Concurrency, Robotics, and RoboDeb

Christian L. Jacobsen and **Matthew C. Jadud**
University of Kent
Canterbury, Kent
CT2 7NF
UK

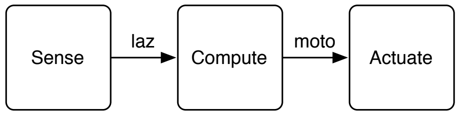
Introduction

Robotics is an engaging and natural application area for concurrent and parallel models of control. To explore these ideas, we have developed environments and materials to support the programming of robots to do interesting tasks in a fundamentally concurrent manner. Our most recent work involves the development of RoboDeb, a "virtual computer" pre-installed with the open-source Player API and Stage simulator to support classroom exploration of concurrency and robotic control using the occam-π programming language.

Concurrency, naturally

We believe in powerful abstractions, and programming languages are abstractions unto themselves. When it comes to developing programs for robots, we want to choose tools that help us correctly implement our concurrency as naturally as possible. This implies to us that the language we use to implement robotic control solutions should have powerful abstractions and constructs for dealing with ideas regarding parallelism and choice.

There are many languages that are closely tied to a formalism of one sort or another; for example, both Haskell and Scheme draw heavily on the lambda calculus. Neither of these languages necessarily requires you to know the lambda calculus—but it provides a model that helps keep the language theoretically self-consistent. Similarly, the programming language occam-π (INMOS Limited 1988; Barnes & Welch 2004) is closely tied to the Communicating Sequential Processes algebra—an algebra for describing concurrently executing processes that interact over synchronizing communications channels (Hoare 1985). Using a language like occam-π, we can think about, design, and implement concurrent control constructs without resorting to low-level primitives like threads, locks, and semaphores. This is in contrast to languages like C, C++, Java, and Python, which are all sequential in nature, and provide limited support for managing the complexity of concurrent solutions to problems(Boehm 2005).



```
CHAN LASER laz:
CHAN MOTOR moto:
PAR
  Sense(laz!)
  Compute(laz?, moto!)
  Actuate(moto?)
```

Figure 1: A three-process network, and occam-π code representing it.

Regarding occam-π

Consider a simple robot: input comes from its sensors, output is directed to its actuators, and it performs some computation in-between. We might imagine these tasks as three communicating processes (Figure 1). The `Sense` process reads from hardware, and passes that data down a channel to the `Compute` process. After decisions have been made in the `Compute` process, motor commands may (or may not) be passed on to the `Actuate` process, which then alters the hardware state in accordance with those commands.

The occam-π code in Figure 1 serves as a description of the diagram above it. In the program fragment two channels are declared, and then three processes are executed in parallel that communicate over those channels. These six lines of code capture several critical aspects of the language, and indirectly, the CSP algebra:

PARallel execution The three processes (`Sense`, `Compute`, and `Actuate`) are running in parallel (because they are grouped under a `PAR` block). We trust the runtime environment to interleave them fairly (if we are on a uniprocessor computational device), or to split them up across multiple computational units if possible.

Unidirectional channels The channels linking the three processes are unidirectional, and have two defined ends. The `Sense` process holds one end of the `laz` channel—the transmission, or sending end. This is denoted with the

'!', which is notation that comes from the CSP algebra. Similarly, the `Actuate` process holds the receiving end of the `moto` channel, which is denoted by the presence of a '?'. As can be seen the `Compute` process holds the receiving end of the `laz` channel, and the transmission end of the `moto` channel.

Synchronizing communications Not visible in the diagram or code is the fact that communications in this paradigm are blocking. When the `Sense` process commits to sending data to the `Compute` process over the `laz` channel, it blocks until the corresponding read action happens on the other end. The blocking nature of communications in the occam-π programming languages is how we know where synchronization between concurrent processes will take place.

Each of these processes is entirely self-contained—there is no global or shared state in occam-π, as this is an obvious source of dangerous race hazards. If data is to move from one concurrent process to another, it happens over a channel, and that process is governed by the runtime environment. Just like the Java compiler and runtime environment handle issues regarding the automatic management of memory, the occam-π compiler and runtime environment helps us correctly implement our concurrency.

For discussion

There is obviously much more to occam-π than can be quickly described here. Hopefully, this rapid tour of the language provides a sense for how concurrent and parallel solutions to interesting problems need not become bogged down in the details of implementing that concurrency. In (Jacobsen & Jadud 2005), we have described our own work applying this to teaching with the LEGO Mindstorms. More recently, and on larger robotics platform, Jon Simpson (a rising 3rd-year undergraduate at the University of Kent) has begun exploring the subsumption architecture and how we might express Rodney Brooks's ideas regarding layered robotic control in occam-π (Simpson, Jacobsen, & Jadud 2006).

In this paper, we want to talk about our use of occam-π and robots in the classroom. We have had three opportunities to put our ideas into practice. In *Cool Stuff in Computer Science*, an extra-curricular series of laboratories for undergraduates in the Computing Laboratory at the University of Kent, we tested early implementations of occam-π on the LEGO Mindstorms. In *CO631: Concurrency Design and Practice*, a second year course in the same department, we developed a series of linked exercises for introducing occam-π in the context of the Pioneer3, both real and virtual. Lastly, in a guest lecture and workshop at the University of Copenhagen in Denmark, we focused less on the language and more on doing interesting things as quickly as possible using the tools we had developed; we will focus on this educational experiment here.

Our work at Kent and in Denmark required the development of RoboDeb, a VMWare virtual appliance that gives

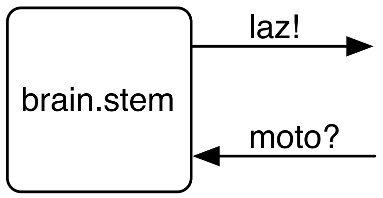


Figure 2: The `brain.stem` process with a sensor output channel, `sick!`, and an input channel, `moto?`.

us a complete simulation environment for large robotics platforms via the open-source Player/Stage API (Gerkey, Vaughan, & Howard 2003). While we prefer physical manipulatives to simulation for many reasons, it is not always possible to bring enough robots with you to run a class or seminar. So, in addition to being portable, the RoboDeb environment has a number of desirable features (e.g. it is easily installed under Windows and Linux, and updates itself in-place), and has made our most recent educational explorations possible.

Robotic caution in Copenhagen

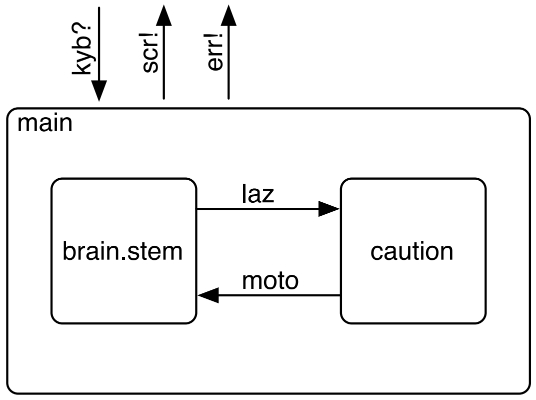
The students at the University of Copenhagen were familiar with concurrent models of programming from their coursework in *Extreme Multiprogramming*, but had little or no experience with either robotics or the programming language occam-π. In a short introductory lecture, we introduced the ideas of processes and channels. A process is a piece of code that runs, perhaps forever. A channel is a "wire" connecting two (and only two) processes that can carry data from one to another... but in one direction only. Lastly, we stressed that communication on these wires is *blocking*; this means that when we begin writing to or reading from a channel, we must wait until the process at the other end decides to engage in the corresponding read or write.

These ideas allowed us to begin our first exercise—to develop a robot that is cautious in its navigation of a space[1]. We defined a cautious robot as one that slows down when it is near an obstacle. Essentially, our robot was a simple Braitenburg vehicle (Braitenberg 1986).

In keeping with our notion of concurrent processes, the first process students saw was the `brain.stem` process. The `brain.stem` is a process that outputs sensor data, and takes in motor commands (Figure 2).

The second process we introduced the students to was `caution`. This process can be "wired up" to the `brain.stem` process, because they are complimentary: one communicates sensor data and reads motor commands, while the other reads sensor data and produces motor commands. Figure 3 depicts the full process network for our

[1]The exercise is included in the download of our robotics simulation environment RoboDeb. It is additionally available online at http://www.transterpreter.org/wiki/Teaching.



```
PROC main(CHAN BYTE kyb?, scr!, err!)
  CHAN LASER sick:
  CHAN MOTOR moto:
  PAR
    brain.stem.ML(moto?, sick!)
    caution(moto!, sick?)
:
```

Figure 3: The `brain.stem` process wired up to `caution`, wrapped in the `main` process, and the corresponding occam-π program.

robot and the complete occam-π program for a "cautious" robot as experienced by the students at DIKU.

At this point in the process, the students do not know how either the `brain.stem` or `caution` processes work internally—they are black-box components for which we only know the API. In thes case of occam-π processes, the API is always the channels a process reads from or writes to, and the type of information carried on those channels. There is no notion of state outside of a process, and the only way for processes to share data is by communicating over channels.

The workshop exercise continued to explore this notion of wiring together ready-made processes to produce robots with interesting behaviors. The worksheet material provided no more than 30 minutes of distraction; students wanted to know how to program their own processes. For this reason, we provided them with the source code to processes like `caution`, and a simplified language reference. Because occam-π is a small language with a consistent, indentation-based syntax, the students (mostly third- and fourth-years) at the University of Copenhagen were able to quickly move on to modifying and extending the processes provided in our small library.

Lessons learned the hard way

Our experiences in Denmark are not the result of isolated brilliance on our part, but instead the result of a learning process. Our first use of occam-π for programming robotics platforms was at the University of Kent in our extracurricular program *Cool Stuff in Computer Science*. Developed by the authors, this series of workshops (open to any student at the University of Kent) attracts motivated Computing and Electronics majors who are interested in engaging with challenging material from outside their normal course of study. It is in this context that we first began exploring the use of occam-π on the LEGO Mindstorms. The experiement was partially successful, but students were frustrated at the time by the immature nature of the tools and documentation. However, because *Cool Stuff in Computer Science* attracts motivated and energetic students, we were able to set them the challenge of being beta testers for our early ideas and software. So while our tools left some of the students wanting, they were glad to have provided us with valuable, early feedback.

These early experiments led to the use of the first editions of RoboDeb in the course *CO631: Concurrency Design and Practice* at the University of Kent. In this course, we presented a series of exercises to the students that culminated in their development of a robot that could wander a maze using a combination of the a forward-facing laser rangefinder and an array of ultrasound sensors. While the students managed the exercises and appreciated them, we felt that too much time was spent "teaching the language," and not enough time was spent by the students thinking about concurrent models of robotic control.

The combination of these experiences led us to reflect on the notion of *no concept before its time*. In producing the lecture and materials for presentation at DIKU, we worked hard to keep things as simple as possible. In this regard, we felt that the notion of concurrent, communicating processes was the single most valuable concept they could experience in a short tutorial session. By minimizing our focus on the language, and instead encouraging the assembly (as opposed to the definition) of concurrent components, the students participating in the workshop were able to quickly "wire up" programs from a small library to create robots with interesting and sometimes surprising behaviors. In this regard, we feel that our work in Copenhagen was very successful.

To support all of these educational explorations, we needed an environment that could easily be used by students for programming a robot both in and out of the laboratory. This need drove our development of RoboDeb as a platform for exploring concurrency and robotics in a virtual environment.

RoboDeb: virtual concurrency and robotics

To support these kinds of explorations, we developed the RoboDeb environment for programming a simulated Pioneer3 in a variety of languages. Put simply, RoboDeb is a virtual machine; to use it, one must first install VMWare Player, a freely available application for both Windows and Linux[2]. Once VMWare Player is installed, RoboDeb can be downloaded from the Transterpreter website, decompressed, and booted[3]. At this point students can begin programming

[2]http://www.vmware.com/products/player/
[3]http://robodeb.transterpreter.org/

simulated robots in a variety of virtual worlds. Figure 4 depicts a RoboDeb session with an editor (and occam-π program) running in the foreground, which in turn is controlling multiple (simulated) Pioneer3 robots running in the background.

The RoboDeb environment was assembled with a number of diverse goals in mind.

Concurrency We wanted an environment that supported a concurrent approach to programming robots that could then be applied in the real world. In this regard, RoboDeb is a success—programs written in the simulator in occam-π can then be run directly on the department's Pioneer3 without modification.

Portability RoboDeb makes it possible for students to develop programs both at home and in the lab, with a minimum amount of effort on the students' part. Most importantly, installation is a simple and non-invasive installation procedure—it is a download and a double-click, for all intents and purposes.

Scalability With similar robotic configurations, it is possible to run the same occam-π program on a Pioneer3 as on a LEGO Mindstorms. The runtime environment for occam-π programs is very small (approx. 12KB); as a result, students can move from one platform to another without sacrificing powerful tools for managing concurrency in the context of robotics and embedded systems (Jacobsen & Jadud 2004).

Neutrality The RoboDeb environment is just a Linux virtual machine. Initally, we included the facility for students to program robots in Java as well as occam-π. In our next release of RoboDeb we hope to include a complete Pyro installation, allowing instructors and students to use this mature framework for exploring AI and robotics (Blank *et al.* 2003). This increases the value of a single RoboDeb installation and opens up other avenues of exploration for interested students who wish to explore robotics and AI further.

Updatability RoboDeb can be updated with a double-click. This allows students and users of the virtual machine to obtain updates to languages, libraries, and software without requiring a complete reinstall of the virtual machine.

RoboDeb serves needs that are not, to the best of our knowledge, easily met by any other freely available robotics simulation environment.

Looking forward

Looking forward, we have many improvements to the RoboDeb environment and our instructional materials supporting the exploration of concurrency and robotics.

For example, if all of the software installed on the virtual appliance was available as Debian packages, then it would be possible to easily "install" RoboDeb on a desktop (running GNU Debian Linux) without VMWare Player. This would also simplify our maintenence and upgrade procedures greatly. Even without these improvements, RoboDeb provides an environment where our students can explore concurrency and robotics under Windows and Linux. Perhaps just as importantly, the occam-π programs they write against the Player/Stage API can be run directly against the department's Pioneer3, which we find is an exciting prospect in many cases.

We have begun exploring the interesting intersections between the design of robotic control and highly concurrent software. Ultimately, we are interested in providing our students with concrete experiences in which they can ground learning regarding concurrent software design. To get there requires a combination of tools, instructional methods, and learning resources that support us, as instructors, in creating challenging learning environments for our students.

References

Barnes, F. R. M., and Welch, P. H. 2004. Communicating Mobile Processes. In *Communicating Process Architectures 2004*, 201–218.

Blank, D.; Kumar, D.; Meeden, L.; and Yanco, H. 2003. Pyro: A python-based versatile programming environment for teaching robotics. *J. Educ. Resour. Comput.* 3(4):1–15.

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

Braitenberg, V. 1986. *Vehicles: Experiments in Synthetic Psychology*. Cambridge, MA, USA: MIT Press.

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

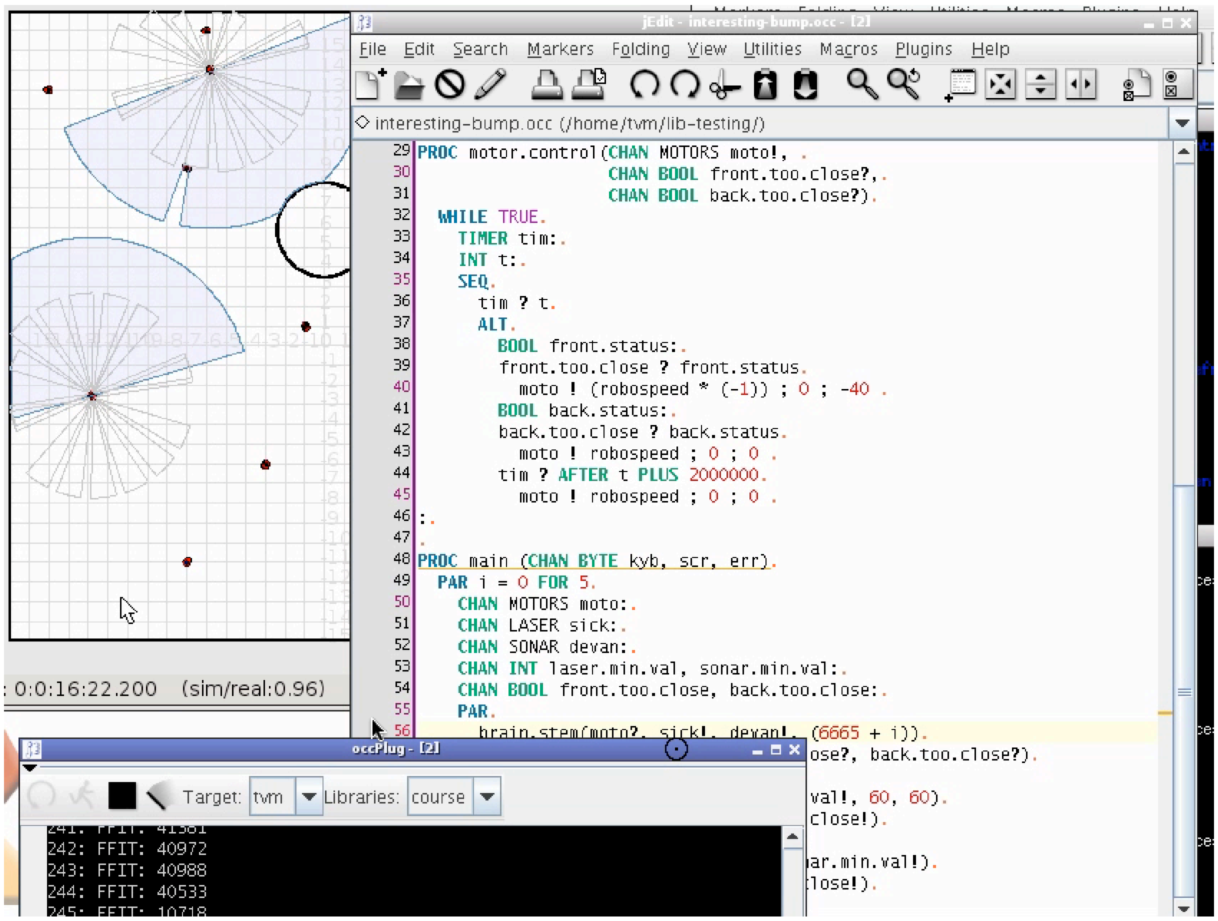
Hoare, C. 1985. *Communicating Sequential Processes*. Prentice-Hall, Inc.

INMOS Limited. 1988. *Transputer reference manual*. Upper Saddle River, NJ 07458, USA: Prentice-Hall. Includes index. Bibliography: p. 315-324.

Jacobsen, C. L., and Jadud, M. C. 2004. The Transterpreter: A Transputer Interpreter. In *Communicating Process Architectures 2004*, 99–107.

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

Simpson, J.; Jacobsen, C. L.; and Jadud, M. C. 2006. Mobile Robot Control - The Subsumption Architecture and occam-pi. In Welch, P.; Kerridge, J.; and Barnes, F., eds., *Communicating Process Architectures 2006*, 225–236. Amsterdam, The Netherlands: IOS Press.


jEdit - interesting-bump.occ - [2]
File Edit Search Markers Folding View Utilities Macros Plugins Help
interesting-bump.occ (/home/tvm/lib-testing/)
occPlug - [2]

```
: 0:0:16:22.200    (sim/real:0.96)
```

```
PROC motor.control(CHAN MOTORS moto!,
                   CHAN BOOL front.too.close?,
                   CHAN BOOL back.too.close?)
  WHILE TRUE
    TIMER tim:
    INT t:
    SEQ
      tim ? t
      ALT
        BOOL front.status:
        front.too.close ? front.status
          moto ! (robospeed * (-1)) ; 0 ; -40
        BOOL back.status:
        back.too.close ? back.status
          moto ! robospeed ; 0 ; 0
        tim ? AFTER t PLUS 2000000
          moto ! robospeed ; 0 ; 0
:

PROC main (CHAN BYTE kyb, scr, err)
  PAR i = 0 FOR 5
    CHAN MOTORS moto:
    CHAN LASER sick:
    CHAN SONAR devan:
    CHAN INT laser.min.val, sonar.min.val:
    CHAN BOOL front.too.close, back.too.close:
    PAR
      brain.stem(moto?, sick!, devan!, (6665 + i))
ose?, back.too.close?)

val!, 60, 60)
close!)

ar.min.val!)
lose!)
```

Target: tvm Libraries: course

```
241: FFIT: 41381
242: FFIT: 40972
243: FFIT: 40988
244: FFIT: 40533
245: FFIT: 10718
```

Figure 4: RoboDeb in action.