Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Visual Process-Oriented Programming for Robotics

Jonathan SIMPSON and Christian L. JACOBSEN

Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, England.
{jon, christian}@transterpreter.org

Abstract. When teaching concurrency, using a *process-oriented* language, it is often introduced through a visual representation of programs in the form of *process network diagrams*. These diagrams allow the design of and abstract reasoning about programs, consisting of concurrently executing communicating processes, without needing any syntactic knowledge of the eventual implementation language. Process network diagrams are usually drawn on paper or with general-purpose diagramming software, meaning the program must be implemented as syntactically correct program code before it can be run. This paper presents *POPed*, an introductory parallel programming tool leveraging process network diagrams as a visual language for the creation of process-oriented programs. Using only visual layout and connection of pre-created components, the user can explore process orientation without knowledge of the underlying programming language, enabling a "processes first" approach to parallel programming. POPed has been targeted specifically at basic robotic control, to provide a context in which introductory parallel programming can be naturally motivated.

Introduction

At the University of Kent, parallel programming is introduced using the occam-π programming language [1]. This is a relatively small language, based on the formalisms of Milner's π-calculus [2] and Hoare's Communicating Sequential Process (CSP) algebra [3]. One of the strengths of occam-π is its simplicity in expressing parallel programs, and that knowledge of the π-calculus or CSP algebra is not required to make effective use of the language. The result is that parallel programming can be taught without referring to the underlying formalisms, whilst maintaining these as subjects for later study within the context of the programming language.

Given the nature of occam-π programs, made up of concurrently executing processes that communicate over well-defined channel interfaces, it is often useful to visualise these as networks of processes. Indeed, when teaching the occam-π language to students, as the focus is on the parallel aspects of the language, they are often asked to draw such diagrams as part of their first programming exercises. As the students complete these first exercises, they are shown the steps that one can perform in order to turn a process diagram into a running occam-π program. They must however, also be taught the syntax of the language, how to operate the compiler, and how to write the sequential parts of a program[1]. It is our view that this early introduction to syntax and tools distracts from learning programming [4]. The initial need to learn sequential programming distracts from what should be the goal of the students' first encounter with occam-π: illustrating the power of process-oriented parallel programming.

[1]While it is possible to provide a toolbox of occam-π processes, and thus require no sequential programming, the syntax and tools must be taught in all cases.

This involves them gaining hands on experience in designing programs composed of multiple, concurrently operating processes which communicate using synchronous message passing. By familiarising themselves with the composition of processes and channels to construct programs, it is our hope that introductory parallel programmers will have a reduced number of concepts to understand when introduced to the occam-π programming language.

We will start by detailing in Section 1 the specific motivation behind the need for another visual programming tool for occam-π, and will then, in Section 2, explore previous work both specifically in relation to the occam family of programming languages as well as popular visual programming tools for other languages. Section 3 will describe the current prototype of the visual programming environment, followed by examples of the kinds of programs we envision using for an introduction to parallel programming using the tool (Section 4). Finally, Section 5 will present the conclusions and the future work required to make the tool described in this paper accessible to a wider audience.

1. Motivation

Programs written in process-oriented languages such as occam-π often have their state represented visually through the use of process network diagrams. When teaching concurrency at the University of Kent these diagrams are used from the very first lecture, allowing students to begin to understand the concepts involved with networks of communicating processes without having any knowledge of occam-π. The use of process network diagrams allows parallel programs to be designed on paper, given a number of well-defined processes, simply by drawing networks of those processes like the diagram shown in Figure 1.

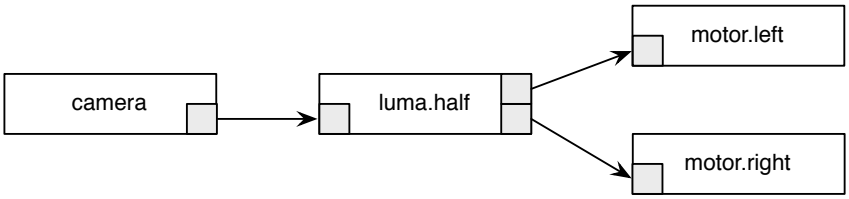


Figure 1. A process network for a simple robotics program

Given the `camera`, `luma.half`, `motor.left` and `motor.right` processes and a knowledge of their interfaces, a student can construct the process network above in occam-π program code, as shown in Listing 1. Nearly all of the requisite information is encapsulated within the process network diagram itself with the exception of the channel types, which are part of the interface to each process.

```
PROC light.seek ()
  CHAN FRAME frame:
  CHAN INT left, right:
  PAR
    camera(frame!)
    luma.half(frame?, left!, right!)
    motor.left(left?)
    motor.right(right?)
:
```

Listing 1. Construction of the process network for the example simple robotics program in occam-π

Robotics lends a useful context in which to frame the visual design of parallel programs. Robotic control can be reduced to a problem of transforming sensory input (the view the

robot has of the world around it) into action, allowing the robot to react to the environment around it. The data-flow model of communicating processes fits naturally into this simplified model of robotic control, with hardware interface processes providing streams of sensory information and outputs for commands to the effectors of the robotics platform. The choice of hardware interface processes used in a program can capture the hardware configuration of a re-configurable robotics platform where different input and output devices can be connected to a selection of ports on the robot.

The visual programming environment presented in this paper has been limited in scope. Specifically, the tool will be limited to creating introductory-level parallel programs for use on small-scale robots, a target platform for introductory programming which the authors have been exploring for some time [5]. The application of simple design patterns on a small robotics platform, for introducing users to parallel programming, has influenced our approach to designing a visual programming tool [6].

Limiting the scope to introductory parallel programming with robotics makes the tasks of creating and distributing a visual editor manageable and provides an environment (a robot) which we feel provides an authentic setting. However there are further applications for the concepts behind our visual editor in the graphical construction of large scale process-oriented systems such as those currently being explored by the RMoX [7] and CoSMoS [8] projects.

1.1. Surveyor SRV-1

We have chosen to focus our prototype environment to support process-oriented robotic control on the Surveyor SRV-1, a tracked mobile robotics platform equipped with two laser pointers and a 1.3-megapixel camera [9]. This platform uses a 500MHz Analog Devices Blackfin processor along with 32MB of SDRAM and equipped with Wi-Fi connectivity, provided as a serial IO interface to the control board. This network capability allows the creation of programming environments for the SRV-1 which communicate with the robot directly to upload new programs "over the air" and offers many opportunities for streamlining the process of working with a remote host separate to the development machine.

The SRV-1 provides significant motivation for our work as we have recently completed a port of the Transterpreter virtual machine to the platform [10]. The Transterpreter is a highly portable runtime environment for occam-π with a cross-platform interpreter core and platform-specific wrappers interfacing the core to the hardware platform to provide a given port. Our SRV-1 port is unique in that it includes a replacement for the standard firmware for the platform written in occam-π which provides a hardware interface composed of parallel processes. This hardware interface makes the SRV-1 a good platform for occam-π robotics and also makes it a compelling environment in which to introduce new users to parallel robotics.

2. Related Work

In designing a graphical editor for use in parallel robotics, a depth of previous work with visual languages, both in robotics and for parallel programming is available. We have separated the two areas and explored each, allowing identification of the best features of visual languages and environments designed for robotics independently of those best suited to representing process-oriented systems.

2.1. Visual Programming Languages in Robotics

The applicability of visual languages to robotic control was given significant attention in academia during the 1996 and 1997 competitions at the Symposium on Visual Languages,

presenting tasks promoting the use of visual languages in robotic control and leading to the development of languages such as VBBL. Commercial visual programming environments such as robotics-focused variants of LabVIEW and the recent Microsoft Robotics Studio have a strong presence within educational robotics due to their widespread availability. A number of these tools are examined below, presented in chronological order.

2.1.1. 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 [11]. The use of external connections within the graphical environment to receive values from sensors and provide values to actuators at development time is an interesting feature, and one that would be useful for a tool accompanied by a simulation environment.

2.1.2. Visual Behaviour-based Language (VBBL)

Cox *et al.* [12] made use of a visual object-oriented dataflow language called ProGraph to develop Visual Behaviour-based Language (VBBL), a rule-based visual language making use of finite state machines (FSM's). These FSM's define various sets of behaviours that are switched between based on conditions and message flows. ProGraph allows the use of a message passing model between components, but all language operations must be represented graphically. 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 [13].

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-π 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 re-configurable robotics platforms such as the LEGO Mindstorms RCX or NXT and its utility and application has been previously explored [14].

2.1.3. LabVIEW and LEGO Mindstorms

There are a commercially available visual programming environments used for robotics, a number of which are derived from National Instruments' LabVIEW product. The LEGO Mindstorms RCX and NXT series of robots has had much influence in promoting visual programming languages in education, having included several visual languages with both platforms. The RCX was supplied to educators with RoboLab, a flowchart language based on LabVIEW [15]. 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. A simplified and unrelated language called *RCX Code* was present in the consumer version of the RCX, which used components which slotted together as puzzle pieces. RCX Code was developed by LEGO from a prototype environment developed by the MIT Learning and Epistemology group called LogoBlocks [16] from which the visual model has been carried forward into Scratch [17], a visual language focused on allowing children eight and up to learn mathematical and computational skills.

The LEGO Group continue to bundle a visual language with the newer Mindstorms NXT, another LabVIEW variant 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 [18].

2.1.4. Microsoft Robotics Studio

Another recent development in commercial visual programming environments for robotics is the Microsoft Visual Programming Language (MSVPL), included only with Microsoft's Robotics Studio (MSRS) [19]. This language is much like NXT-G in being a robotics focused dataflow language, and it generates code which runs on top of the Concurrency and Coordination Runtime (CCR), used to execute robotics programs designed with MSRS. The CCR uses asynchronous communication between components to allow the design of parallel systems. The toolbox and component canvas model used in the MSVPL is very similar to that which we propose for POPed. MSVPL can be used to program the Surveyor SRV-1, our target robotics platform. 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. 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 2.

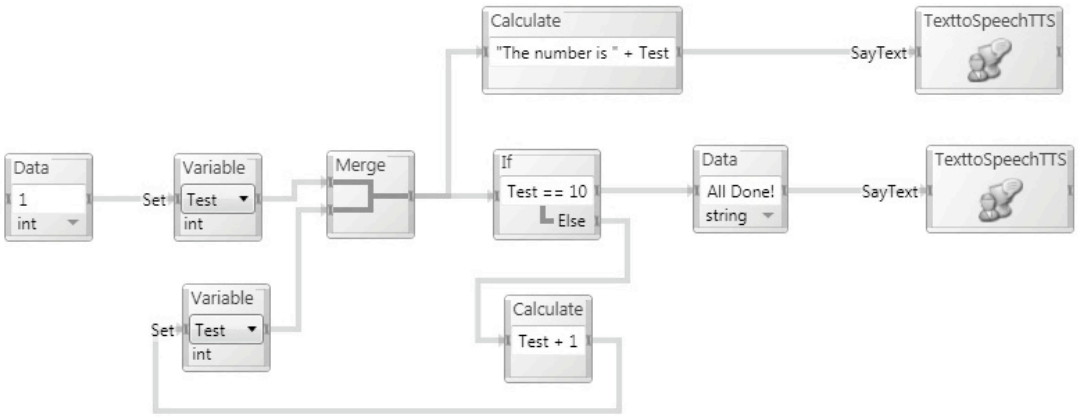


Figure 2. A sample program in the Microsoft Visual Programming language, showing its visual representation for variable assignment, conditional and hardware interface, from [19]

2.2. Visual occam and CSP-based Tools

Historically a number of visual occam programming tools, as well as program visualisation tools have been created and used; where the former is used for creating new programs visually, and the latter for visualising existing programs. It is also possible to find a number of other, CSP-based tools that are not specific to occam. Notably, of the occam tools which can be found in the literature, very few of them are in a state in which they can be used today.

Whilst examining occam tools designed for use with networks of Transputers, it is clear that there are a number of additional features commonly present to support the implementation and optimisation of occam programs on such parallel hardware. Visual layout of software processes across processors in a Transputer network and the instrumentation of programs to provide processor and link utilisation data for optimisation are two development processes that currently have no equivalent in the development of modern occam-π applications, and these may be opportunities for future expansion.

2.2.1. 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 [20,21]. 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, as discussed in Section 5.1.

2.2.2. Visputer and Millipede

Other graphical tools have traditionally focused on the design of programs for execution on networks of Transputers, a once widespread hardware processor designed for use with the occam programming language. Visputer by Zhang and Marwaha provided a complete set of graphical tools for program composition, processor allocation, performance monitoring and debugging [22]. Visputer provided the option for nested processes to be expressed in a visual language, with low-level logic provided textually. Millipede by Aspnas *et al.* provided visual process layout integrated along with performance monitoring of processes and communication links [23]. 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.

2.2.3. TRAPPER

TRAPPER by Scheidler *et al.* offered a graphical programming environment comprising four separate component tools: Designtool, Configtool, Vistool and Perftool [24]. 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 3 on the next page. 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.

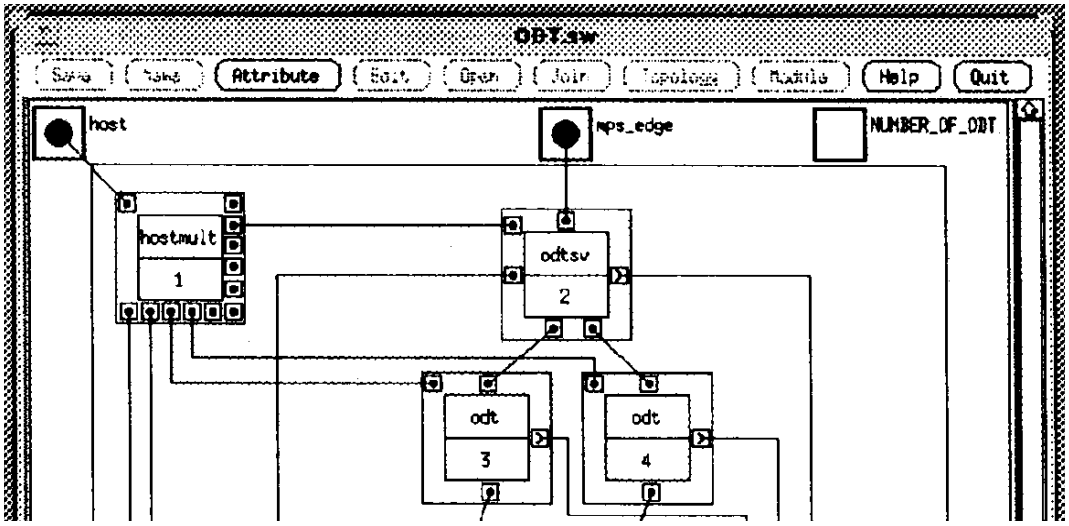


Figure 3. TRAPPER's Designtool showing its connection points outside of the canvas for interfacing to external components, from [24]

2.2.4. The occam Design Tool)

The occam Design Tool (ODT) by Beckett and Welch allowed for process networks to be designed graphically using common paradigms of occam programming [25]. 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 ODT to enforce the Client/Server, IO-SEQ and IO-PAR parallel design patterns [26], ensuring that networks designed with ODT 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 ODT. 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.

2.2.5. gCSP

The gCSP tool by Hilderink and Broenink *et al.* [27] offers a visual environment for designing concurrent programs using a graphical modelling language based on CSP [28]. Both dataflow 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 [29]. 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.

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 [30]. The model of code generation from connected components provides benefits we also seek in our solution, such as the removal of syntax. The visual language used in gCSP is designed for the fully express programs graphically and has stronger ties to CSP than occam-π. The generality of gCSP is problematic for its use as an introductory tool for occam-π programmers.

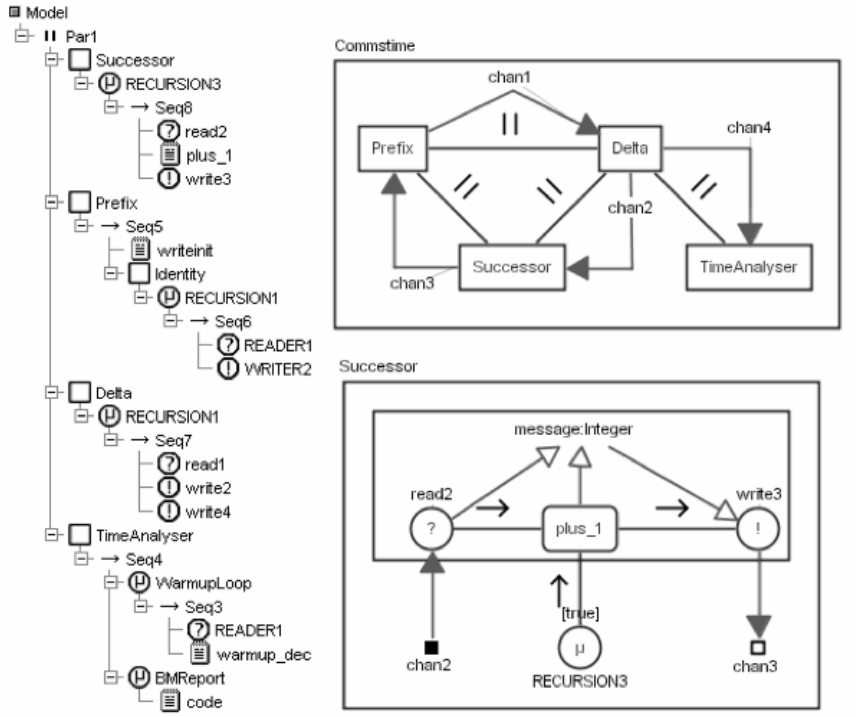


Figure 4. A gCSP session showing sequential and parallel composition of processes, along with the hierarchical browser, from [30]

2.2.6. GATOR

The Graphical Analysis Tool for occam Resources (GATOR) by Slowe and Tanner aimed to provide a debugging aid for occam programs and to progress towards a graphical environment for the program creation [31]. GATOR parsed already written textual occam programs to generate visual representations and offered no way to edit create new or edit existing processes in a loaded network, instead focusing on providing basic interrogation of an executing occam program.

2.2.7. LOVE

The Live occam Visual Environment or LOVE presents a graphical framework for audio processing through the creation of networks constructed from a palette of synthesiser components [32]. Popular synthesis tools have used a dataflow approach, beginning with the MAX graphical language by Puckette *et al.* [33] and its contemporary, the open source Pure-Data [34] which builds upon four primary "atoms" to construct more complex objects which are constructed in dataflow networks.

LOVE shares scope with our intended tool, in presenting a predefined set of components for the user to connect together on a canvas on which they can be arranged. 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. LOVE also has graphical selection of channel ends to aid in making connections, offering visual cues for correct input connections from a selected output. Enforcing type rules, through the control of connections made, is a desirable feature in a visual editor as type errors can be eliminated.

2.2.8. POPExplorer

POPExplorer by Jacobsen takes a very different approach to manipulating process networks, interfacing with the Transterpreter virtual machine runtime for occam-π [35]. POPExplorer

contains a toolbox of processes whose bytecode is loaded into the virtual machine when they are dropped onto the canvas, and subsequently executed when the user chooses for them to be 'live'. Channels can be connected and disconnected whilst processes are running, and the communication state and type of a given channel end is displayed. Problems are caused by the capabilities for run-time connection and disconnection of channels, caused mainly by the design of the occam language to assume a reliable and fault-free environment. Solving these problems and providing an interactive process canvas is an area for further work.

3. Prototyping POPed

To explore the notion of a visual process editing tool for parallel robotics, we have created POPed, a prototype with which to further develop and evaluate our ideas. We have written our prototype in the Python programming language using the wxPython user interface library, as Python has a good range of inbuilt libraries and the use of wxWidgets for the GUI allows our software to run across multiple platforms. Python also streamlines the process of generating distributions for platforms like Windows and Mac OS X.

The POPed tool contains a set of processes designed for use in creating simple robotic control programs to run on the Surveyor SRV-1 mobile robot. To write programs built-in processes are dropped onto a canvas and connected together with channels, using connection points representing the channel ends of the processes. When the user wishes to execute the program, the environment checks that the process network is fully connected and then creates an executable occam-π program 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 automation removes an entire category of potential errors relating to the wiring together of process networks. The set of processes supplied with POPed could be customised to allow the use of the tool for a number of applications and the underlying code generation technique could equally use another process-oriented language such as PyCSP [36] or JCSP [37].

The POPed user interface has three main components: a Toolbox, Canvas and Information Panel. These are shown in the interface diagram in Figure 5.

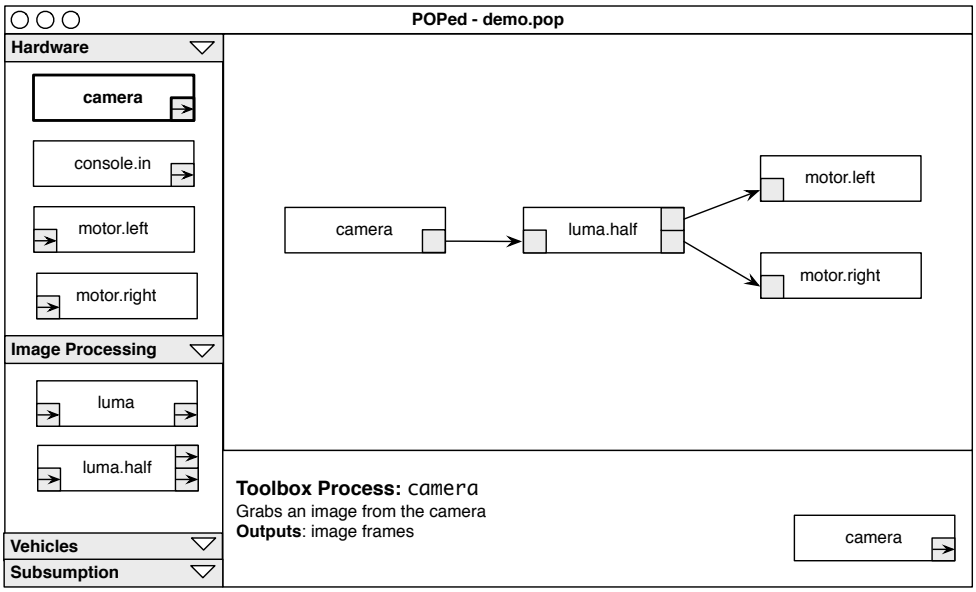


Figure 5. An Illustration of the POPed User Interface

3.1. Toolbox

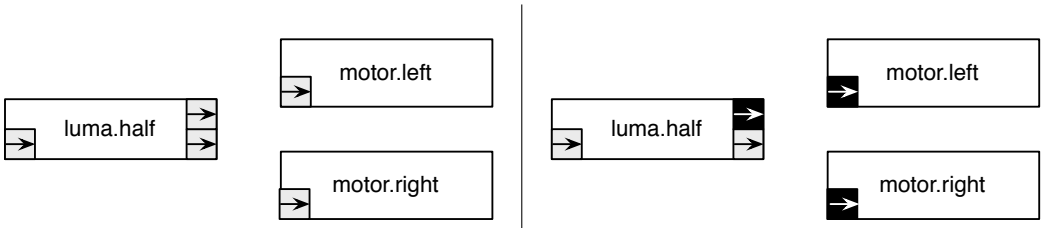
The 'toolbox' is located on the left-hand side of the window and contains a graphical list of the processes available to the user with small diagrams of each process. The diagrams have connection points showing the direction of the channel ends specified in their interfaces to help clarify the connectivity of the process. The location of the arrows follows the general principle of data-flow from left to right, with inputs on the left and outputs on the right. Using a visual representation of the process in the toolbox provides cues for the data-flow behaviour of processes. When a process is selected in the toolbox, 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 3.3 on the next page. Processes within the toolbox are logically grouped together. To make use of components to build a program, users drag them from the toolbox onto the canvas, at which point a instance of the process in the toolbox appears and can be freely positioned by the end user.

Toolbox processes can be specified in generic terms, despite the lack of generics in occam-π, whereby a process can be parameterised by a type. POPEd makes use of templating to generate valid code and substitute real types for the generic placeholder types, ensuring that occam-π's strict type rules are met. The ability to use generics means that we can specify processes such as `delta` generally, and allow the tool to customise the delta for basic types such as `INT`, or `BOOL` depending on what is connected to it. 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 substituted into these false generics. As our interfaces to hardware on the Surveyor SRV-1 make extensive use of `PROTOCOL`s, this will need to be improved.

3.2. Canvas

The canvas is the central focus of POPed, 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, giving a visual representation of both type-compatibility and unconnected nodes. Only the points to which connections can be made to are made active reducing the potential for erroneous clicks on other channel ends and attempting to reduce the possibility for mistakes. This selection mechanism is shown in the figure below. 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. This functionality could be extended to include editing of existing processes, but that is outside the scope of our prototype. Ensuring the underlying code is not hidden is important as we aim for POPed to be a first tool, with its users moving on to write programs textually in the occam-π language.



(a) Possible connection points, no selection made

(b) Connection point selected, possible connections highlighted

Figure 6. Connection points and their type highlighting mechanism

3.3. 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 5 on page 373, while an example of a selected canvas process is shown in Figure 7.

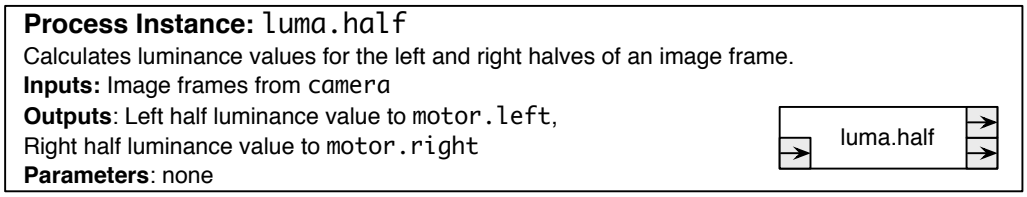


Figure 7. The information panel with a process instance selected on the canvas

This presentation of additional textual information is intended to allow the student to fully understand the components and connections that make up their program, along with the relationship between the connected processes. We can infer information from the connections between processes to present the user with textual descriptions of the inputs 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 Figure 7, where a `camera` process has been connected to input and two motor control processes (`motor.left`, `motor.right`) are connected to the outputs.

4. Example Scenarios

Two specific robotics control applications are being considered in the construction of POPed's default process set. 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's *Vehicles* and the subsumption architecture.

4.1. Behavioural Control: Vehicles

When targeting first explorations in concurrent robotics, Braitenberg's *Vehicles* offers a useful introduction to what can be accomplished with small numbers of processes connected together [38]. By connecting sources of sensory input directly (or almost directly) to outputs, very simple programs can be created.

The program shown in Figure 1 on page 366 shows a simple program for the Surveyor SRV-1 which uses very few processes to achieve a useful 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, which change the speed that the motors on each side of the robot run at 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, as a stronger light reading on the left-hand side will cause the left track to speed up, and the right track to be slowed down. The inverse is also true: by slowing the motors as the light level increases, we can program a robot that seems to 'like' light and heads towards it. In the Vehicles set of processes, an `invert` process is included, to make switching between positive and negative relations of inputs and outputs easier. Also worth noting for the implementation of 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.

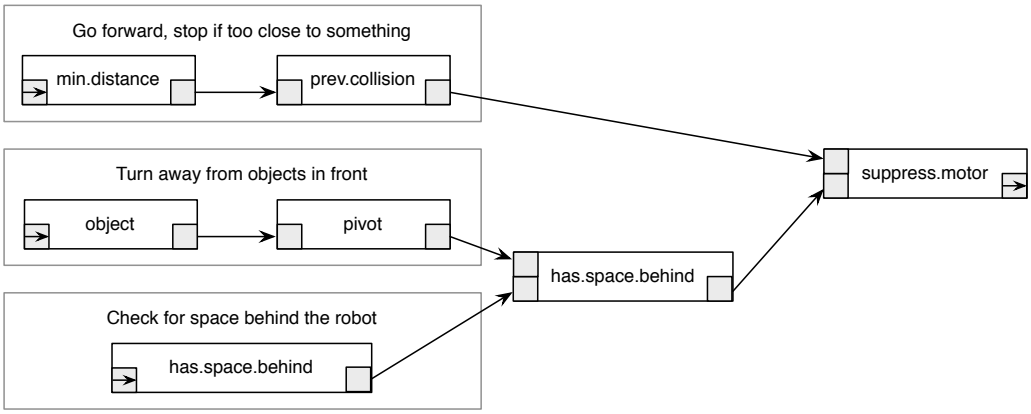


Figure 8. A small robot control program built with the subsumption architecture which uses two types of sensor input and three behaviours to manoeuvre around a space

4.2. Subsumption Architecture

Brooks' *subsumption architecture* involves building robot control systems with increasing levels of competence composed of concurrently operating modules [39]. It offers a design paradigm for robotic control that emphasises re-use, decentralisation and concurrent, communicating processes. The application of the subsumption architecture for designing parallel robotics programs has been previously explored in [40], and provides substantial motivation for the use of a graphical process network configuration tool.

We have found the use of diagrams representing subsumption architectures particularly useful due to the additional complexity of the interactions between their levels of behaviour. A first step towards supporting their development of within POPed has been to include processes for suppression and inhibition, the two primary operations carried out between different levels of behaviour. To fully 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 behaviour and allow the expansion of that behaviour on demand, hiding complexity when it is not required. The ability to expand and contract sub-networks of processes is discussed further as an area of future work in Section 5.1 on page 378.

To illustrate the advantages of a visual approach to the design of subsumption architectures, we can look at a process network diagram for a simple program, and the occam-π code required to set up the network. The diagram in Figure 8 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. Each behaviour is broken out separately in the diagram, with an outer box enclosing the sets of processes which are composed to create the behaviour. The occam-π 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 2. The complexity inherent to this code, having to write code to connect the network with named channels, can be completely eliminated with a 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 based on many different input sensors.

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

Listing 2. Construction of the process network for the example simple robotics program in occam-π

5. Conclusions

In this paper we have discussed the design features of an environment for introducing parallel programming in a "processes first" methodology, allowing the demonstration of parallel design patterns and manipulation of process networks without having to learn the syntax of a parallel language. These design features are the product of studying other tools for robotics programming and process network design in occam and related languages. We continue to develop our prototype tool, with the aim of including it along with our port of the Transterpreter runtime to the Surveyor SRV-1 as an illustration of process-oriented design for robotic control systems. It is our intention to improve upon this prototype to the stage where it can be used in the classroom, allowing the use of feedback from students to shape further development.

By focusing on robotics applications we have been able to begin designing a graphical parallel program development environment for the occam-π program language and create a tool which can be used to illustrate and explore simple process-oriented robotics with those unfamiliar to occam-π. We feel there is much left to do in this area, and a large number of questions remain in the development of effective visual tools for pedagogic process-oriented programming.

5.1. Future Work

The occam-π programming language contains features for mobility of channels and processes at run-time which are heavily used in larger applications. At the University of Kent there are currently two research projects which could see potential benefit from graphical system construction tools.

One is RMoX, an experimental occam-π operating system which could allow the generation of operating systems for a given embedded hardware platform by the visual composition of the correct hardware drivers and system components, an approach previously illustrated by gCSP [30]. The second is CoSMoS, a complex systems modelling and analysis project which aims to allow non-programmers to be able to build their own systems. The CoSMoS project could see benefit from an environment to allow these users to express systems visually.

Both of these systems share a heavy use of the dynamic language features in occam-π, with large amounts of process creation and mobility meaning that static process network diagrams are significantly reduced in utility, useful only as 'snapshots' of an executing system. Developing a visual language and model for representing and designing these dynamic systems is an open area of research. A pedagogic tool which encompasses the entire feature-set of occam-π would have significant impact in introducing new users to building complex applications.

Work is currently ongoing into addition of debugging facilities to the Transterpreter virtual machine to provide high-quality debugging of occam-π programs, allowing the user to single-step, slow down execution and monitor the activity of programs as they execute. Exposing these facilities to users in ways that allow them gain a better understanding of how their code executes will be an exercise in user interface design. We consider the problems of representation in debugging and performance monitoring similar to those involved in creating a visual programming language. Integrating the program design and implementation stages with a live runtime would provide similar facilities to the POPExplorer tool discussed in Section 2 on page 367 to be built upon futher.

Having worked with the BlueJ environment for Java where objects can be instantiated individually on an "object bench" and interacted with, we see potential in allowing users to interact with individual processes to explore their run-time behaviour [41,42]. We envisage that the additional control in the runtime could be used to provide a live test environment with provided values or data sets for channel inputs, giving a "process bench" for vivisection of processes. The integration of testing abilities and live experimentation with processes or networks of processes has much potential for use with introductory parallel programmers.

Stepney's GRAIL [20] included a visual representation of programs written in occam based on the structure of the program code itself, and this representation bears a resemblance to Scratch [17], a visual programming language developed at MIT and used to encourage school-age children to learn programming. This resemblance serves to highlight the potential for a fully-visual programming language inspired by occam-π, and exploring the possibilities for graphical representations to remove syntax and provide an augmented editing environment for the textual occam-π language may yield improvements for those new to occam-π.

Acknowledgements

Many thanks to all who continue to work on and support the Transterpreter project. Howard Gordon of Surveyor Corporation has provided support for our work with the SRV-1 by supplying robotics platforms to work with. Carl Ritson has both made improvements to the Transterpreter and provided an excellent, fully-featured port to the Surveyor, facilitating a wide range of further robotics work. Additional thanks to Adam Sampson, Fred Barnes and

Peter Welch who have provided both formative input and extremely helpful feedback on this work.

References

[1] P.H. Welch and F.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.

[2] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

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

[4] Matthew C. Jadud. Methods and tools for exploring novice compilation behaviour. In *ICER '06: Proceedings of the 2006 international workshop on Computing education research*, pages 73–84, New York, NY, USA, 2006. ACM Press.

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

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

[7] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 269–288, IOS Press, Amsterdam, The Netherlands, September 2003. ISBN: 1-58603-381-6.

[8] Susan Stepney and Peter H. Welch. The CoSMoS Research Project. http://www.cosmos-research.org/caseforsupport.html, October 2007.

[9] Surveyor Corporation. Surveyor SRV-1 Blackfin Robot. http://www.surveyor.com/, July 2008.

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

[11] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, and A. 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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):2000, 2000.

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

[17] 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.

[18] Team VictorTango - 2007 DARPA Urban Challenge. http://www.me.vt.edu/urbanchallenge/, November 2007.

[19] Microsoft Corp. Microsoft Robotics Studio: VPL Introduction. http://msdn.microsoft.com/en-us/library/bb483088.aspx, 2008.

[20] 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.

[21] 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.

[22] K. Zhang and G. Marwaha. Visputer–A Graphical Visualization Tool for Parallel Programming. *The Computer Journal*, 38(8):658–669, 1995.

[23] M. Aspnas, R. Back, and T. Langbacka. Millipede: A programming environment providing visual support for parallel programming, 1992.

[24] C. Scheidler, L. Schafers, and O. Kramer-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.

[25] D.J. Beckett and P.H. Welch. A Strict occam Design Tool. In C.R. Jesshope and A. Shafarenko, editors, *Proceedings of UK Parallel '96*, pages 53–69, Guildford, UK, July 1996. Springer-Verlag, London.

[26] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands.

[27] 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.

[28] 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.

[29] 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.

[30] Jan F. Broenink, Marcel A. Groothuis, and Geert K. Liet. gCSP occam Code Generation for RMoX. In *Communicating Process Architectures 2005*, pages 375–383, sep 2005.

[31] Matthew Slowe and Ben Tanner. Graphical Analysis Tool for Occam Resources. Final year B.Sc project report, University of Kent, 2004.

[32] Adam Sampson. What is LOVE? Available at: https://www.cs.kent.ac.uk/research/groups/sys/wiki/LOVE, September 2006.

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

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

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

[36] 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.

[37] Peter H. Welch and Neil Brown. The JCSP Home Page: Communicating Sequential Processes for Java. http://www.cs.kent.ac.uk/projects/ofa/jcsp/, March 2008.

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

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

[40] 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.

[41] Michael Kölling and John Rosenberg. Tools and techniques for teaching objects first in a Java course. *SIGCSE Bulletin*, 31(1):368, 1999.

[42] Michael Kölling and John Rosenberg. Objects first with Java and BlueJ (seminar session). In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, page 429, New York, NY, USA, 2000. ACM Press.