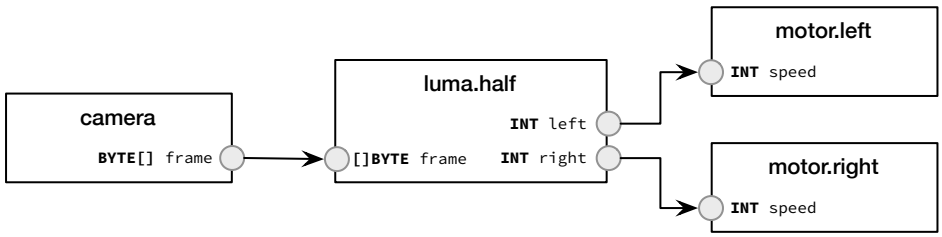


Figure 4.15: *A process network for a Braitenberg vehicle which appears to 'avoid' light*

The program shown in Figure 4.15 shows a simple program for the Surveyor SRV-1 which uses a small number of processes to achieve an interesting result. Image data from the camera is averaged to provide a light level reading for the left and right of the image. These light level values are subsequently sent to the `motor.left` and `motor.right` processes respectively,

which change the speed that the motors on each side of the robot run proportionally to the value received. By connecting the light levels and motor speeds together using a direct relationship, the robot will turn away from light sources. A higher light reading on the left-hand side than the right will cause the left motor to run faster than the right, effecting a right turn away from the direction with the higher light value. The inverse is also true: by slowing the motors as the light level increases, a robot may be programmed that seems to *like* light and heads towards it. To achieve the inverse, the connections between the two halves of the light reading and the motors may simply be swapped over, so each light reading goes to the motor on the opposite side of the robot. Of note in the implementation of Braitenberg Vehicles is that processes which generate values within the network have had their values scaled to the range 0–100, such that the brightness reading from `luma.half` can be used directly as a motor speed.

Subsumption Architecture

Brooks' *subsumption architecture* involves building robot control systems with increasing levels of competence composed of concurrently operating modules [Bro85]. The application of the subsumption architecture for designing process-oriented robotics programs is fully explained in Section 3.3.1, and provides substantial motivation for the use of a graphical process network configuration tool.

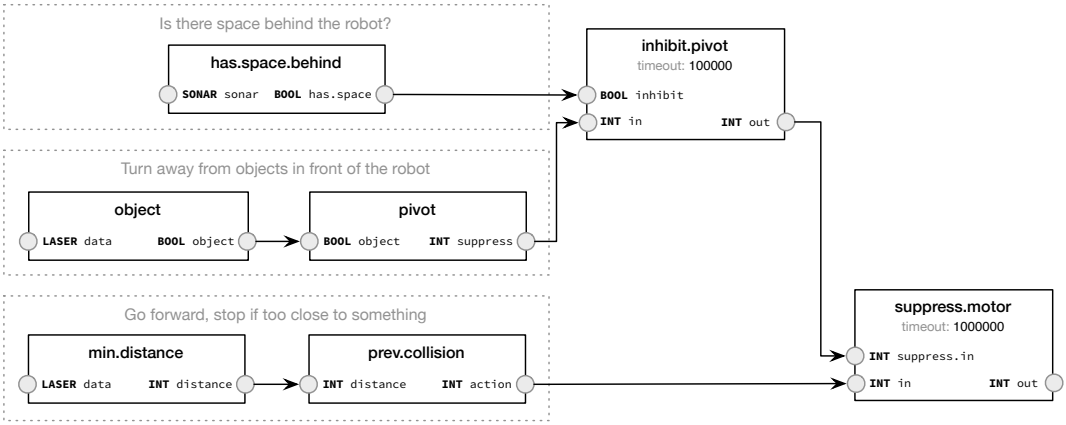


Figure 4.16: *A small robot control program built with the subsumption architecture which uses two types of sensor input and three behaviours to manoeuvre around a space*

Diagrams are invaluable tools for structuring and reasoning about subsumption architectures, as the relations between components and positioning of inhibition and suppression primitives

are visible. To support the creation of subsumption architectures in POPed, the suppression and inhibition primitives are implemented as toolbox processes.

The demonstrator tool is currently limited to a flat process network structure, as discussed in Section 4.3.1, meaning that subsumptive behaviours cannot be distinguished from the groups of processes making them up. To better support the development of large subsumption-based control programs, it would be necessary to allow sub-networks of processes to be collapsed into a single top-level composition process. These compositional processes map to 'behaviours' in the subsumption architecture.

To illustrate the advantages of a visual approach to the design of subsumption architectures, it is useful to make a comparison between a process network diagram for a simple subsumptive program and the occam-pi code required to set up the network as presented in the diagram. Figure 4.16 shows a graphical representation of a simple program which moves around a space and backs away from objects if the robot gets too close to them. The occam-pi program code to set up the process network as shown in the diagram (assuming that all components and hardware interfaces exist and are available) is shown in Listing 4.1.

Each behaviour is broken out separately in the diagram, with the outer dotted box enclosing the sets of processes which are composed to create a single behaviour. The complexity inherent to this code, declaring and connecting the network with named channels of appropriate types, can be completely eliminated using the visual approach. The removal of this complexity is desirable as subsumption architectures grow beyond two or three behaviours to have ten or fifteen levels of behaviour.

4.4 Implementation

The POPed tool is implemented using Python and the wxPython GUI toolkit. Python was chosen as it is already used for scripts in the Transterpreter toolchain — most notably `occbuild.py`, which manages driving the compiler and linking with libraries and `occamdoc` which generates HTML documentation from structured markup comments in source code. wxPython is a Python wrapper around the wxWindows GUI toolkit chosen as it allows a native look-and-feel to be achieved on all three major desktop platforms (Windows, Mac and Linux). Cross-platform support in the composition tool is essential as students on the concurrency design and practice course use a variety of these platforms; the Transterpreter run-time's support for all three modern desktop platforms is also a factor.

```
PROC explore.space ()
  CHAN MOTORS motor.control:
  CHAN INT min.distance:
  CHAN INT motor.in, motor.out, motor.suppress:
  CHAN INT pivot.motor.in, pivot.motor.out:
  SHARED ? CHAN LASER laser.data:
  CHAN SONAR sonar.data:
  CHAN BOOL object, inhibit:
  PAR
    motor(motor.out?, motor.control!)
    brain.stem(motor.control?, laser.data!, sonar.data!,
               default.player.host, default.player.port)
    min.distance(laser.data?, minimum.distance!)
    prev.collision(min.distance?, motor.in!)
    object(laser.data?, object!)
    pivot(object?, pivot.motor.in!)
    suppress.motor(suppress.time, pivot.motor.out?,
                   motor.in?, motor.out!)
    inhibit.pivot(inhibit.time, inhibit?,
                  pivot.motor.in?, pivot.motor.out!)
    has.space.behind(sonar.data?, inhibit!)
:
```

Listing 4.1: *Construction of the process network for the example simple robotics program in occam-pi*

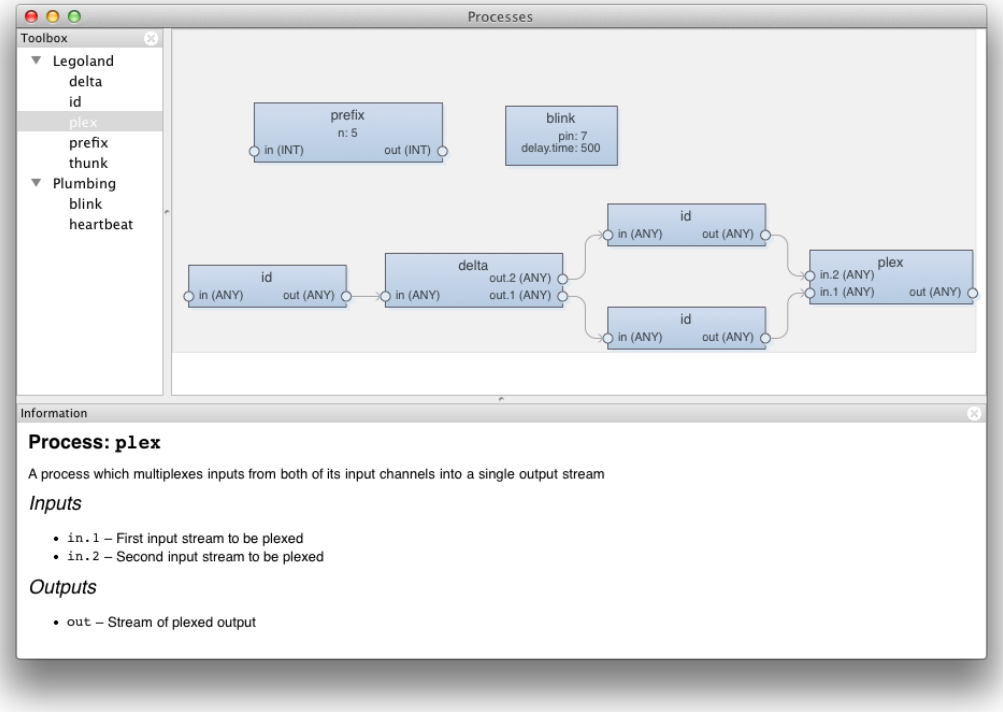


Figure 4.17: *The user interface of* ***POPed*** *, with a number of processes connected by channels*

4.4.1 User Interface

A screenshot of POPed's user interface is shown in Figure 4.17. The same three user interface components as in the design are present: a process canvas, a toolbox of processes and a information panel for documentation about the currently selected process. The user interface is reconfigurable, with the panels arranged by default as specified in Section 4.3. The toolbox and process information panels can be moved to another edge of the window, split or floated independently of the main window to suit individual's preference and different usage patterns.

4.4.2 Process Canvas

Right clicking on a process allows the inspection of its parameter values, as shown in Figure 4.18.

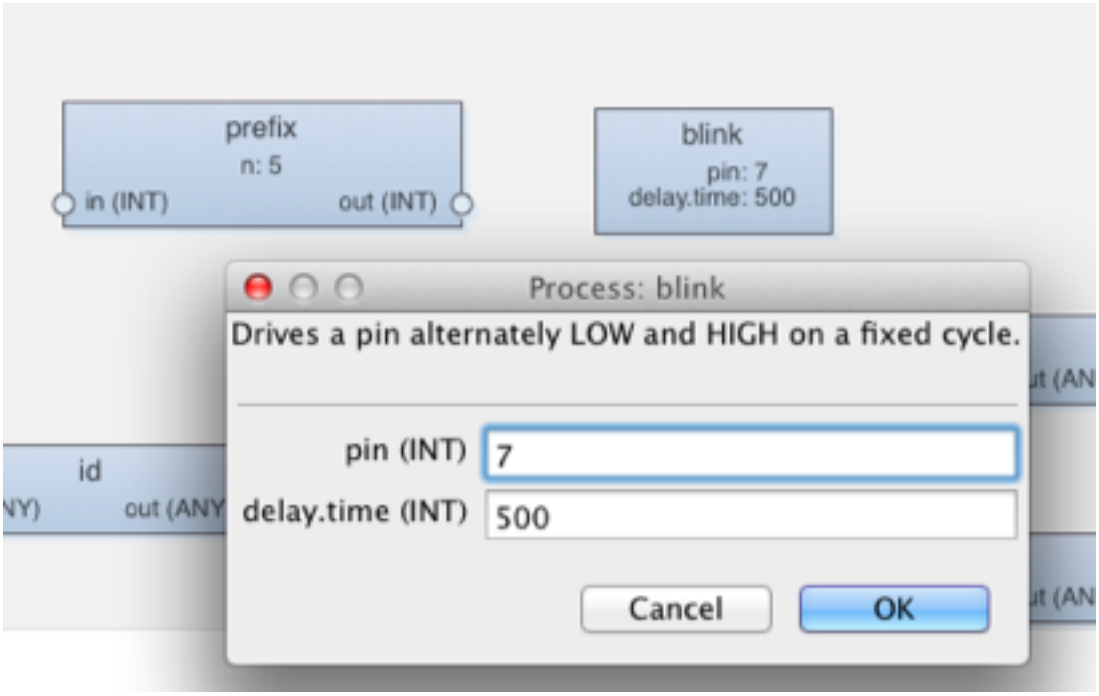


Figure 4.18: *User interface for setting the parameters of a process in* ***POPed***

4.4.3 Process Definition Blocks

The toolbox, to the left of the screen, contains a hierarchical list of process names separated into groupings. These correspond to *blocks*, files organised in a directory structure under a `blocks` directory of the software. Groupings are controlled by the directory layout of the contents of the blocks directory, allowing related groups of blocks to be grouped together and providing a structure to avoid occam-pi's lack of namespace support. An example block for an `id` process, annotated to define the purpose of each metadata field, is shown in Listing 4.2. The example `id` block also demonstrates the generic type emulation supported in these blocks and detailed in Section 4.4.6.

Each block file contains a metadata header in YAML (Yet Another Markup Language) format followed by a dividing mark (`--- Code`), and then the program code of the process. YAML aims to be a human readable serialisation of data structures; a YAML section such as the one in the block files can be loaded with a single library call and produces a set of native dictionaries and typed values, avoiding the complexity of parsing more structured data formats, such as XML.

Storing the interface of the process as separate metadata rather than parsing the header of

```
# Process Name
name: id
# Modules this process uses.
requires:
# List of parameters this process takes, along with their types.
params:
# List of input channel ends: name, type carried and purpose.
input:
 - name: in
   type: ANY.T
   desc: Input values to be buffered
# List of output channel ends: name, type carried and purpose.
output:
 - name: out
   type: ANY.T
   desc: Last value received by the id process.
# Description of the process' purpose.
desc: >
  A process that provides a one slot buffer for incoming values.

--- Code
PROC id (CHAN ANY.T in?, out!)
  WHILE TRUE
    ANY val:
    SEQ
      in ? val
      out ! val
:
```

Listing 4.2: *A process block definition for the POPed visual environment, providing an* `id` *process implemented in occam-pi and using generic (*`ANY`*) type specification*

the process means the environment could very quickly be extended to support processes implemented in other process-oriented programming languages such as PyCSP [ABV07] or JCSP [WB08]. Given that the composition of the network uses the block metadata, only a specialised code generator with knowledge of how to construct a language specific top-level process connecting the network is required to adapt the environment to a new process-oriented programming language.

4.4.4 Toolbox Processes

The set of processes in the toolbox, as identified in the comparison of environments (Section 4.2.3) are critical to the utility of the environment in the absence of the ability for the programmer to define additional components and add them to the toolbox. Given the aim of the environment to allow composition of Welch's Legoland components and the construction of robot programs, groups of processes are present in the toolbox corresponding to these tasks.

Utility Processes

A set of non-task specific processes of general use in process-oriented programming is provided in the environment. The visual representations of these processes are shown in Figure 4.19.

- `id`, a process which provides a one-place buffer by reading input and forwarding it to output.
- `tail`, a process which discards the first value sent and then effectively has the same semantics as `id`, forwarding input to output.
- `delta`, a process which forwards its input to output but also duplicates the output to a second channel, and `plex` which does the inverse, reading from two input channels and multiplexing the two streams of input into a single output.
- `prefix`, a process which first outputs an initial value, supplied as a parameter, then forwards input to output.
- `threshold`, a process which outputs a boolean value corresponding to whether the value it is supplied is over a threshold set as a parameter.

- `black.hole`, a process which discards all input it receives.
- `scale`, a process which scales integer values from one range to another range, specified as parameters.
- `timer`, a process which outputs a tick signal at a given frequency specified by a period.
- `int.diff`, a process which outputs the difference between two input streams.

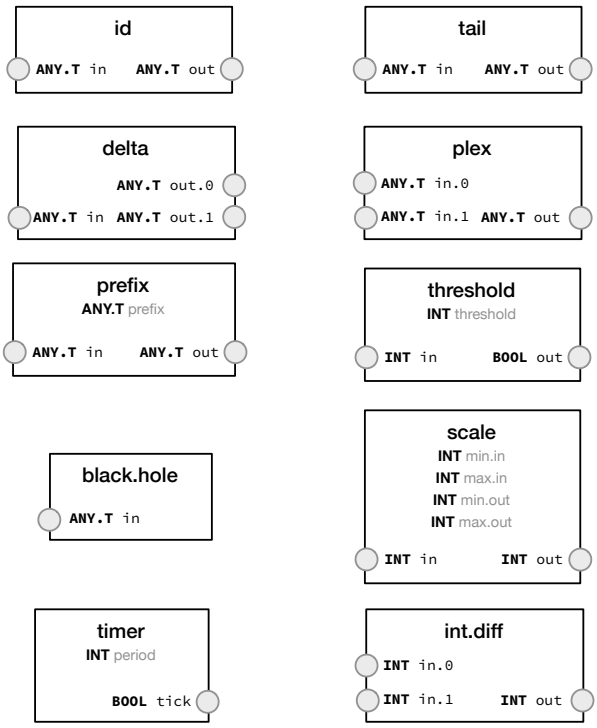


Figure 4.19: *Processes in the 'Utility' group, including Welch's Legoland components and a number of helper processes.*

LEGO Mindstorms RCX

A group of processes are present in the toolbox for interfacing to the Mindstorms RCX and are shown in Figure 4.20. Rather than use two sensor processes, one for all 'active' sensors on the RCX that require power and one for all 'passive' sensors which do not, each type of sensor that can be attached to the RCX has a separate process. Having a separate type of process per sensor retains a mapping between the physical connectivity on the robot and the

process network diagram, as discussed in Section 3.1.5. Interface processes are also present for the outputs and actuators of the RCX: motor driver processes, `lcd.out.int` for displaying numbers on the internal LCD and `speaker` for playing tones. Values scaled to the range -100–0–100 where possible to facilitate direct connections between sensors and actuators; for example motor speeds run through this range, while the temperature and angle sensors will return unscaled values.

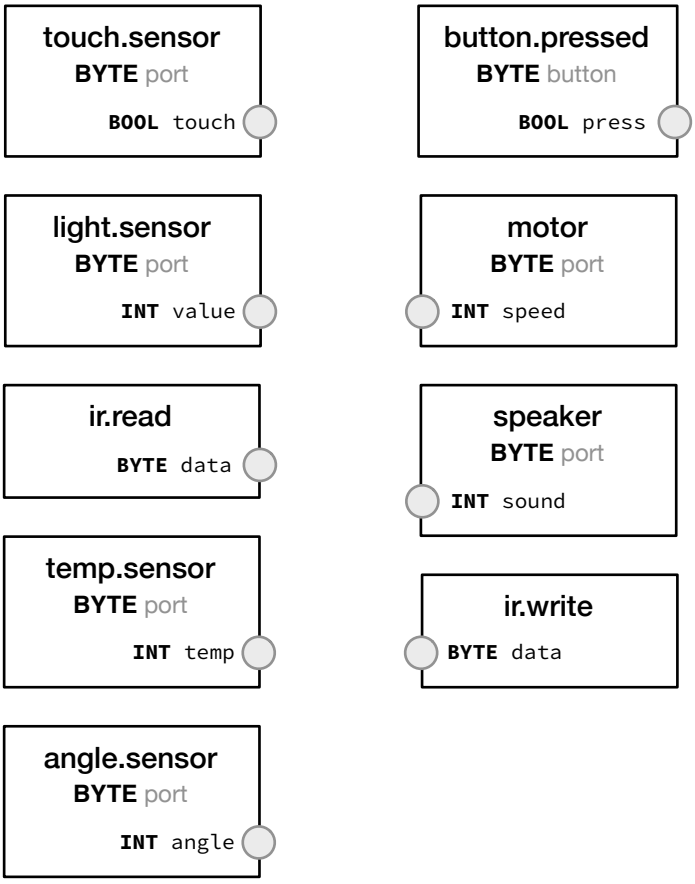


Figure 4.20: *Hardware interface processes for the LEGO Mindstorms RCX*

Surveyor SRV-1

The Surveyor SRV-1 presents challenges to the implementation of toolbox processes. As the high level process-oriented interface to the robot (as described in Section 3.1.6) uses channel bundles, an entirely custom set of processes would have to be added to the environment if the native interface was exposed. Additionally, the generic type emulation implemented in the

environment does not have the ability to distinguish channel bundles. To work around these limitations in the environment a set of processes, designed specifically for use in the POPed environment, are used to simplify the channel bundle operations to basic data types and standard channel communications. These processes are shown in Figure 4.21, their purpose is as follows:

- `camera`, a process which outputs an entire frame from the camera in RGB format as an array of bytes.
- `out.int`, a process which allows a stream of integers to be output the serial console and `out.frame` which permits the same for frames from the camera. The serial console of the robot is transmitted over WiFi to the host computer, where the terminal used is able to pull out image frames and display them to the user based on the header.
- `motor.left` and `motor.right` correspond to the pair of motors driving the tracks on each side of the robot. Values in the range -100–0–100 are accepted with the boundaries correlating to full speed backward, stop and full speed forward respectively.
- `laser.left` and `laser.right` control the two front mounted laser pointers on the SRV. These pointers would typically be used for ranging, as the size of the dot they project will become larger as the robot approaches an object.
- `luma.half` and `luma` both act as a filter on the camera frame data, extracting brightness information. The `luma.half` process calculates two values for the left and right halves of the frame while the `luma` process calculates for the entire frame.

This set of hardware processes for the SRV-1 facilitates the use of the camera on the SRV-1 as a light sensor and as an image source for the host computer. As the SRV-1 was explicitly designed for investigations into vision processes it is not sensor rich; therefore the majority of applications would require additional processes providing other high-level camera functionality.

Subsumption Architecture

As presented in Section 3.3.1, the subsumption architecture makes use of two primitives to provide interactions between groups of components: suppression and inhibition. These two components effectively act as a straightforward `id` process when no interaction between

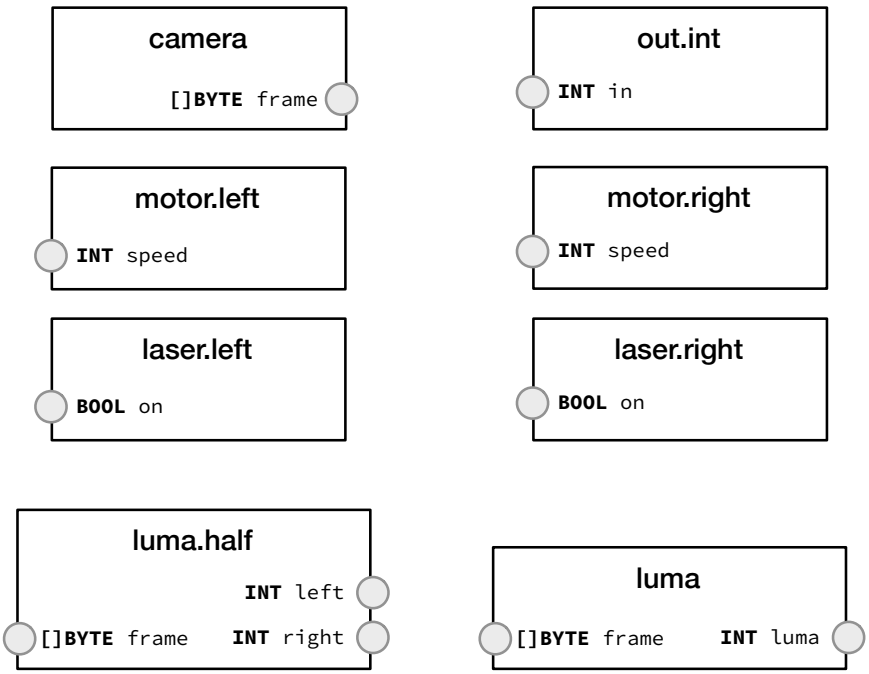


Figure 4.21: *Hardware interface toolbox processes for the Surveyor SRV-1*

Figure 4.22: *Visual representations of generic* `suppressor` *and* `inhibitor` *primitives for use in implementing subsumption architectures.*

groups is occurring, passing input to output. When the suppressor is active, values from input are discarded and replaced with values of the same type from the suppression input; when the inhibitor is activated, via a boolean channel, all output is blocked for the specified timeout. These properties allow generic implementation of `inhibit` and `suppress` in the environment, shown in Figure 4.22.

4.4.5 Use of Toolbox Processes

The use of toolbox processes to construct robot control programs is an important step in proving the utility of a composition-only design tool for program creation. This section presents two example programs for two different robot platforms: the Mindstorms RCX and the Surveyor SRV-1 constructed using a mixture of the general purpose and hardware specific processes.

The Mindstorms RCX program, shown in Figure 4.23, uses an RCX with two light sensors connected and spaced apart at the front of the robot and two tracks on either side controlled by a motor at each side. The hardware configuration of the RCX manifests as sets of processes at the left and right edges of the process network. The program is designed to be a line follower, guiding the robot along a path indicated by a black line on a white background. The light sensors sit on either side of the line, effecting a turn when the sensor intersects the black line (i.e. the brightness value from a particular sensor drops). The program performs this behaviour by taking the difference between the two light values; this difference value is then duplicated and fed into a scaler on each side. The scalers perform the logic in this program, raising the motor speed on one side and lowering it on the other as the value moves from one side to the other. When the difference between the light sensor readings is 0, both motor speed values are scaled to 50 and the robot drives straight; when the difference is -100 one of the motors speed is 0 and the other's speed is 100. As this process happens continuously and reactively to the input values, any condition in the input should correct itself due to motion by the robot and the line is followed.

The expression of this program using a composition-only approach is different to that of an ordinary process-oriented solution. Usually a custom process with sequential logic would be responsible for taking the two light sensor readings and determining appropriate output levels for the motors. The compositional approach has an advantage that the exact path of data flow is obviated at the top level of the program, while the use of a custom process would simplify and remove redundancy from the expression of the scaling logic rather than the use

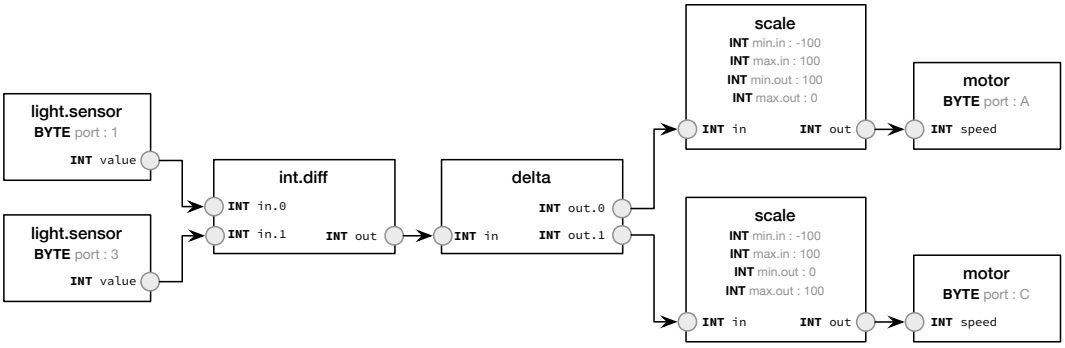


Figure 4.23: *A compositional line following program for the Mindstorms RCX consisting of processes from the RCX hardware interface and utility toolbox groups.*

of two processes.

The Surveyor SRV-1 example program, shown in Figure 4.24 is designed to demonstrate the use of multiple pipelines with image frames. The program both passes the frames into the main network, which controls two laser pointers based on the brightness values in the camera frame, and into a second process which sends the frame back to the host system over the Surveyor's serial link. The ability to add additional pipelines or processes whilst keeping existing networks and processes static is a significant advantage to the compositional model; in this case a second delta could be added and another image processing step used, or if the `out.frame` is being used temporarily for debugging the output from the `camera` process may be connected directly to the `luma.half` process when the programmer is satisfied with the program without having side-effects on the rest of the program. The control system uses a `threshold` process for logic, controlling the `laser` processes directly based on the processed brightness values from `luma.half`.

While this section has demonstrated the application of process toolbox components via composition producing robot control systems, there are a number of limitations these examples highlight in the approach. The introduction of high level processes for logical operations, like `threshold` or the use of the `scale` process to achieve certain effects in the output is not typical of process-oriented programming. Toolbox processes require significant care in their design to avoid requiring a number of conversion or casting processes to allow different components to connect together; a lot of information is lost in the input data in refining it to values which may be applied elsewhere in different contexts.

As discussed in the limitations of the visual environment (Section 4.3.1), programs grow quickly in width due to the representations chosen for processes; a property also evident in

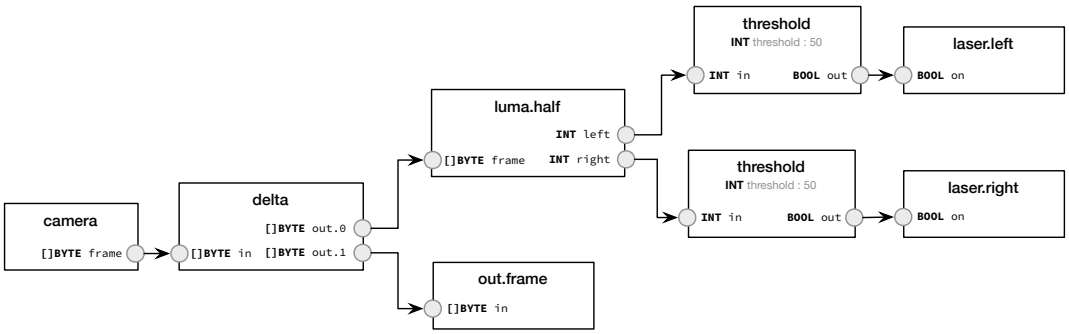


Figure 4.24: *A compositional program for the Surveyor SRV-1 which turns on the laser pointer on a particular side of the robot if the brightness on that side exceeds a threshold and outputs camera frames to a host computer.*

the illustrations provided in this thesis which use a very similar representation. Requiring the implementation of all program logic using composition results in additional complexity in the network of the program; rather than being able to decompose the program functionally, the program becomes a series of transformations and filters over streams of data to reach the intended output form. This requirement for transformations and filters also pushes the interfaces used to expose hardware toward basic data types which pose least difficulty in being repurposed for use as input to other processes.

4.4.6 Emulation of Generic Types

A generic type in a programming language allows a placeholder type to be used at design time and the reference made concrete, resolved to a specific type, either at compile or run time. Generics are typically used to create re-usable code which is able to be parameterised to a given concrete type. For a visual programming environment designed around process composition the lack of generic types in occam-pi is a significant problem. In the absence of generic types, general purpose processes used for data-flow control such as `delta` (splitting one channel into two) and `plex` (short for multiplex, combining two input channels onto one output channel) must be replicated for every type they are to be used with, using different names to differentiate the processes.

In building POPed, this limitation adds such complexity to a toolbox model that it was essential to provide a solution; the alternative would involve having entire groups consisting of variants of these general purpose components in the toolbox. As the program code output from POPed is the combination of process block files and generated composition processes,

it is possible to use template substitution in the environment to simulate generic types for process blocks.

To implement this template-based generic emulation, the block definition parser in POPed recognises keywords beginning with `ANY.` as a type variable for substitution when referenced in occam-pi source code and YAML block metadata. A process may have any number of `ANY.` channel types, and its body may use the `ANY.` type as if it were a concrete type, as it will be substituted before reaching the compiler. A specific identifier is placed after the `ANY.` prefix to permit meaningful naming of generic types and allow multiple generic types in a single process (e.g. `ANY.T, ANY.X, ANY.FOO`). This template-based approach places the burden of correctness on the designer of the process blocks in the environment. Care must be taken to ensure that a specific `ANY.` identifier must be used consistently when reading and writing messages and values inside the process.

In the visual environment itself, a specific `ANY.` identifier is specialised as soon as a channel connection is made to any input or output, and the diagram updated. The concrete type populated is stored alongside the generic identifier, allowing the process to revert to the generic type if the connection is broken. A number of processes can be connected together without specialisation if all connections share the same `ANY.` identifier; once a connection is made with a concrete type the type is propagated to specialise all generic connection points, as shown in Figure 4.25.

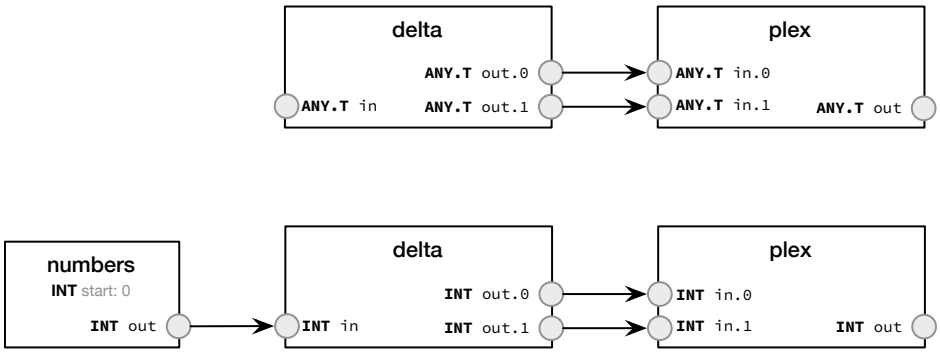


Figure 4.25: *An incomplete process network before and after propagation of a concrete type*

During code generation the `ANY.` identifiers in occam-pi code are specialised to the type defined by the connections made, as shown in Listings 4.3 and 4.4 where a generic `delta` process with a single `ANY.T` identifier is specialised for use with integers via templating.

```
PROC delta (CHAN ANY.T in?, x!, y!)
  WHILE TRUE
    ANY.T val:
    SEQ
      in ? val
      PAR
        x ! val
        y ! val
:
```

Listing 4.3: *Generic definition of a* `delta` *process.*

```
PROC delta (CHAN INT in?, x!, y!)
  WHILE TRUE
    INT val:
    SEQ
      in ? val
      PAR
        x ! val
        y ! val
:
```

Listing 4.4: *Generated specialisation of the* `delta` *process when connected to an input or output carrying integers.*

This limited solution works to address many generic component definitions and make the use of general purpose network routing components feasible in the visual environment without replicating every process per-type. Cases in which the data type is complex, such as a `PROTOCOL` variant or collection types which must be indexed into would require actual support for generics in occam-pi. As the specialisation process is completely separated from the program code, it can be difficult or impossible to create generic processes whereby two different generic types are passed in and used in a single operation, as it is complex to produce conditional logic using the choice of type at run-time.

4.4.7 State of Implementation

The development of the tool is not fully complete, and as such, only a subset of the complete set of desired functionality is currently implemented. It is worth re-iterating that this tool is intended to serve only as a demonstrator, as a full implementation of an educational tool for parallel programming in occam-pi would require a more detailed analysis of use cases and the interaction between students and the visual representation of their programs. The work required to apply the principles in POPed to general purpose process-oriented programming is discussed in Section 6.2.1.

4.4.8 Reflections on Implementation

The implementation work carried out to date in building POPed has provided a number of insights into building a visual programming tool which are significant to any effort to develop a similar tool or for further development of POPed.

As the Transterpreter virtual machine is highly portable to allow the use of occam-pi in more places, a deliberate choice was made to avoid restricting the visual environment to a single desktop operating system. While Python has worked well as an implementation language, and wxPython's initial promise of providing a native-feeling UI, wxPython hampered implementation through lack of documentation and poor implementation of cross-platform features.

Choosing to implement a process canvas from scratch, rather than re-using an existing graph component yielded significant flexibility in the interaction model and exact representation of processes. However, this slowed development considerably and in reflection, customisation of an existing graph library would be an appropriate trade-off to benefit from developments in graph theory and layout algorithms for loading existing programs without a prescribed layout.

Chapter 5

Introspection and Debugging

The effective application of process-oriented programming for robot control, as described in Chapter 3, is subject to the limitations of the program development environment. Process-oriented programs, like any computer program, can and will contain errors despite the best intentions of the programmer. Debugging these errors is an essential part of the edit, compile, test cycle of development. Support for debugging and examining the behaviour of concurrent programs is critical to permitting effective use of the programming model.

Support for debugging is especially critical to the applications of process-oriented programming in robotics. Where the robot is a separate platform to the host computer — the majority of cases in our use of occam-pi for robotics — run-time errors occurring in the program will at best manifest as a printed message containing a virtual machine error code, at worst as a non-responsive run-time environment with significant safety concerns.

Tools and techniques for debugging programs written in imperative languages containing a single thread of control are limited in their efficacy for concurrent programs. Debugging software which uses concurrency is a difficult problem in general; the temporality of multiple threads of control introduces complex problems: non-determinism, deadlocks and race conditions [MH89]. Even building tools to facilitate the debugging of these problems is complex; temporality means problems can change or disappear when the run-time performs additional activity to monitor the behaviour of the program (the "Probe Effect" [Gai86]). Debugging process-oriented programs is complex. While the programming model and compiler support prevent some classes of concurrency error, each process executes independently, distributing execution state throughout the program.

The need for rich debugging tools was present at the inception of occam as a language for

programming the Transputer, and a number of debugging and program visualisation tools were designed for use when developing occam programs on these hardware architectures. With the shift away from transputer hardware to modern desktop run-time environments for the occam-pi language and several decades of operating system advancement, these tools are no longer available or applicable.

The design of a tool for examining execution of an occam-pi program for effective debugging of robot programs is presented. This tool may also be used as a demonstrator, with a pedagogic motivation, allowing the inspection of programs specifically designed to encounter conditions of livelock and deadlock.

Using the visual language and environment established in Chapter 4 as a basis for visualisation of program state, given information from the run-time environment about the behaviour of the program, gives a head start on establishing such a tool from scratch. One of the most significant challenges is in representing the execution in a way the programmer can relate to their program. Starting with a program which has been designed in a visual tool means a layout has been established by the user and allows the user's choice of diagram to be used for representation.

The motivation of this tool is to allow observation of robot program behaviour, and identification of errors in the software.

5.1 Errors

To classify the handling of concurrency errors in different programming models and languages, it is important to first establish the classes of error which can occur in the program development cycle. There are three classes of error: compilation, run-time, and logic. Depending on the programming model and language in use, concurrency errors can manifest in any of these three classes of error.

5.1.1 Compilation Errors

Compilation errors occur, as their name would suggest, at compile time when the user performs an action to convert the program code they have written into machine code. Compilation errors can broadly be broken into syntactic and semantic errors. Syntactic errors are where the composition of the program is incorrect; the user has forgotten a bracket,

a semicolon or an argument to a function. Semantic errors occur where the meaning of the program is clear, but the operation as specified is incorrect; a number may be too large for the type it is stored in, an argument to a function may be of the wrong type. These errors prevent the program being compiled, and therefore coming into existence as machine code - a program with compilation errors can't be compiled. Compilers may also produce warnings to encourage good code hygiene (such as identifying unused and undefined variables) or identify potential run-time errors in cases where certainty cannot be established.

5.1.2 Run-time Errors

Run-time errors occur where operations that occur whilst the program are running produce invalid instructions for the computer (or virtualised run time, in the case of a virtual machine) to execute. For example, in the statement `speed = distance / time`, if `time` were to be zero, then the division operation generated by the compiler will fail. At compile time, the values of these variables are not known, so the error can't be detected. Static analysis is able to move an ever increasing number of errors of this kind back into the compilation error category, analysing the produced machine code to produce ranges of likely values and producing errors if it can definitely identify a path through the code that will end in a state suggesting run-time error. Static analysis is not a complete solution - it cannot account for invalid user input or a number of run-time conditions that can be encountered (although it may be able to warn about unchecked inputs). The kinds of run-time error categorised here are fatal, producing inconsistent state in the machine that results in program termination.

5.1.3 Logic Errors

Logic errors are a second kind of run-time error, but one which is far deeper, bears less traction for static analysis techniques and is most critical to the programmer who is able to create syntactically valid programs. A logic error is effectively a difference between what the programmer expects the program to do, and what it actually does when run; a program with a logic error in it is still a valid program and the error is invisible to the computer itself. For the programmer, resolving the difference between their intent and the actual function of the program requires understanding why the program behaves as it does, given the code of the program. While this may be possible to achieve by inspecting the program code and reasoning about it line by line, often programmers have to debug the program, by examining

its run time state for variations from what would be expected.

5.2 Concurrency Errors

As far as concurrency is concerned, many languages must treat errors as run time logic errors; the fact that unsafe memory usage patterns exist are not typically treated as hard compiler errors that must be fixed before the program will compile. At compile time, occam-pi programs are checked for safe concurrent usage of variables inside processes, and the program will fail to compile if two processes are able to write to a variable at the same time. Parallel usage is therefore removed as a class of run-time error, as the compiler is able to analyse for it. Process-oriented programs written in languages like Java or Python, using primitives implemented as libraries cannot do these checks, as the underlying language still contains the potential to create unsafe parallel usage operations. Whilst the output of the compiler, `occ21`, can be cryptic at times, the most commonly encountered errors are reported clearly, making writing an occam-pi program which compiles not tremendously difficult. The motivations of this thesis, Section 2.2.1, discuss the advantages of using occam-pi for process-oriented programming rather than an implementation of the primitives as a software library.

5.2.1 Non-determinism

Non-determinism in program execution presents significant challenges when debugging; when something goes wrong with a program, the state in which the system was in is critically important for a programmer to identify the issue. In an imperative program with a single thread of control, the program follows a series of statements to produce the current state of the machine; the execution behaviour can be reasoned about from the source code. Where the system executes in parallel and a scheduler provides access to machine resources, the behaviour of the scheduler is a factor in the execution behaviour of the program. Where programs are running in parallel, the execution time of components affects the scheduling behaviour, and the state of the machine; an error may happen in a very small number of cases where particular execution times overlap. Non-determinism also negatively effects testing, once an error is fixed the programmer may not be able to replicate the scheduling or execution state conditions in which the program reached the error. It is often the case that testing for the presence of concurrency errors requires running critical sections very large

numbers of times to extract temporal and scheduling edge cases.

5.2.2 Livelock and Deadlock

Livelock and deadlock are introduced as concepts relatively early on in teaching process-oriented programming, as they are the most commonly encountered run-time errors in process-oriented programs.

Deadlock is a condition where two or more actions are waiting on each other to finish, and thus the actions can never complete; a real world example being two people meeting in a corridor where they cannot pass each other, and both stopping to let the other go first, neither making progress. Figure 5.1 shows two examples of process networks that encounter deadlock, used as the first introductions to deadlock on the concurrency design and practice course at the University of Kent. A network of `succ` processes, which receive a number on their input and output the number incremented, all of the processes in this network wait cyclicly for a number on their input anti-clockwise. A network of `n` processes, which output the number they are supplied with as a parameter when they start, and then revert to behaving as an `id` process, passing input to output; this network deadlocks as every process is waiting, trying to output to its clockwise neighbour.

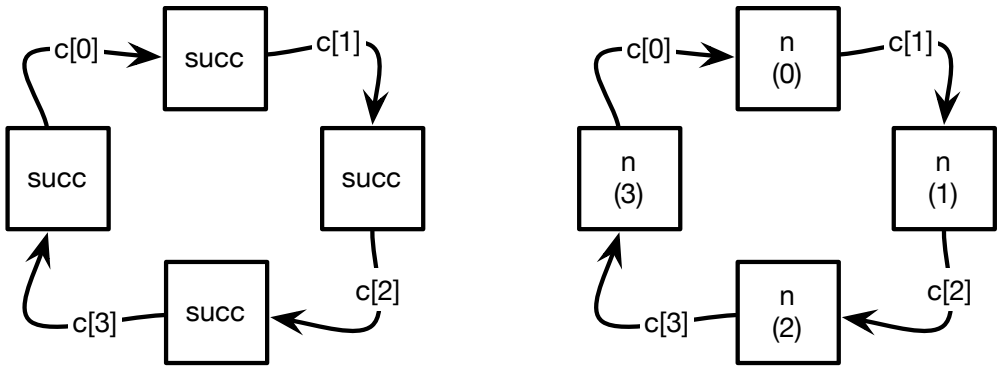


Figure 5.1: *Two examples of process-oriented programs designed to illustrate deadlock.*

Livelock is a condition where a group of actions engage in communications with each other infinitely, not responding to the outside world; the same real world corridor example would be both people moving to the other side, meaning that after both move they are still blocked.

An incorrectly designed process-oriented program may contain a cycle of processes which can enter a state where they are waiting for input from each other, or the ability to add

additional traffic to the network.

5.2.3 Race conditions

The process-oriented model is designed to facilitate the sharing of data by communication. If a process needs a particular piece of data, it should receive this data over a channel, where the exchange will explicitly move the data from one process to another, as the sending process will lose the data. Where references are used, for `MOBILE` data in occam-pi or as the standard variable access method in other languages, ownership of references should pass with the communication, meaning the sending process cannot maintain access to the reference, or an alias for it after the communication. The occam-pi toolchain does parallel-usage and aliasing checks on the program when compiling it to verify that data is used safely; programs that pass the verification checks can be guaranteed free of race conditions.

Where process-oriented programming is achieved via library support such as JCSP in Java or PyCSP in Python, the compiler and/or interpreter are not able to reason about the concurrent aspects of the program. In other languages, the programmer must be careful to observe the anti-aliasing and parallel usage rules of process-oriented programming without assistance from the compiler, again making race conditions a possible source of error.

5.2.4 Debugging

Debugging is a human process; finding the difference between the programmers' intended behaviour of the program and the program as expressed and executed. The goal of debugging is to obviate the state of execution of the program to determine where the actual behaviour has diverged from the expected behaviour. There are two main approaches to debugging: tracing, outputting information about program state and use of a debugger, to control execution of and allow inspection the state of the program.

Tracing

A programmer's first experience of debugging logic errors is often via "`printf` style debugging", named for the `printf` function that allows output of text and variables to the terminal using the C programming language. Early on when learning to program we learn how to print a message to the screen (the classic "Hello World" example being prevalent). This one

of the first functions encountered is the variant of `printf` style debugging for that particular programming language. As there is a single thread of control through a program written in an imperative language, insertion of commands to print messages containing program state at particular points allow the programmer to observe what is happening during execution of the program. A program annotated with print statements will output a trace of its execution, limited to the elements annotated by the programmer. The efficacy of this method is entirely determined by amount of thought applied by the programmer in placing the print statements, determining which state to print and the appropriate points in the program's execution to print it. As Kernighan states: "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements" [Ker84].

The advantage of this practice is simplicity; the programmer can quickly annotate the elements of the program that they are interested in to be printed or logged. The disadvantages are numerous; adding and subsequently removing the print statements will change the semantics of the program, and may change its behaviour in ways that either resolve or change the original bug. This is particularly true for race conditions, where the temporal ordering of reading and writing to memory is responsible for the error. Once the logic error is fixed, the programmer must remove or disable the `printf` style debugging without affecting or otherwise modifying the rest of the program - if another related or similar error is found subsequently, the messages must be put back. There are various strategies for managing this kind of output, such as `printf` style functions that automatically disable themselves when the program is not in a 'debug' mode, but the source code will still be obscured by the debugging annotations.

In a imperative program using concurrency, multiple threads of control are introduced, making the use of `printf` style debugging more difficult, especially where the issue being debugged spans multiple threads. Debugging alters the behaviour of concurrent programs using threads; if a thread is stopped to inspect its state its synchronisation and resource usage patterns will change. Using `printf` statement debugging in a multi-threaded program can expose bugs that did not previously manifest; printing is relatively expensive and will slow down the operation of threads, changing the timings of execution in critical sections of the program or waiting to acquire locks. This behaviour is known as the "Probe Effect"; where an incorrectly synchronised concurrent program behaves differently when delays are introduced [Gai86]. These temporal effects can also result in a program which behaves correctly when debugging statements are added, and incorrectly when they are removed — meaning there is a synchronisation error in the program as specified before adding debugging

delays.

The efficacy of tracing for concurrent programs, particularly those based on a synchronization model of communication events has been previously explored. LeBlanc et al. identified that by maintaining the sequence of events and decisions made, parallel programs can be replayed" [LMC87]. Brown and Smith have explored the recording of traces for reasoning about and understanding the underlying CSP formalisms of process-oriented programming [BS09].

Debuggers

A debugger executes the program to be debugged in an environment where it can control execution flow and monitor the state of the program. The GNU Debugger (`gdb`) is typical of the widely used debugging tools for imperative programs, allowing inspection of program state such as the current call stack (showing the nesting of function calls resulting in the code being run) and the values held in variables. The debugger makes this inspection available when the program reaches a fatal error, although the user is also able to set points, known as 'breakpoints', where the debugger will pause the program in a state allowing this inspection. The execution flow of the program can also be controlled once a breakpoint is reached; the user can choose resume execution until the next breakpoint (resume) or step forward a single statement in the code (step), optionally jumping into functions to step individually therein (step into). These debuggers are powerful tools, allowing the user to leave their program unmodified and inspect its state in detail, but they have fairly steep learning curves. `gdb` has a command line interface, with commands for the above actions, and the information they return is purely textual. Integrated development environments often wrap user interfaces over this functionality, allowing the user to click buttons and see more structured output.

Support for debugging concurrent programs, with multiple threads of execution in the same program, while available in `lldb` and `gdb` is limited to switching the state inspection and control of execution flow abilities between different threads in the program. An important part of debugging via a tool is relating the state of the program at a breakpoint or fatal error effectively to the programmer, in a way that allows them to relate the fault to their mental model of the program's execution. In a sequential program with a single thread of control, the source file and line number (potentially aided by a call stack trace) are enough to be able to allow the programmer to identify the source of the error and make assumptions about how the program reached that point. The programmer's mental model of the system's behaviour in combination with the sequence of statements in the program and the state reached at the

error is often sufficient to identify logical errors in the program.

In a concurrent program there are multiple threads of control, so knowing a single position in the source or how the process was started is significantly less useful. The error being debugged could be the result of synchronisations between these threads of control, or be the effect of a cascading error from another process. It is useful to be able to see the state of each concurrently executing process and each of their positions, along with the state of communications and synchronisations between them.

For occam-pi programs, KRoC supports basic post-mortem debugging; when a program terminates due to fatal error the process being executed and the line position of execution are printed to the terminal. A standard approach to providing richer debugging information is using a logging channel, connected throughout the program and shared for writing, where all processes can print log messages to create a trace of important events during execution. However, this approach changes the temporality and synchronisation patterns of the program and creates the probe effect. Printing to a shared channel also manifests many concurrency problems: messages are output character by character, leaving the potential for them to interleave and be corrupted, and any kind of buffering on the output channel may cause messages to be lost on execution or break the relationship between the output and the events of the program.

5.3 Related Environments

There have been a number of previous efforts in providing debugging tools for process-oriented programs written in occam and occam-pi, especially for developing with physical Transputer hardware. The majority of these tools have suffered from occam and the Transputer's decline, and do not exist in a form in which they can be adapted or extended to work with modern occam-pi toolchains and runtimes. However, their design serves to inform the design of a debugging environment for modern occam-pi programs and their functionality provides a useful starting point for establishing the need of programmers in debugging process-oriented programs.

5.3.1 INMOS Transputer Development System Debugger

The Transputer Development System Debugger supplied by INMOS for use with occam on the original Transputer hardware allowed inspection of the processes running on a particular Transputer including the values of variables inside them and contents of channel communications [O'N87]. The interface was completely command based, with no visual representation of the layout of the processes or network. The feature-set of the TDS Debugger primarily replicated the standard set of debugging tools available for imperative language, breaking on errors and allowing the user to step the program. While stepping, the user could inspect named variables in scope and perform the equivalent of jumping into functions in an imperative language, jumping along channel communications into the process on the other end of the communication.

5.3.2 GRAIL

Stepney's Graphical Representation of Activity, Interconnection and Loading (GRAIL) was a tool for representing an occam program graphically to examine its parallel structure, communications and performance characteristics [Ste87]. There is no support for program creation or editing in GRAIL, these functions are left to existing editors; the visualisations are primarily intended for examination of program design and performance analysis results. GRAIL represented the process network structure as shown in Figure 5.2, using rectangular boxes for processes with arrowed lines drawn over the top indicating channel connections; compositional processes are shown as dark shaded outlines. GRAIL also used a visual layout for sequential and parallel code inside processes, indicating the structure of the code using nested boxes and placing sections of code which were run in parallel horizontally level with each other.

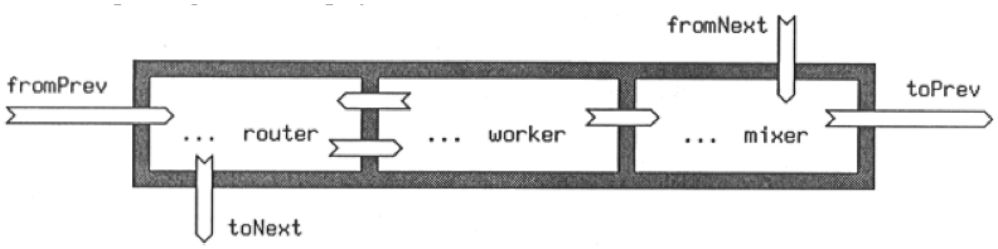


Figure 5.2: *GRAIL displaying a network of three parallel processes connected by channels, from Stepney [Ste87]*

The activity monitoring of GRAIL is the most significant element as regards development of an introspection tool — the TDS compiler was modified to insert profiling statements capturing information about execution counts and this information was returned from the Transputer hardware when the program terminated (successfully, or otherwise). Based on the returned statement execution counts process boxes were shaded from red (for 'hot' sections, with large execution counts) to blue (for 'cool' sections, more infrequently executed with low execution counts). This simplistic analysis and visualisation was designed to give the programmer insight into where the computation hotspots were in the program, informing and evaluating of choices about process placement across across the hardware Transputer cores. While process topology on physical hardware is not a concept with directly relevance in modern occam-pi runtime environments, annotating process network diagrams with performance information would be a clear form of communication for the results of such analysis.

5.3.3 POPExplorer

Jacobsen's POPExplorer is a graphical program builder created to take advantage of the properties of a virtual machine runtime in allowing data collection about program execution [Jaco6]. A number of extensions were made to the Transterpreter virtual machine to enable a textual command interface for loading code and controlling individual process execution; these modifications were invasive and as such not integrated into the virtual machine on an ongoing basis. The POPExplorer UI is shown in Figure 5.3, the interface presents a list of processes by name and allows users to drag them onto a canvas, where they can be wired together. The environment is live; the action of dragging a process onto the canvas does not merely update the visualisation, as when dropped the byte-code for the process is loaded into the VM and the process is instantiated.

The state of channel ends is indicated in POPExplorer by the colour of the channel end points, and the overall execution of the constructed program can be controlled (run, step, step to next communication). POPExplorer is an indication of the promise in instrumenting the Transterpreter to provide run-time information about programs without changing their execution behaviour. Cleanly integrating support for tracing into the Transterpreter VM would provide high-level, rich debugging functionality for all programs and keep execution behaviour consistent.

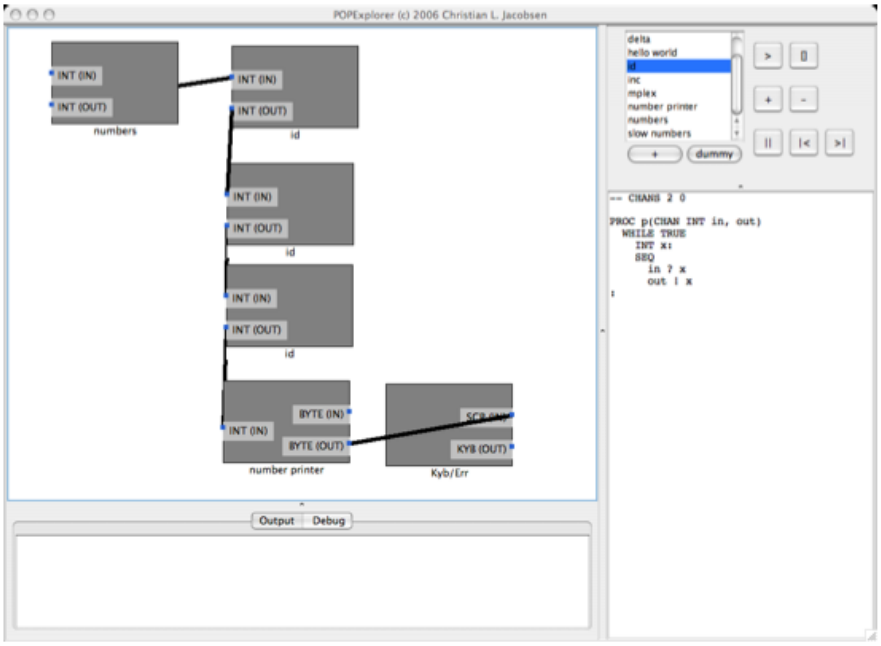


Figure 5.3: *The POPExplorer Environment, From Jacobsen [Jaco6]*

5.4 A Debugging Environment for Process-oriented Programs

In using process-oriented programming for robotics, as described in Chapter 3, it has become clear that as programs grow it is harder to reason about unexpected behaviour by the robot. Posso's larger scale testing of subsumption architectures in occam-pi specifically identifies issues of behavioural stall that in the absence of debugging tools for occam-pi, cannot be definitively traced to the architecture itself or the specific implementation [Poso9].

Reasoning about programs becomes more difficult due to the observed action of the program being achieved through the interaction and resolution of choice between many different components and layers of state. Adapting the diagramming strategies for representing process-oriented programming described in Chapter 4 in the context of visualising program execution would allow for a clear picture of the system's execution behaviour and internal state to be presented.

In designing a debugging environment for robotics there are other concerns; facilitating real-time control of the execution environment and streaming of execution state information from a mobile platform is difficult and requires many of the facilities usually required only for teleoperation. The probe effects discussed earlier in this chapter are exaggerated by the need to synchronise and update a remote visualisation environment and therefore significantly

likely to cause different behaviour when the program is under visualisation. The use of tracing, recording execution state and allowing it to be replayed after execution or the occurrence of an error provides more flexibility on small mobile robot platforms and permits consistency of execution, addressing a number of the concurrency debugging issues identified previously in this chapter.

5.4.1 Visualisation of Execution State

The existing principles for drawing process network diagrams of program design established in Section 4.1 cover only program design. Given the amount of information contained in a process network diagram for this purpose, it is challenging to add information about program execution state without overwhelming the diagram. A number of previous tools discussed above (Section 5.3) have drawn state onto network diagrams or process representations, but no conventions have been established. Extending conventions established in Section 4.1 for drawing process network diagrams with annotations and adaptations for execution state visualisations provides a framework for using diagrams for debugging.

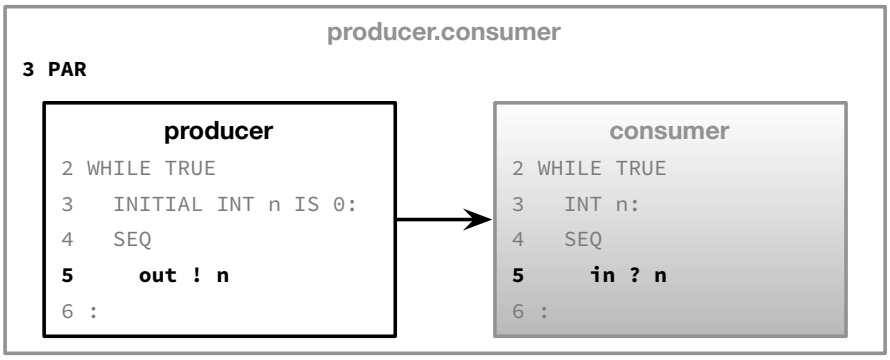


Figure 5.4: *A producer and consumer process being run in parallel, where the current execution position is line 5 of the producer process*

To reason about the behaviour of a program, knowledge of the current execution position or positions (in the case of a multi-core scheduler) is essential. Figure 5.4 shows a process network diagram with adaptions for highlighting the current execution position; the currently executing process is shaded in grey and the current source line position of the `producer` process is highlighted in red. The scheduling behaviour of the runtime is obviated in the movement of the highlight around the network.

Showing the context of the process' source code position in the diagram itself aims to allow

the programmer to better understand the current position and allows for value inspection of nearby variables as discussed later in this section. As the Transterpreter is a single threaded run-time at this time, there is only a single highlighted process at any time.

The top-level process in the network shown in Figure 5.4, `producer.consumer`, is an example of visualisation of process hierarchy. Both `producer` and `consumer` are forked in parallel from `producer.consumer` and as such, the processes are drawn inside their parent process. While this method captures the exact structure of the network, it is expensive in terms of space, and as the nesting levels get deeper the size inflates further, as the size of processes do not scale. To resolve this problem, processes may be dynamically expanded and contracted as the sub-networks inside them are being executed, but this yields additional problems in terms of resizing the diagram.

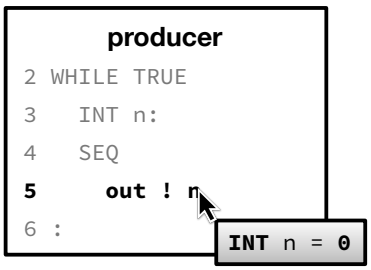


Figure 5.5: *Inspecting the current value of a variable inside a process being executed*

Value inspection is a key accompaniment to the current execution point while a fairly standard feature of debugging tools in integrated development environments like Microsoft's Visual Studio, is significantly useful as all state is retained within processes. A small information popup is shown next to the cursor when the user hovers over a variable name in the source code snippet in the process which has already been executed, with the type and current value of the variable. Hiding the information in this way keeps the process the same size, while allowing the user to examine state as required.

While variable inspection allows some reasoning about what is being communicated along channels, it is helpful to be able to reason about a history of values communicated between processes. A channel inspection popup, similar to the one suggested for variables, is shown in Figure 5.6, the channel's name and type are referenced from the parent process where the connections are made and shown at the top of the popup. A history of the last five values is provided, and as in the example figure, this communication history can be extremely helpful in elucidating the behaviour of a component and the network in general. Values that are

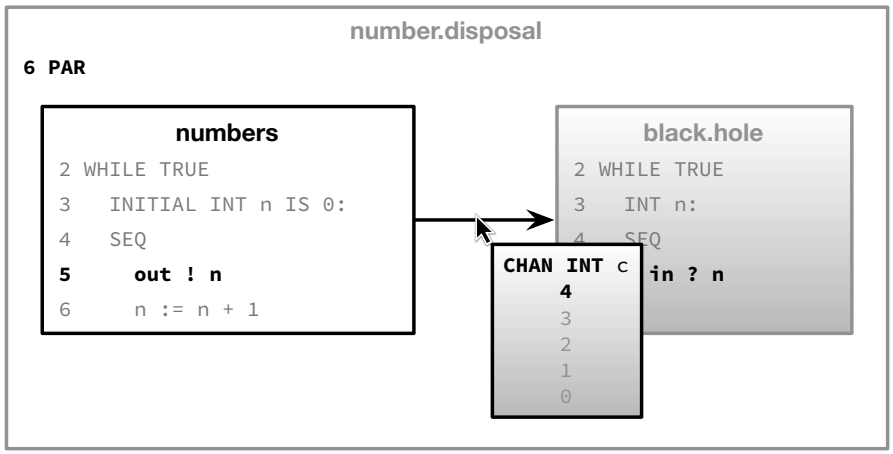


Figure 5.6: *Inspecting the state of a channel, with the last 5 communicated values and the current value on the channel waiting to be read highlighted*

waiting to be communicated on a channel are highlighted in red, as shown in Figure 5.6 – the `black.hole` process has not yet been rescheduled to continue from its synchronisation point waiting to read from the channel, and as such the value 4 is still waiting on the channel itself.

5.4.2 Layout

The structure of a program is used by the programmer to break down a solution into meaningful components with defined relationships between each other. When designing a program visually these relationships are also captured in the layout of the process network; the user creates a spatial model for the program.

When visualising a program which has not been graphically designed there is no source of explicit layout information in the program source or run-time state. Automatic graph layout algorithms, such as spring embedding are a possible solution to this problem; however the results do not resemble process network diagrams as drawn by programmers. While drawing a process diagram showing the execution state of the program is a technical challenge.

If the diagrams drawn do not relate to the programmer's own mental or visual model of the program there is still an indirection in mapping between the two. Where the program has been graphically designed, using a tool such as that described in Chapter 4, the user specified visual representation for the relation between processes in the network should be respected as closely as possible.

5.5 Proof of Concept Implementation

To demonstrate and examine the utility of the ideas described in this chapter the author collaborated with Ritson to create a proof of concept implementation [RS08]. This proof of concept was designed to be used in place of drawings and explanations to introduce students to the concept of deadlock in process-oriented programs, introduced in Section 5.2.2. This forms a minimal viable example of program visualisation, to present the example the processes involved must all be shown, their current execution positions reasoned about and the state of their channel communications shown. The proof of concept implementation consists of three pieces of technical work: adding support for execution control and querying runtime state to the Transterpreter, a tracing program which uses the runtime state query support in the VM to output traces, and a trace visualisation tool *TC1*.

5.5.1 Virtual Machine Support for Debugging

Use of a virtual machine (VM) runtime facilitates the use of introspection; a VM must necessarily virtualise all execution state by definition. Hence it is possible to modify a virtual machine to make the state available externally, or to allow external interference with execution behaviour. This section has covered the dangers of doing so, in introducing timing effects or runtime behaviours which do not exist in an unmodified environment. This is not to say that a compiled program may not be instrumented; code may be introduced at compile time which instruments the program, or a modified runtime library may be compiled with the program. However, the use of a virtual machine runtime for introspection means programs remain unmodified and the virtual machine may have its state inspection features enabled or disabled as desired — a significant aid when debugging.

To facilitate the development of introspection and debugging tools with the Transterpreter, a number of extensions were made to the virtual machine to permit execution state monitoring. These extensions are based on principles established in hardware debuggers, such as JTAG [IEE01] and are designed to support the creation of program introspection features as described earlier in this chapter.

Existing support for executing multiple programs within the same virtual machine context, discussed in Section 3.1.6 in the context of allowing a robot hardware interface and user programs to coexist, was extended to support a program in one execution context controlling and inspecting the execution state of a program in a second context. This support is achieved

via a channel based interface which supports the following operations:

- **Run**, execute byte-code instructions until the next breakpoint, error or program termination occurs.
- **Step**, execute a single byte-code instruction.
- **Dispatch**, execute a set of instructions supplied along with the command, injecting code into the execution stream.
- **Get/set state**, access or modify virtual machine registers, stack and clock; particularly useful for capturing state at a given time to later being restored.
- **Read/write memory**, restricted to when program execution is stopped to preserve consistency, allowing access to and modification of virtual machine memory.

These operations permit detailed state logging and execution control, the building blocks for any kind of program introspection or live programming tool. The transputer byte-code format has also been extended to include source line information allowing the currently executing byte-code to be matched up to a position in a source file. There are limitations of this approach, external inputs cannot be controlled, and timers introduce temporality to program behaviour, meaning the debugging interactions can affect the semantics of the program. The issue with timers can be resolved by switching to using a timer based on the cost of instructions executed, meaning the real-world time differences if the debugging commands interact are not evident to the program.

5.5.2 Tracing

A small tracing program is loaded into the virtual machine which dumps a plain-text trace of process activity; this trace is lightweight, containing only the current process or channel pointer memory address, source line information (available directly from the bytecode, as detailed in Section 5.5.1) and a keyword per operation. These keywords are: `start` for process creation, `=>` for the virtual machine switching context between processes, `call` for a function call, `input from` for a channel read and `output to` for a channel write.

Despite additional information being available at run-time, the initial trace is purposefully lightweight as to make its effect on execution of the program being logged as minimal as

```
#00804CB8 @ sort_pump.occ:308
#00804CB8 start #00804630
#00804CB8 @ sort_pump.occ:309
#00804CB8 call sort
#00804CB8 => #00804CA8
#00804CA8 @ sort_pump.occ:74
...
#00804C10 @ sort_pump.occ:57
#00804C10 input from #00804CC8
#00804938 @ sort_pump.occ:310
#00804938 call test.rig
#00804938 => #00804928
#00804928 @ sort_pump.occ:278
```

Listing 5.1: *A sample of state records from the lightweight trace of* `sortpump`

possible. Source line information is stored to facilitate post-processing and enrichment of the log; a more detailed XML formatted trace is created through this enrichment process, combining each line with the actual source code from the program and attaching names to processes from the code. This detailed trace forms the input to the visualisation program, allowing a separation of concerns between visualisation, tracing of execution behaviour and parsing program source code.

5.5.3 Trace Visualisation

The trace visualisation tool, TC1, shown in figure 5.7, is an experimental implementation based on the principles established in Section 5.4. Drawing process network diagrams for introspection purposes poses a different set of constraints and requirements to drawing the networks for program design, as considered in Chapter 4. Adding execution state information to the process network diagram and presenting an interaction model with which to expose relevant information and appropriate control over the visualisation of the program is significantly challenging. There are therefore a number of differences between the representations of process networks used in the POPed tool from Chapter 4 and the representations used in TC1.

Process networks in TC1 are laid out using a force directed graph algorithm, moving the processes away from each other as they are added to the diagram. This approach conserves and allows the effective use of space as the number of processes program grows and shrinks

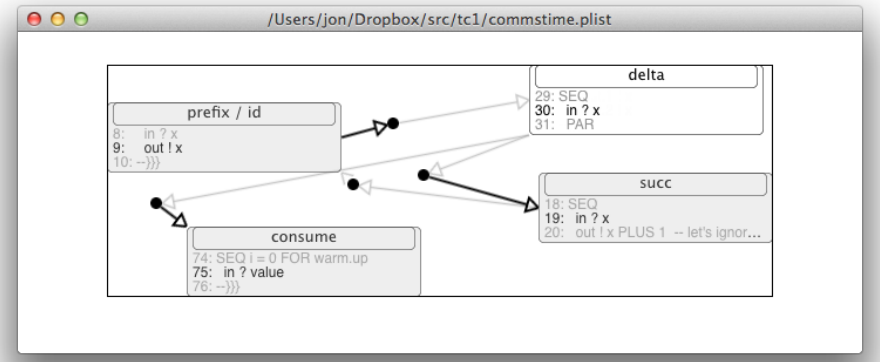


Figure 5.7: *The TC1 trace visualisation tool replaying a trace of* `commstime`*, an occam-pi communication benchmarking program.*

during execution. This leads to layouts and channel routing which are very different to that which programmers would typically create when designing by hand, as connected sets of components would be grouped together by the programmer. This automated approach shares a limitation with the program design tool; the diagram may overflow the canvas if there are too many processes to be rendered, as there is a minimum amount of space in the diagram that the representation of a process may consume.

Processes in TC1 are labeled with their name and contain three lines of source code, including line numbers, truncated to fit for width at around 35 characters to preserve horizontal space. Context switching information encoded in the trace from the VM allows the currently executing process to be tracked. Currently executing processes have a white background and those not currently being executed are shaded in grey (as shown in Figure 5.7).

The trace contains source line information for each state change in the program, indicating the statement currently being executed in each process. This source position information is used to extract three lines of contextual source which is displayed in the process itself. The three lines form a sliding window of context around the current execution position of the process. The middle line of source code is always the current execution position of the process, or in the case of processes not currently executing, where the VM will resume executing. The minimum representation size of a process is restricted by the choice to show this source code and program execution position information as part of the representation.

A process may spawn other processes internally, and in fact this is one of the first actions that happens in most process-oriented programs; the top-level process forks a number of

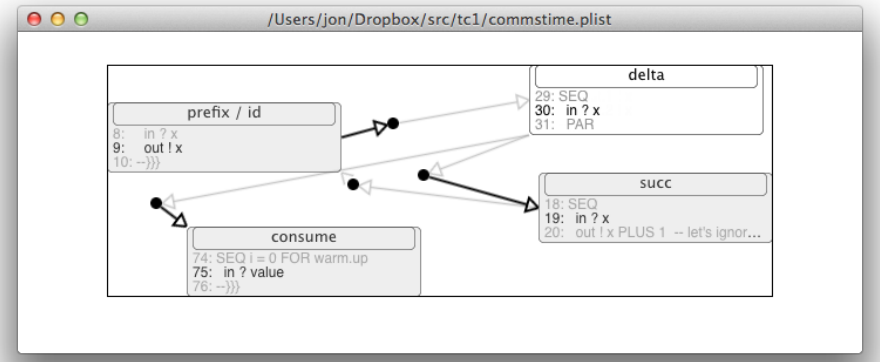


Figure 5.8: *The TC1 trace visualisation tool replaying a trace of* `commstime` *after the delta process has begun to fork parallel subprocesses*

sub processes after initialising their channel connections. This spawning of sub-processes cannot be ignored in the visualisation and given the established constraints on the minimum visual representation size, any significant use of sub-processes would fill the available space. To manage this complexity TC1 draws the internal process network of the process and scales it down, as shown in Figure 5.8. Interaction is built into the tool to allow the user to double click to zoom into the network inside a particular process and double click again to zoom back out. This interaction model allows the processes to be represented at a consistent scale and gives a visual overview of the complexity of the program from the top level.

Channels in TC1 use a representation not seen in the design tool, with a central connection point between two separate arrows forming the channel. This choice has several rationales. The visualisation tool must show that a channel input or output exists before it is fully connected on both sides, using a connection point allows one end of the channel to be represented without the other. The use of a connection point provides additional drawing flexibility, rather than requiring a clear straight path for an arrow to connect the two processes, the connection point may be placed so as to allow routing around other processes.

The visualisation tool must represent the current state of the channel, showing whether either of the processes connected to it are currently blocked on reading from or writing to it. This is achieved in TC1 by highlighting the channel arrow component which is blocked on, as shown in Figure 5.9.

TC1 allows the speed of visualisation to be controlled by use of the left and right arrow keys and the space bar is used to pause the visualisation. This control is provided to facilitate step

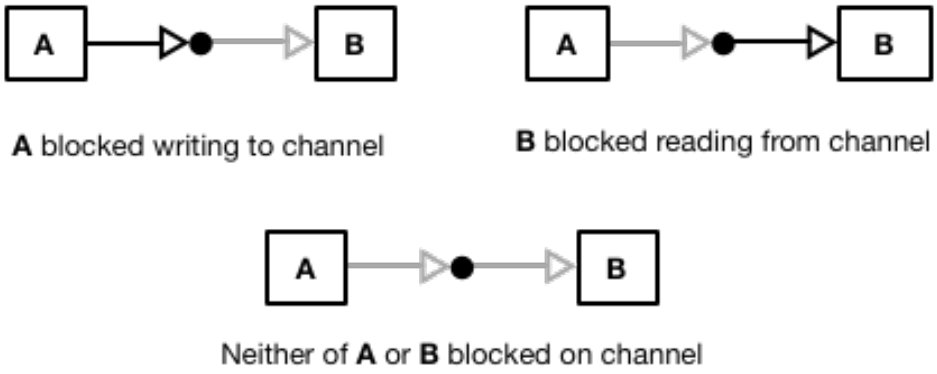


Figure 5.9: *Channel representations and the highlight used to indicate blocking states in the TC1 visualisation tool*

by step walkthroughs of process network behaviour by stepping back and forward over key transition points and speeding up replay when the network stays in a steady state.

This proof of concept implementation was able to present the state visualisations necessary to present the concept of deadlock in process-oriented programs. The example program, as introduced in Section 5.2.2, deadlock is produced in a ring of processes in which a single value is being passed through by introducing a process which discards the value; Once the value is discarded all processes in the network end up waiting to read from their neighbour. The visualised state of this program once it reaches deadlock is shown in Figure 5.10, with all channels blocked on reading from their neighbours.

The channel and value inspection functionalities were not implemented in this tool, meaning the actual values communicated over channels must be reasoned about by following the execution of the code. Adding these features would allow better reasoning for examples where the actual values of traffic on the network are significant. In the deadlock example, only the volume of traffic in the network (i.e. a single message, transitioning to no messages) is used to reason about the behaviour of the program. This proof of concept implementation provided feedback for further refinement of the introspection design. An implementation tied into the program design tool constructed in Chapter 4 would benefit from layout specified by the user and produce an environment in which programs can be both visually designed and debugged. The potential for such an implementation is discussed as further work in Chapter 6.

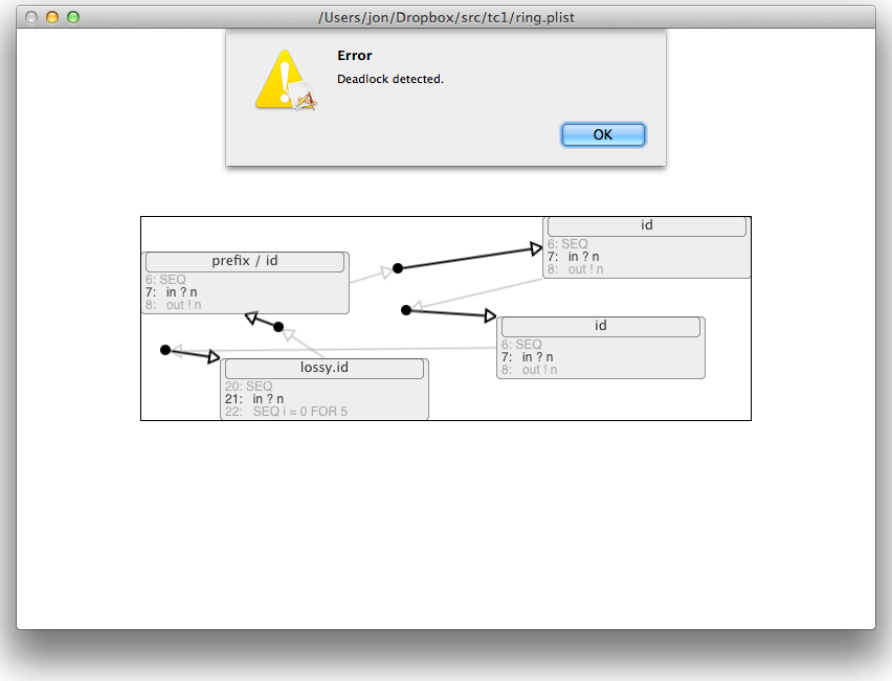


Figure 5.10: *The TC1 visualisation tool showing a program intentionally designed to deadlock and informing the user of the deadlock condition.*

Chapter 6

Conclusions and Further Work

Through re-design and re-implementation of existing robot architectures and hardware interfaces, this thesis has established a number of patterns and principles for the application of process-oriented concurrency to the problem of robot control. This work has aimed to demonstrate the suitability of the process-oriented programming model for robotics, due to a closeness of mapping between the robotics problem domain and process-oriented programming.

This thesis has presented interfaces and concurrent architectures which allow the expression of robot control as message passing data-flow, and that the application of these parallel programming abstractions in the context of small embedded platforms is practical – for example, even on a platform running an interpreted occam-pi run-time response times were equivalent or better than from a native-C implementation. However, while the applicability and suitability of process-oriented programming for robotics has been established for small-scale example programs, this work forms only a basis for investigation into its suitability to larger scale programs or industrial robot control. As detailed in Section 1.3, the work presented in this thesis has already been extended by researchers investigating the applicability of process-oriented robot architectures to larger scale control programs.

This thesis has aimed to facilitate learning and reasoning about program design and behaviour through exploitation of visual representations of process-oriented programs. Building on successes and a rich history of visual programming for both robotics and process-oriented programming, this work aimed to provide tooling suited to both problem domain and programming model. Through the design and implementation of a tool allowing the creation of programs purely through composition of existing components, the feasibility of using top-

level design as an implementation tool for very simple programs using predefined components has been established.

Scaling a visual model beyond small numbers of processes and managing complexity has not been addressed in this tool and remains an issue for the application of these concepts to general purpose process-oriented programming. Significant limitations exist in a purely compositional tool such as the one presented here. Any significant pedagogic experiences of process-oriented programming typically introduce custom sequential logic and require the design of new components; while a visual model has been designed, bridging the gap between textual, sequential logic and high level design is necessary for such a tool to be generally applicable. Design constraints for extending visual programming tools to capture the dynamics of run-time process and network creation (afforded by advanced language features in occam-pi) have been raised and are fertile ground for future research; this area has significant application to complex emergent systems and modelling as well as robotics.

This thesis has also demonstrated the application of visual representations to elucidating the runtime behaviour (e.g process state change and channel history) of a process-oriented program, allowing programmers to reason about program behaviour using a model closer to that used its design. A program introspection tool has been presented which demonstrates the collection and presentation of run-time state through visualisations that reflect the process network diagrams used in program design. The utility of this tool in explaining and reasoning about concurrency errors in process-oriented programs has been demonstrated.

The introspection tool presented in this thesis is limited in that it offers only post-mortem visualisation of a program and a number of processing steps are needed to generate the visualisations and animations of the program. To be integrated into the development cycle of a process-oriented program, such a tool would need to be able to control and reflect live execution of the program, together with the ability to visualise larger systems, with hundreds, thousands or even millions of processes in a meaningful manner. While some facilities have been designed and implemented for the management of complexity, through the hiding or showing the context of a particular process, where hierarchies are flat and there are large numbers of processes at the same level these techniques are less effective.

6.1 Future Work: Process Architectures for Robotics

This section details extensions to the work on process-oriented robotics architectures presented in Chapter 3. These process-oriented robotics architectures form a basis from which further architectures or platforms can be added and evaluated. Additional programming language features or libraries may be employed in further extensions to the work to investigate alternate or specialised implementations of process-oriented robot architectures for particular hardware platforms or computationally difficult tasks.

6.1.1 Hybrid Architectures

Work reported in this thesis to apply robot architectures in the process-oriented model focuses on behavioural robotics, but there is significant use of hybrid architectures in mobile robots. Hybrid architectures such as Connell's SSS [Con92] and Gat's ATLANTIS [Gat92] are candidates for implementation as process architectures to fully evaluate the abilities of a process-oriented model in applying across robotics architectures.

6.1.2 Platforms

The types of robotics platform available have expanded considerably since the outset of this work; small platforms capable of flight are now available commercially and the next logical step beyond small robots for motivation would be these kinds of unique problem domains. Armstrong et al. have successfully explored the use of the Transterpreter and occam-pi on the Arduino micro-controller for flight control on a glider, a problem which emphasises the importance of response speed and feedback control [APBSJ11]. The Arduino support libraries have already been applied successfully for robotics by a team entering the Trinity College Fire Fighting Home Robot contest [HCM+12].

More recently, the Raspberry Pi, an ARM-based System on a Chip (SoC) embedded board capable of running a full Linux operating system for $25 has attracted significant attention for lowering the barriers of entry to computing and use in education [Ras13]. Commodity platforms like the Raspberry Pi and Arduino are excellent platforms to provide distributions and tools for exploring hardware control and robotics due to their widespread availability and the hobbyist communities surrounding them. Small computers and embedded boards suitable for building mobile robots have also become more available; the Arduino embedded

board has been a fruitful target of more recent work by Jadud et. al. to provide a platform library called Plumbing for introductory electronics and embedded systems programming with the Arduino [JJK+13]. These embedded boards designed for hardware experimentation present a target for application of the visual programming environment detailed in this thesis; enabling a cross-domain constructivist approach to both hardware and software design.

6.1.3 Dynamic occam-pi language features

The process-oriented implementations of established robot architectures detailed in this thesis have, in the most part, not made use of the dynamic language features present in occam-pi. This has been due to the relative difficulty of implementing the underlying support for these instructions when porting the virtual machine runtime to environments without an underlying operating system. Use of these functions enables dramatic performance optimisation in cases where large data is being communicated between processes; this has been the only application to date in providing acceptable performance when handling camera frames on the Surveyor SRV-1.

In occam-pi it is possible to create and connect channels and processes at run time. This dynamic reconfiguration may be combined with existing robot architectures to allow them to adapt to environmental conditions or changes in robot hardware configuration. Application of dynamic reconfiguration may permit relatively static pre-defined hierarchical control systems to be controlled at a higher level by reactive, behavioural elements. There are challenges in the graphical construction of such programs, as the changing process network would need to be represented in a number of states at design-time.

The application of dynamic channel and process creation language features, introduced in occam-pi, may be applied to the design of hardware interface processes. Current approaches use processes which supply a channel interface providing a feed of incoming sensor data, or accept a client/server command protocol (as described in Section 3.1.6. Designing process interfaces which require subscription to a feed of values from a sensor process delivered at a specific rate would make the development of efficient, event-driven systems with many levels of behaviour more practical. The effective use of an event driven model would provide power saving opportunities; when using a sparse communication model the system itself could sleep in the gaps between responses, lengthening battery life.

A similar practice for control of motors and camera access, where multiple sets of behaviour can share access through an arbiter would also help with effectively *multiplexing* input and

output to the platform from the highly concurrent control program. Investigation into the use of processes for hardware abstraction, via the use of PROTOCOL inheritance could allow effective component and program re-use between different robotics platforms.

6.1.4 Parallel Languages and Robot Control Frameworks

In applying a parallel programming language to robotics, there are a number of other languages with which it would be useful to compare and contrast occam-pi. For small scale robotics, Gostai SAS's Universal Real-time Behaviour Interface (URBI) supplies a universal platform for a number of robots which provides a parallel, event-driven programming language [Gos08]. URBI is supplied with a graphical programming environment based on state diagrams, and has a number of features for concurrent execution of code. A less educational and more industrial platform is supplied by Evolution Robotics in the form of the Evolution Robotics Software Platform (ERSP) [MOP05]. ERSP uses Python program code or a graphical behaviour composer to build systems from predefined components, specifically a range of vision-processing, mapping and localisation algorithms supplied with the package itself.

As presented in Section 3.4, the Robot Operating System (ROS) has emerged as a popular middleware framework for integrating disparate software components for robotics. The creation of a ROS client library for a process-oriented language, such as occam-pi, would permit the creation of systems in ROS which are able to take advantage of fine grained concurrency. Using ROS to interface to third party libraries such as Player or OpenCV would avoid the foreign function and wrapping complexities detailed in Section 3.1.3, providing a message passing interface (albeit asynchronous) to these libraries from the process-oriented environment.

Other parallel programming languages are also a target for examination, as the process-oriented paradigm can be applied in languages outside of occam-pi. Ada has been used for real-time robotic control and shown promise as a teaching language on small robots in the classroom [SB94, FME01, Fag03], making it an excellent choice to examine for design paradigms we would like to replicate using occam-pi or as a target for code generation. Erlang has had minor usage in autonomous mobile robotics, but has not seen significant use due to the requirements of its runtime environment. The Actor model of asynchronous process communication can be replicated on top of occam-pi, and as such it is useful to examine approaches taken so far [San07]. Even in the absence of an Erlang runtime suitable for use on small robot platforms, the principles of the Actor model could be explored using the

Transterpreter runtime and asynchronous occam-pi programs.

6.1.5 Network Distribution

Many small robotics platforms are network aware and can be connected to a host system via Bluetooth [Blu07], Zigbee [IEE03] or WiFi [IEE99]. These connections function traditionally to allow tele-operation by control programs written for and running on the host system, or the presentation of an Operator Control Unit (OCU) visualising the sensor data and providing a control panel for the actuators on the platform. Previous explorations of occam-pi robotics have traditionally executed the control program on the robot itself, allowing it to operate independently and taking advantage of the very low memory and storage requirements of the virtual machine runtime.

As the robot control programs we design are already constructed of communicating parallel processes there is potential for these processes to be distributed across hosts without significant restructuring of the program to use an external middleware layer. As discussed in the previous section, a number of robotics frameworks provide middleware to facilitate the co-ordination of independent processes to form a single system. The advantage of using a network library in occam-pi is consistency of model; all communications inside the program and between the components can be reasoned about as synchronous channel communications.

Network distribution of process network components allows latency sensitive processing (such as processing image frames) to be done on the robot platform, and more complex, less temporal, behaviours to be computed on the host system.

The KRoCoccam-pi runtime contains a library called Pony, which allows occam-pi channels to be routed over TCP/IP networks between clusters of computers [Sch06]. Pony requires a separate server allowing the nodes to find each other and is relatively heavyweight, we would prefer to take advantage of the fact that we are using a virtual machine runtime. We envisage that a more straightforward method for linking a single host to a robot via stub processes executing concurrently with the user's programs on the robot and desktop which accept network communications and marshal them over the network could be created, or that the virtual machine runtime could be enhanced to support network transparency.

In terms of a visual tool, being able to graphically move processes between the two different targets (host computer or robot platform), reconfiguring the distribution of the network would be a large step forward for examining distribution patterns. In combination with

debugging information exposed from the runtime environment about the number and size of channel communications, the load and latency of particular connections could be annotated, allowing different distribution patterns to be evaluated.

6.1.6 Multi-core and Many-core Robotics

The Transterpreter Virtual Machine is not multi-threaded, making the use of occam-pi on the TVM concurrent but not parallel. Given the availability of multi-core and many-core embedded boards with low power consumption and appropriate form factors for small robots it would be interesting to write control programs which take full advantage of robot platforms equipped with hardware parallelism. Multi-core algorithms for occam-pi scheduling have already been implemented in CCSP by Ritson [RSB12] and could be reused in enhancing the Transterpreter with support for scheduling processes across multiple cores.

6.2 Future Work: Visual Design and Debugging

This section details extensions to the visual programming demonstrator tool, POPed, shown in Chapter 4 and the program introspection tool, *TC1*, presented in Chapter 5

6.2.1 Visual Programming

The POPed visual programming demonstrator tool described in Chapter 4 is currently restricted to composing process networks; it is only a starting point in the development of a pedagogic environment for process-oriented programming. The aggregation of the features of past tools and opinionated choices made whilst designing the demonstrator environment represent only a single data point in forming a pedagogically sound programming environment. The use of feedback or a formal user study from the classroom would enable the development of a pedagogic tool for teaching parallel programming based on sound principles. Adapting the environment to make it useful in more problem domains, providing different sets of processes and access to the various libraries outside of robotics would widen the environment's application beyond robotics.

Effectively combining both design and debugging would provide an end-to-end visual environment for process-oriented programming. As discussed in Section 4.4.8, library choices have

hampered implementation of the tool and continue to make the tool fragile. Re-implementing the program builder using Javascript and HTML5 would significantly widen the reach of the software and allow embedding of process network diagrams in webpages.

6.2.2 Formal Visual Language Design

While consideration has been made to existing practices in drawing diagrams of process-oriented programs, there is significant additional scope in designing a visual language based directly on programmers' mental models. The work of Petre and Blackwell in relating cognitive processes to visual programming methodologies [PB99] and Whitney and Blackwell's later work surveying the cognitive effects of visual programming in LabVIEW [WB01] provide a background to work at the intersection of cognitive processes and visual languages. Building a visual programming model for process-oriented programming based on cognition research and mental models may yield improvements in representation beyond those organically discovered by the evolution of diagrams and the study conducted in Section 4.1. Formally documenting the semantics of process network diagrams would be of aid to the process-oriented programming community, giving a fixed point of reference and allowing better re-use of software components and tools.

6.2.3 Code Editing

The demonstrator environment, while allowing composition of processes already defined in its toolbox, does not allow the programmer to create new processes from scratch. A number of tools identified in Section 4.2.1, most notably Scratch, use an entirely visual syntax for sequential logic and program composition. While the application of a framework such as OpenBlocks [Roq07] would permit the implementation of occam-pi syntax as a visual language, this would leave programmers with two visual programming models to interact with which have little commonality.

The ability to express sequential logic and algorithms is a strength of textual code. Rather than implement a second visual language, is desirable to provide basic text editing features such that new processes may be defined within the POPed environment. While adding this editing ability, it is desirable to constrain and structure the interface of the process to the rest of the program and limit the scope of textual code visible at a given time.

The current practice in process-oriented programs written in occam-pi is for all processes to

be contained in a single file, with the top level process at the bottom of the file. The default process for execution being at the bottom means process definitions are spatially and logically close to one another. Inspired by the treatment of Java classes in Kölling's BlueJ [KQPR03], editing processes in isolation and using a structured UI for defining process interfaces would emphasise the isolation of components within the process-oriented model.

6.2.4 Code Generation Advancements

As the visual environment primarily deals with dragging fully formed components together and allowing connections between compatible interfaces, given a common communication library between different process-oriented languages, blocks could be implemented in a number of different programming languages and composed into a single program. A common wire format for exchange of messages on channels has been previously suggested, for both local and networked communication between process-oriented programs. This would open up interesting possibilities of swapping block implementations between languages; if a robot program was to be distributed between a host computer and some activity on the robot itself, blocks moved to the robot could use occam-pi implementations where the host used JCSP implementations. Having this flexibility built in would make moving processes between host and robot to achieve performance straightforward, given tools to monitor performance and design effective distribution strategies as discussed below.

6.2.5 Introspection

In Chapter 5 this thesis presents the design and implementation of a tool supporting the run-time state inspection, or 'introspection' of process-oriented programs. Development of this introspection tool required the design and implementation of underlying support in the virtual machine to expose control over program execution and program state. This underlying support facilitates the creation of tools or additional software to observe and interact with program execution for other purposes, such as visualising program performance for optimisation or allowing live creation of programs.

Program Optimisation

To be able to improve the performance of a system, it is first important to be able to observe its behaviour and measure its performance. Tools for occam on the Transputer had the ability to colour channels depending on the size of data or frequency of communication that occurred along them. This was important in hardware, a channel could become a bottleneck for the entire program. This type of performance data and traffic analysis would be generally useful for modern occam-pi programs, to identify communication patterns which make certain subsections of the program candidates for distribution across multiple physical hosts. If a channel link communicated very infrequently with quite large chunks of data it would be a good candidate to be sent across a network. Conversely, a channel communicating very frequently with very small amounts of data would be an indication that two components need low latency communication, and shouldn't be distributed over the network.

Bibliography

[Abe09] Hal Abelson. App Inventor for Android. http://googleresearch.blogspot.co.uk/2009/07/app-inventor-for-android.html, July 2009. 110

[Ade12] Adept MobileRobots. ARIA - MobileRobots Research and Academic Customer Support. http://robots.mobilerobots.com/wiki/ARIA, March 2012. 60

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. 27

[ABV07] Otto J. Anshus, John Markus Bjørndalen, and Brian Vinter. PyCSP - Communicating Sequential Processes for Python. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 229–248, Amsterdam, The Netherlands, jul 2007. IOS Press. 29, 136

[Arc11] Katrina Archer. Practical Game Architecture for Multi-core Systems. http://software.intel.com/sites/billboard/article/practical-game-architecture-multi-core-systems, November 2011. 26

[Ark87] Ronald C. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *IEEE International Conference on Robotics and Automation*, volume 4, pages 264–271, Mar 1987. 87

[Ark98] Ronald C. Arkin. *Behavior-based Robotics*. MIT Press, Cambridge, MA, USA, 1998. 86

[AB97] Ronald C. Arkin and Tucker Balch. AuRA: Principles and Practice in Review. *Journal of Experimental and Theoretical Artificial Intelligence*, 9:175–189, 1997. 87

[ARM12] ARM Ltd. Cortex-A9 Processor. `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`, March 2012. 26

[APBSJ11] Ian Armstrong, Michael Pirrone-Brusse, A Smith, and Matthew C. Jadud. The Flying Gator: Towards Aerial Robotics in occam-pi. In Peter H. Welch, Adam T. Sampson, Jan B. Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 329–340, jun 2011. 171

[AVWW93] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang, 1993. 29, 40

[ABL92] Mats Aspnäs, Ralph-Johan Back, and Thomas Långbacka. Millipede: A programming environment providing graphical support for parallel programming. In *Proceedings of the European Workshop on Parallel Computing*, pages 236–247, 1992. 114

[Bar02] David J. Barnes. Teaching Introductory Java through LEGO MINDSTORMS Models. In *Proceedings of the 33rd SIGCSE technical symposium on computer science education*, pages 147–151. ACM, February 2002. 44

[Bar05] David J. Barnes. ROBOLAB-based Key Stage 3 Programming Materials. `http://www.cs.kent.ac.uk/people/staff/djb/robolab/`, October 2005. 111

[Bar00] Frederick R. M. Barnes. Blocking System Calls in KRoC/Linux. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 155–178, September 2000. 59

[Bar05] Frederick R. M. Barnes. Interfacing C and occam-pi. 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 Series*, pages 249–260, IOS Press, The Netherlands, September 2005. IOS Press. 59

[BW96] David J. Beckett and Peter H. Welch. A Strict occamDesign Tool. In Chris R. Jesshope and A. Shasha Shafarenko, editors, *Proceedings of UK Parallel '96*, pages 53–69, Guildford, UK, July 1996. Springer-Verlag, London. ISBN 3-540-76068-7. 115

[BCHS00] Michael Bedy, Steve Carr, Xianlong Huang, and Ching-Kuang Shene. A visualization system for multithreaded programming. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 1–5, New York, NY, USA, 2000. ACM Press. 52

[Beg96] Andrew Begel. LogoBlocks: A Graphical Programming Language for Interacting with the World. Technical report, MIT Media Laboratory, Cambridge, MA, USA, May 1996. 109

[BK07] Andrew Begel and Eric Klopfer. Starlogo TNG: An introduction to game development. *Journal of E-Learning*, 2007. 109

[BA07] Mordechai Ben-Ari. Teaching Concurrency and Nondeterminism with Spin. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '07, pages 363–364, New York, NY, USA, 2007. ACM. 50

[BAK99] Mordechai Ben-Ari and Yifat Ben-David Kolikant. Thinking parallel: the process of learning concurrency. In *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 13–16, New York, NY, USA, 1999. ACM Press. 50

[BE02] Alan F. Blackwell and Yuri Engelhardt. A meta-taxonomy for diagram research. In Michael Anderson, Bernd Meyer, and Patrick Olivier, editors, *Diagrammatic Representation and Reasoning*, pages 47–64. Springer London, 2002. 104

[BKM+12] Douglas Blank, Jennifer S. Kay, James B. Marshall, Keith O'Hara, and Mark Russo. Calico: a multi-programming-language, multi-context framework designed for computer science education. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 63–68, New York, NY, USA, 2012. ACM. 43

[BKMY03] Douglas Blank, Deepak Kumar, Lisa Meeden, and Holly Yanco. Pyro: A python-based versatile programming environment for teaching robotics. In *Journal of Educational Resources in Computing (JERIC)*, pages 1–15, 2003. 43

[Blu07] Bluetooth SIG. Bluetooth Core Specification v2.1+ EDR. `http://bluetooth.com/Bluetooth/Technology/Building/Specifications/Default.htm`, July 2007. 174

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268, New York, NY, USA, 2005. ACM Press. 29, 63

[Bra86] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, MA, USA, 1986. 69, 129

[BH75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, 1975. 15

[BGL05] Jan F. Broenink, Marcel A. Groothuis, and Geert K. Liet. gCSP occamCode Generation for RMoX. In *Communicating Process Architectures 2005*, pages 375–383, sep 2005. 9, 116, 117

[BJ04] Jan F. Broenink and Dusko S. Jovanovic. Graphical Tool for Designing CSP Systems. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 233–252, September 2004. 116

[Bro85] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical report, MIT, Cambridge, MA, USA, 1985. 130

[Bro86] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986. 72

[Bro89] Rodney A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. Technical report, MIT, Cambridge, MA, USA, 1989. 73, 82

[Bro07] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, jul 2007. 29

[BS09] Neil C. C. Brown and Marc L. Smith. Relating and Visualising CSP, VCR and Structural Traces. In 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*, pages 89–103, nov 2009. 154

[Bun09] David P. Bunde. A short unit to introduce multi-threaded programming. *Journal of Computing Sciences in Colleges*, 25(1):9–20, October 2009. 21

[CMS03] Steve Carr, Jean Mayo, and Ching-Kuang Shene. Threadmentor: a pedagogical tool for multithreaded programming. *J. Educ. Resour. Comput.*, 3(1):1, 2003. 52

[Con89] Jonathan H. Connell. A colony architecture for an artificial creature. Technical report, MIT Artificial Intelligence Laboratory, Cambridge, MA, USA, 1989. 82, 84

[Con92] Jonathan H. Connell. SSS: a hybrid architecture applied to robot navigation. In *Robotics and Automation, 1992. Proceedings of the 1992 IEEE International Conference on*, pages 2719–2724 vol.3, 1992. 171

[CRS98] Philip T. Cox, Christopher C. Risley, and Trevor J. Smedley. Toward concrete representation in visual languages for robot control. *Journal of Visual Languages & Computing*, 9(2):211–239, 1998. 108

[CS98] Philip T. Cox and Trevor J. Smedley. Visual programming for robot control. In *VL '98: Proceedings of the IEEE Symposium on Visual Languages*, page 217, Washington, DC, USA, 1998. IEEE Computer Society. 108

[DHWN94] Mark Debbage, Mark Hill, Sean Wykes, and Denis Nicole. Southampton's portable occam compiler (spoc). In *Proceedings of WoTUG-17: Progress in Transputer and occam Research, volume 38 of Transputer and occam Engineering*, pages 40–55. IOS Press, 1994. 31

[Dij87] Edsger W. Dijkstra. Twenty-eight years (EWD1000). Circulated privately, January 1987. 49

[Dim09] Damian J. Dimmich. *A Process Oriented Approach to Solving Problems of Parallel Decomposition and Distribution*. PhD thesis, University of Kent, Canterbury, Kent, England, June 2009. v, 26

[DJo5] Damian J. Dimmich and Christan L. Jacobsen. A Foreign Function Interface Generator for occam-pi. In Jan F. Broenink, Herman W. Roebbers, Johan P. E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures 2005*, pages 235–248, Amsterdam, The Netherlands, September 2005. IOS Press. 59

[DJJo6] Damian J. Dimmich, Christian L. Jacobsen, and Matthew C. Jadud. A Cell Transterpreter. In Peter H. Welch, Jon M. Kerridge, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2006*, pages 215–224, Amsterdam, The Netherlands, September 2006. IOS Press. 57

[ECRoo] Ben Erwin, Martha Cyr, and Chris Rogers. LEGO engineer and ROBOLAB: Teaching engineering with LabVIEW from kindergarten to graduate school. *International Journal of Engineering Education*, 16(3):2000, 2000. 9, 110, 111

[Extoo] Chris Exton. Elucidate: a tool to aid comprehension of concurrent object oriented execution. In *ITiCSE 'oo: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 33–36, New York, NY, USA, 2000. ACM. 51

[Fago3] Barry S. Fagin. Ada/Mindstorms 3.0. *Robotics & Automation Magazine, IEEE*, 10(2):19–24, 2003. 44, 173

[FMEo1] Barry S. Fagin, Laurence D. Merkle, and Thomas W. Eggers. Teaching computer science with robotics using ada/mindstorms 2.0. *Ada Lett.*, XXI(4):73–78, 2001. 173

[Feko9] Alan D. Fekete. Teaching about threading: where and what? *SIGACT News*, 40(1):51–57, February 2009. 21

[FMNo8] Paul Fitzpatrick, Giorgio Metta, and Lorenzo Natale. Towards long-lived robot genes. *Robotics and Autonomous Systems*, 56(1):29–45, January 2008. 92

[Foroo] Formal Systems (Europe) Ltd., 3, Alfred Street, Oxford. OX1 4EH, UK. *FDR2 User Manual*, May 2000. 51

[Gai86] Jason Gait. A probe effect in concurrent programs. *Softw. Pract. Exper.*, 16(3):225–233, March 1986. 147, 153

[Gat92] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the tenth national conference on Artificial intelligence*, AAAI'92, pages 809–815. AAAI Press, 1992. 171

[GVH03] 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. 45, 61, 92

[GIL+95] Jim Gindling, Andri Ioannidou, Jennifer Loh, Olav Lokkebo, and Alexander Repenning. LEGOsheets: a rule-based programming, simulation and manipulation environment for the LEGO Programmable Brick. In *VL '95: Proceedings of the 11th International IEEE Symposium on Visual Languages*, page 172, Washington, DC, USA, 1995. IEEE Computer Society. 108

[GvN63] Herman H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument, part ii. In A.H. Traub, editor, *John von Neumann, Collected Works Volume V, Design of computers, theory of automata and numerical analysis*, page 30. Pergamon Press, Oxford, 1963. 9, 103

[Goo12] Google, Inc. The Go Programming Language. `http://golang.org`, March 2012. 29

[Gor08] Howard Gordon. Surveyor SRV-1 Blackfin Robot. `http://www.surveyor.com/`, July 2008. 65

[Gos08] Gostai SAS. URBI: Universal Real-time Behavior Interface. `http://www.gostai.com/`, August 2008. 92, 173

[Gow94] Jay Gowdy. Sausages: Between planning and action. Technical Report CMU-RI-TR-94-32, Robotics Institute, Pittsburgh, PA, September 1994. 90

[GP96] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996. 16

[Gro11] Thomas R. Gross. Breadth in depth: a 1st year introduction to parallel programming. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 435–440, New York, NY, USA, 2011. ACM. 21

[HCM+12] Kathryn Hardey, Eren Corapcioglu, Molly Mattis, Mark Goadrich, and Matthew C. Jadud. Exploring and evolving process-oriented control for real and virtual fire fighting robots. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 105–112, New York, NY, USA, 2012. ACM. 171

[Har94] Stephen J. Hartley. Animating operating systems algorithms with xtango. In *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*, pages 344–348, New York, NY, USA, 1994. ACM Press. 53

[HE91] Michael T. Heath and Jennifer A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, September 1991. 52

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. 29

[Hil02] Gerald H. Hilderink. A Graphical Modeling Language for Specifying Concurrency based on CSP. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 255–284, September 2002. 116

[Hoa85] Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985. 15, 28, 30

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997. 50

[HPR89] Housheng Hu, Penny J. Probert, and Bobby S. Y. Rao. A Transputer Architecture for Sensor-based Autonomous Mobile Robots. In *Intelligent Robots and Systems '89. The Autonomous Mobile Robots and Its Applications. IROS '89. Proceedings., IEEE/RSJ International Workshop on*, pages 297 –303, September 1989. 41

[IEE01] IEEE. IEEE standard test access port and boundary scan architecture. *IEEE Standard 1149.1-2001*, pages 1–212, 2001. 162

[IEE99] IEEE 802.11 Working Group. Wireless LAN medium access control (MAC) and physical layer (PHY) specification. Standard, IEEE, 1999. 174

[IEE03] IEEE 802.15 Working Group. Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (WPANs). Standard 802.15.4-2003, IEEE, 2003. 174

[INM84] INMOS Limited. *occam2 Reference Manual*. Prentice Hall, 1984. ISBN: 0-13-629312-3. 15

[Int04] Intel Corporation. Architecting the Era of Tera. Technical report, Intel Research and Development, 2004. 26

[Jac06] Christian L. Jacobsen. *A Portable Runtime for Concurrency Research and Application*. PhD thesis, University of Kent, Canterbury, Kent, England, December 2006. 10, 19, 32, 57, 120, 157, 158

[JJ04] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Dr. Ian R. East, Prof David Duce, Dr Mark Green, Jeremy M. R. Martin, and Prof. Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering*, pages 99–106. IOS Press, Amsterdam, September 2004. 17, 32

[JJ05] 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. 21, 41, 62

[JJ07] 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. 7, 45, 46, 48, 61

[JCS03] Matthew C. Jadud, Brooke N. Chenoweth, and Jacob Schleter. Little languages for little robots. *PPIG*, 2003. 44

[JJK+13] Matthew C. Jadud, Christian L. Jacobsen, Omer Kilic, Adam T. Sampson, and Jonathan Simpson. Concurrency.cc - parallel programming for the rest of us. `http://concurrency.cc`, July 2013. 172

[JJRS08] Matthew C. Jadud, Christian L. Jacobsen, Carl G. Ritson, and Jonathan Simpson. Safe parallelism for robotic control. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, November 2008. 20

[JJS10] Matthew C. Jadud, Christian L. Jacobsen, and Adam T. Sampson. Plumbing for the Arduino. `http://concurrency.cc/pdf/plumbing-for-the-arduino.pdf`, January 2010. 21

[JSJ08] Matthew C. Jadud, Jonathan Simpson, and Christian L. Jacobsen. Patterns for programming in parallel, pedagogically. *SIGCSE Bulletin*, 40(1):231–235, 2008. 19, 93

[JE88] Dewi I. Jones and Paul M. Entwistle. Parallel computation of an algorithm in robotic control. In *CONTROL 88., IEE International Conference on*, pages 438–443, April 1988. 40

[Ker84] Brian W. Kernighan. *Unix for Beginners*. Bell Laboratories, 1984. 153

[KS84] Jon M. Kerridge and Dan Simpson. Three solutions for a robot arm controller using pascal-plus, occam and edison. *Software: Practice and Experience*, 14(1):3–15, 1984. 40

[KQPR03] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4), December 2003. 18, 177

[KBB+08] Deepak Kumar, Douglas S. Blank, Tucker R. Balch, Keith J. O'Hara, Mark Guzdial, and Stewart Tansley. Engaging computing students with ai and robotics. In *AAAI Spring Symposium: Using AI to Motivate Greater Participation in Computer Science*, pages 55–60. AAAI, 2008. 44

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, April 1987. 154

[Lyn] Lynxmotion, Inc. AH3-R Walking Robot. http://www.lynxmotion.com/Category.aspx?CategoryID=92. 69

[Mae89a] Pattie Maes. The dynamics of action selection. In *IJCAI*, pages 991–997, 1989. 85

[Mae89b] Pattie Maes. How to do the right thing. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1989. 85

[MBK+04] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. Scratch: A sneak preview. In *C5 '04: Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*, pages 104–109, Washington, DC, USA, 2004. IEEE Computer Society. 109

[Mar07] Fred G. Martin. Real robots don't drive straight. In *AAAI Spring Symposium: Semantic Scientific Knowledge Integration*, pages 90–94. AAAI, 2007. 44

[MW97] Jeremy M.R. Martin and Peter H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4), August 1997. 69

[May83] David May. occam. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983. 30

[MTWS78] David May, Richard J.B. Taylor, and Colin Whitby-Strevens. EPL - An Experimental Language for Distributed Computing. In *Trends and Applications 1978*, Gaithersburg, Maryland, May 1978. National Bureau of Standards. 30

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, December 1989. 147

[Mic08] Microsoft Corporation. Microsoft Robotics Studio: VPL Introduction. http://msdn.microsoft.com/en-us/library/bb483088.aspx, 2008. 9, 112, 113

[MIT02] MIT Media Lab. Introduction to Logo Blocks. http://llk.media.mit.edu/projects/cricket/doc/help/logoblocks/startingwithlogoblocks.htm, June 2002. 9, 110

[Mob] Mobile Robots, Inc. Pioneer 3-DX Mobile Robot. http://www.activrobots.com/ROBOTS/p2dx.html. 46

[Moo65] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965. 26

[Moo99] James Moores. CCSP - A Portable CSP-Based Run-Time System Supporting C and occam. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 147–169, mar 1999. 31

[MOP05] Mario E. Munich, Jim Ostrowski, and Paolo Pirjanian. ERSP: a software platform and architecture for the service robotics industry. *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 460–467, August 2005. 173

[NWN88] Fazel Naghdy, C. K. Wai, and Golshah Naghdy. Multiprocessing control of robotic systems. In *Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on*, volume 2, pages 975–977, 1988. 40

[Nee08] John Neeson. occam-pi for Multiple Robotic Systems. Master's thesis, University of York, May 2008. 21, 82

[Nil84] Nils J. Nilsson. Shakey the robot. Technical Report 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Apr 1984. 35

[Nog04] Marcus L. Noga. The brickOS Home Page. `http://brickos.sourceforge.net`, June 2004. 63

[oCC13] ACM/IEEE-CS Joint Task Force on Computing Curricula. Computer Science Curricula 2013. Technical report, ACM Press and IEEE Computer Society Press, December 2013. 26

[O'N87] Conor O'Neil. The TDS occam 2 debugging system. In Traian Muntean, editor, *OUG-7: Parallel Programming of Transputer Based Machines*, pages 9–14, sep 1987. 156

[Pap80] Seymour Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, 1980. 44

[Pap86] Seymour Papert. *Constructionism: A new opportunity for elementary science education*. Massachusetts Institute of Technology, Media Laboratory, Epistemology and Learning Group, 1986. 44

[Par13] Parallela. Parallela Computer Specifications. `http://www.parallella.org/board/`, June 2013. 27

[Pat81] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1981. 43

[PB99] Marian Petre and Alan F. Blackwell. Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51(1):7–30, 1999. 176

[Poo96] Michael D. Poole. occam-for-all – Two Approaches to Retargeting the INMOS occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments – Proceedings of WoTUG 19*, pages 167–178, Nottingham-Trent University, UK, March 1996. World occam and Transputer User Group, IOS Press, Netherlands. ISBN 90-5199-261-0. 31

[Pos09] Jeremy C. Posso. occam-pi for Behaviour-based Robotics. Master's thesis, The University of York, 2009. 21, 81, 82, 158

[PSST11] Jeremy C. Posso, Adam T. Sampson, Jonathan Simpson, and Jon Timmis. Process-Oriented Subsumption Architectures in Swarm Robotic Systems. In Peter H. Welch, Adam T. Sampson, Jan B. Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, volume 69 of *Concurrent Systems Engineering*, pages 303–316. IOS Press, June 2011. 8, 20, 83

[Puc91] Miller S. Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, Fall 1991. 118

[Puc96] Miller S. Puckette. Pure Data: another integrated computer music environment. In *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996. 118

[QCG+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009. 92

[Ras13] Raspberry Pi Foundation. Raspberry Pi. http://www.raspberrypi.org, June 2013. 171

[RAA+07] Charles Reinholtz, Thomas Alberi, David Anderson, Andrew Bacha, Cheryl Bauman, Stephen Cacciola, Patrick Currier, Aaron Dalton, Jesse Farmer, Ruel Faruque, et al. DARPA Urban Challenge Technical Paper. http://archive.darpa.mil/grandchallenge/TechPapers/Victor_Tango.pdf, April 2007. 111

[RS95] Alexander Repenning and Tamara Sumner. Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *IEEE Computer*, 28(3):17–25, 1995. 108

[RMSS96] Mitchel Resnick, Fred G. Martin, Randy Sargent, and Brian Silverman. Programmable bricks: toys to think with. *IBM Systems Journal*, 35(3-4):443–452, 1996. 44, 109

[RSB12] Carl G. Ritson, Adam T. Sampson, and Frederick R.M. Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727–740, June 2012. 175

[RS08] Carl G. Ritson and Jonathan Simpson. Virtual Machine-based Debugging for occam-π. In Peter H. Welch, Susan Stepney, Fiona A. C. Polack, Frederick R. M. Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, pages 293–307, Amsterdam, The Netherlands, September 2008. IOS Press. 20, 162

[Roq07] Ricarose V. Roque. Openblocks: an extendable framework for graphical block programming systems. Master's thesis, Massachusetts Institute of Technology, 2007. 110, 176

[Ros95] Julio K. Rosenblatt. DAMN: A distributed architecture for Mobile Navigation. In H. Hexmoor and D. Kortenkamp, editors, *proceedings of the 1995 AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*, Menlo Park, CA, March 1995. AAAI Press. 90

[Sam06] Adam T. Sampson. What is Love? https://www.cs.kent.ac.uk/research/groups/sys/wiki/LOVE, September 2006. 9, 118, 119

[Sam08] Adam T. Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, Computing, University of Kent, CT2 7NF, September 2008. 9, 93, 104, 105

[San07] Corrado Santoro. An Erlang framework for Autonomous Mobile Robots. In *Erlang '07: Proceedings of the 2007 SIGPLAN workshop on Erlang Workshop*, pages 85–92, New York, NY, USA, 2007. ACM. 173

[SSKF95] Christian Scheidler, Lorenz Schäfers, and Ottmar Krämer-Fuhrmann. Software engineering for parallel systems: the TRAPPER approach. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 349, Washington, DC, USA, 1995. IEEE Computer Society. 9, 114, 115

[Sch06] Mario Schweigler. *A Unified Model for Inter- and Intra-processor Concurrency*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, UK, Canterbury, Kent, CT2 7NF, United Kingdom, August 2006. 174

[SGS95] SGS-THOMSON Microelectronics Limited. *occam 2.1 reference manual*, May 1995. 30

[SJ08] Jonathan Simpson and Christian L. Jacobsen. Visual Process-oriented Programming for Robotics. In Peter H. Welch, Susan Stepney, Fiona A. C. Polack, Frederick R. M. Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, pages 365–380, Amsterdam, The Netherlands, September 2008. IOS Press. 19

[SJJ06] 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*, pages 225–236, Amsterdam, The Netherlands, September 2006. IOS Press. 19

[SJJ07] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. A Native Transterpreter for the LEGO Mindstorms RCX. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process*

Architectures 2007, volume 65 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, July 2007. IOS Press. 19

[SR09] Jonathan Simpson and Carl G. Ritson. Toward Process Architectures for Behavioural Robotics. In 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*, pages 375–386, Amsterdam, The Netherlands, November 2009. IOS Press. 20

[ST04] Matthew Slowe and Ben Tanner. Graphical Analysis Tool for occamResources. Final year BSc project report, University of Kent, 2004. 9, 117, 118

[Sta95] John T. Stasko. The PARADE environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95-03, Georgia Institute of Technology, 1995. 52

[SK93] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993. 52

[SB94] Robert D. Steele and Paul G. Backes. Ada and real-time robotics: Lessons learned. *Computer*, 27(4):49–54, 1994. 173

[Ste87] Susan Stepney. GRAIL: Graphical representation of activity, interconnection and loading. In Traian Muntean, editor, *7th Technical meeting of the occam User Group,Grenoble, France*. IOS Amsterdam, 1987. 10, 114, 156

[Ste89] Susan Stepney. Pictorial representation of parallel programs. In Alistair Kilgour and Rae A. Earnshaw, editors, *Graphical Tools for Software Engineering*, BCS conference proceedings. CUP, 1989. 114

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005. 25

[The06] The LEGO Group. *LEGO MINDSTORMS NXT Hardware Developer Kit*, 1.0 edition, 2006. 27

[TIO12] TIOBE Software. TIOBE Programming Community Index for march 2012. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, March 2012. 29

[VHB+00] Erik H. J. Volkerink, Gerald H. Hilderink, Jan F. Broenink, W.A. Veroort, and André W. P. Bakkers. CSP Design Model and Tool Support. In Peter H. Welch and André W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 33–48, September 2000. 116

[Wel99] Peter H. Welch. Parallel and Distributed Computing in Education (Invited Talk). In José M. L. M. Palma, Jack J. Dongarra, and Vicente Hernández, editors, *VECPAR'98: Third International Conference on Vector and Parallel Processing - Selected Papers*, volume 1573 of *Lecture Notes in Computer Science*, pages 301–330. Springer-Verlag, June 1999. 45

[Wel12] Peter H. Welch. List of occam Enhancement Proposals. https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP, March 2012. 32

[WB05] 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. 32

[WB08] Peter H. Welch and Neil C. C. Brown. The JCSP Home Page: Communicating Sequential Processes for Java. http://www.cs.kent.ac.uk/projects/ofa/jcsp/, March 2008. 29, 136

[W+94] Peter H. Welch et al. occam For All: Case for Support, 1994. 31

[WJW93] Peter H. Welch, George R. R. Justo, and Colin J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In Reinhard Grebe, Jens Hektor, Susan C. Hilton, Mike R. Jane, and Peter H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 36 of *Transputer and occam engineering series*, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands. ISBN 90-5199-140-1. 115

[WB01] Kirsten N. Whitley and Alan F. Blackwell. Visual Programming in the Wild: A Survey of LabVIEW Programmers. *Journal of Visual Languages & Computing*, 12(4):435–472, 2001. 176

[Win12] Alan F. T. Winfield. *Robotics: A Very Short Introduction*. Oxford University Press, 2012. 36

[Woo98] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent at Canterbury, August 1998. 58

[XMO12] XMOS Ltd. xCORE Multicore Microcontrollers Overview, March 2012. 26

[ZM95] Kang Zhang and Gaurav Marwaha. Visputer–A Graphical Visualization Tool for Parallel Programming. *The Computer Journal*, 38(8):658–669, 1995. 114