Citation for published version

Jacobsen, Christian L. and Jadud, Matthew C. (2005) Torwards Concrete Concurrency: occam-pi on the LEGO Mindstorms. In: SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education.

DOI

http://doi.org/10.1145/1047344.1047485

Towards Concrete Concurrency: occam-pi on the LEGO Mindstorms

Christian L. Jacobsen
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF
clj3@kent.ac.uk

Matthew C. Jadud
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF
matthew.c@jadud.com

ABSTRACT

In a world of ad-hoc networks, highly interconnected mobile devices and increasingly large supercomputer clusters, students need models of computation that help them think about dynamic and concurrent systems. Many of the tools currently available for introducing students to concurrency are difficult to use and are not intrinsically motivating. To provide an authentic, hands-on, and enjoyable introduction to concurrency, we have ported occam-π, a language whose expressive powers are especially compelling for describing communicating dynamic reactive processes, to the LEGO Mindstorms.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Human Factors, Languages

Keywords

LEGO, occam-π, concurrency, parallelism, CSP, fun

1. INTRODUCTION

This paper is about a philosophy of instruction and the tools we have developed for teaching concurrency in the context of this philosophy.

Our philosophy regarding instruction is that students should have fun engaging in authentic, hands-on learning, and they should look forward to those learning experiences. When we say fun, we mean our students should find learning to be enjoyable, challenging and enriching in obvious ways. "Hands-on" means that the learning process is not passive from the learner's perspective (like a typical lecture), but active, requiring students to participate mentally and physically in the learning process. We define authentic learning experiences as those that are true unto themselves; they are not contrived. And when a student walks into our classroom, we want them to look forward to the lesson—even if they don't know what it is going to be.

Our goal is to remain true to our philosophy, and at the same time develop a platform upon which we can explore concurrency and parallelism with our students. We believe the LEGO® Mindstorms™ provides an ideal starting point in this regard. Little robots have to deal with big problems, and the problems students face programming these robots are *real*: navigating around a room, while reading from multiple sensors and communicating with other little robots is an obvious goal, but a difficult task nevertheless. In bringing occam-π[5] to the Mindstorms, we believe we can explore concurrency more deeply, more tangibly and more enjoyably than with the technologies otherwise available to us.

We begin our paper with a brief introduction to occam-π and the run-time environment that we have developed for use in our own classrooms. We then examine a number of other tools and methods for teaching concurrency in section three; in particular, we consider these tools through the lens of our own philosophy of instruction. Lastly, we provide a worked example demonstrating occam-π's expressiveness on the LEGO Mindstorms, and close with a brief discussion of future directions of our work.

2. BACKGROUND

occam-π is a new, explicitly concurrent language, which combines the best features of the Communicating Sequential Process (CSP) algebra, first introduced by Professor Sir Tony Hoare in 1985[15], and the π-calculus, developed by Robin Milner[22]. occam-π has a small number of syntactic constructs (like Scheme) and uses indentation to denote logical blocks of code (like Python). Modeled closely on the CSP algebra, occam-π compilers provide guarantees about the run-time behavior of programs. For example, it is not possible for data race-hazards to take place at run-time. Additionally, one of the defining features of the language is the ability to express non-deterministic choice over communications channels. For example, it is possible to easily respond to any one of many sensors on a little robot. This process of alternating over communications channels (as expressed by the `ALT` construct) is demonstrated in our worked example in section four.

SIGCSE'05, February 23–27, 2005, St. Louis, Missouri, USA.

The CSP model of concurrency provides a clear and simple framework for expressing parallel programs. This is accomplished through the use of unidirectional, point-to-point, blocking *channels* through which data is passed from one process to another while executing in parallel. Unidirectional, point-to-point channels are part of what make occam-π a safer language for programming concurrently. Additionally, because all communications block, each communication becomes an explicit synchronization point in our program. This makes it unnecessary to use spin locks, semaphores, and other error-prone constructs commonly employed when writing parallel programs in other languages.

The CSP model of communicating processes is widely used today: Erlang[2] and Handel-C[3], for example, both build on concepts that originated in CSP. Additionally, occam-π is continually evolving in the form of KRoC –the Kent Retargetable occam-π Compiler[4, 32]. Our efforts extend this work: we have built the *Transterpreter*, a virtual machine interpreting a byte-code that includes instructions directly supporting CSP primitives for concurrency[16]. The byte-codes generated are an integral step of the KRoC compiler, thereby providing the Transterpreter with an existing and proven tool-chain.

Written in strict ANSI C, the Transterpreter runs on all major operating systems and architectures. Because of the Transterpreter's small memory footprint (roughly 5Kb), it is well suited to embedded applications. In addition to being able to execute occam-π programs on Macintosh OS X, Linux, Solaris and Windows, we planned from the beginning for our software to run on small devices like the LEGO Mindstorms.

Because of the extremely portable nature of the Transterpreter, we can execute the exact same occam-π program on the LEGO Mindstorms as we would in our simulator or on larger robotics platforms. The Transterpreter also offers an excellent run-time environment for exploring concurrency; in this regard, occam-π is a language that allows students to develop their skills in concurrency over examples that range from real-time systems to high performance clusters and grid-like architectures.

3. TEACHING CONCURRENCY

Motivation matters. We believe students should have fun exploring authentic tasks in constructive ways[17]. In this section, we examine some pedagogic approaches to introducing concurrency, from the perspective of our beliefs that learning should be fun, authentic and constructive.

3.1 Learning should be fun

In their paper "Using robotics to motivate back door learning," Marion Petre and Blaine Price studied children using the LEGO Mindstorms in robotics competitions[25]. Their research echoes what Fred Martin[21] and others[7] have observed in their own work: little robots can provide a focus for learning and collaboration. It is this engaging, motivational element that we feel is missing from languages like SR[1] and Oz[29] that have been developed for introducing concurrency to students.

Micro-worlds have often filled this motivational void in the past. StarLogo, developed by Mitchel Resnick and others at the MIT Media Labs, is a massively parallel micro-world designed to help children explore and play with decentralized systems[27]. occam-π, like LOGO, is a small and simple language, designed originally for use in embedded systems. We believe the use of the Transterpreter on the Mindstorms will provide us with an environment and metaphor for exploring concurrency in the real world.

3.2 Learning should be authentic

Many students studying operating systems encounter the *dining philosophers problem*[15]. Invented by Dijkstra, this problem may involve (for example) five philosophers who share five chopsticks and one plate of spaghetti. They sit, they think, and they eat, repeating this process indefinitely. Problems arise when one philosopher is infinitely refused the use of a chopstick, by another greedy philosopher, who always pick up their chopstick immediately after putting it down. This will lead to starvation of the first philosopher. Another bad condition, known as deadlock, can occur when everyone picks up one chopstick and no one can pick up the second needed to eat the spaghetti; this leads the philosophers to wait indefinitely for the second chopstick to become available. A number of pedagogic environments have been developed to allow students to explore this problem, visually or otherwise[20, 28].

The dining philosophers problem is authentic in that it accurately captures the problem of sharing resources (like memory and disk) by two or more simultaneous processes in a computing system. However, the problem lacks real-world authenticity: it is an analogy. When programming a LEGO Mindstorms equipped with multiple sensors, process starvation, deadlock, livelock and race-hazards are *real* problems. Failing to read from one of two light sensors may prevent a robot from following a line, or cause it to wander off a student's work surface (sometimes much to their delight). We think using a language like occam-π on the LEGO Mindstorms introduces students to the challenges and delights of concurrent real-time system design; and it provides a powerful tool for executing those designs.

3.3 Learning should be constructive

We agree with Einstein when he said: "Things should be made as simple as possible—but no simpler." Relevant here is that there is no reason why learning to program in the concurrent paradigm should be any more difficult than learning any other paradigm. Too many approaches to teaching concurrency are too complex, introduced too late and their value is therefore obscured.

Many examples exist where industry-standard libraries like PVM[13] or OpenMP[31] have been employed in the classroom[10, 18, 24]. However, all of these industrial-strength packages suffer from the same problem: while their primitives may be few and simple, correct and safe application of them can be surprisingly hard. They are designed for professional software engineers, not first-year undergraduates; the usability issues that can result from this mismatch may have a significant impact on what students can accomplish[9].

In an attempt to deal with this dissonance, pedagogic libraries like ThreadMentor[8] have been developed—but all of these (industrial and pedagogic alike) suffer from a larger problem. Libraries provide students with primitives for implementing concurrency in their programs, but they do not help students to *design* solutions with concurrency in mind. These libraries represent the imposing of one computational paradigm (concurrency) in a fundamentally serial paradigm.

occam-π, on the other hand, has concurrency built into the heart of its language, by design, that makes it natural to express ideas about processes, networks, communication, time-outs, non-deterministic choice etc. Furthermore, we only ever need to think about one component process at a time, so that there is true compositionally and, hence, scalability.

Our goal is to make students fluent in concurrent design and implementation, not to teach them how to use one set of primitives for concurrency before they fundamentally understand the paradigm. We encourage students by giving them programming tasks that are fundamentally concurrent in nature: programming little robots, for example. In this respect, we appreciate Lynn Andrea Stein's work in "Rethinking CS101," which involves motivating students to think about agent—and event—based computing sooner, rather than later, in the curriculum[30].

3.4 Learning on the LEGO

Our implementation of occam-π for the Mindstorms opens new possibilities for the teaching and learning of concurrent programming using this small robotics platform. Languages like Not Quite C (NQC)[6] and ROBOLAB[26] provide basic multitasking facilities that students can use, although interprocess communication is difficult and awkward at best. The implementation of Ada for the Mindstorms, developed by Fagin et al., translates Ada programs to NQC, and therefore shares many of NQC's limitations[11]. Despite the existence of concurrency primitives in Ada, Ada/Mindstorms does not currently take advantage of these. Tasking is however a future goal for the Ada/Mindstorms environment[12].

Unlike many languages available for the Mindstorms, our implementation of occam-π will be complete. At the time of writing, all core concurrency mechanisms have been implemented. Capabilities for dynamic memory, mobile processes and channels (allowing the creation of dynamically evolving process networks) are currently work in progress. Other complete languages do exist for the Mindstorms. pbForth is a complete Forth implementation for the Mindstorms by Ralph Hempel[14]. There also exists a complete environment for programming the Mindstorms in C, BrickOS, written by Markus Noga[23]. BrickOS requires GCC to build C programs for the Mindstorms, which is a non-trivial environment to set up and maintain. We would hesitate to use it in the classroom. However, we have made extensive use of BrickOS in our own work, as we host the Transterpreter within it.

We believe Klassner's work on Mnet, a LISP environment for the LEGO Mindstorms, is motivated by concerns regarding authenticity similar to our own[19]. In teaching the fundamentals of classic AI (search, planning, etc.), it is much more interesting to do the work on real robots as opposed to working in a virtual microworld. Further comparison, however, is unfair to both projects: Frank Klassner is interested in motivating students studying AI, while we are looking for an authentic environment for studying concurrency in real-time systems.

4. OCCAM-PI ON THE MINDSTORMS

We have discussed occam-π and the Transterpreter, and how our pedagogic goals relate to other approaches to teaching concurrency. We have also discussed how the Transterpreter relates to other languages and environments available for the Mindstorms, and will now provide a worked example to further ground this discussion in the technologies we are making available to the larger computer science education community.

In all the languages and environments available for the Mindstorms, a robot that can bump-and-wander its way around a room is a simple task. A more difficult challenge is to build a robot which allows the bump-and-wander robot to be interrupted, if it reverses into an obstacle, during its timed reversal sequence. The robot this example is using has two bump sensors, one in front and one at the back. When the robot bumps into an obstacle at the front, it starts a reverse turn. The reverse turn can be interrupted either by a time-out, or at any time the robot reverses into an obstacle, which would be detected by the triggering of the back bump sensor. Due to the use of two separate conditions for termination of the robot's reversal, implementing this program can be a difficult task in many other languages.

occam-π is a language of communicating processes. The primary mechanism for these communications are *channels*, which are unidirectional, synchronizing, unbuffered pipes through which data (or references to data) can be safely passed. These channels can carry everything from single booleans to complex structured data, and serve as explicit synchronization points in occam-π programs, and guarantee the complete absence of data race-hazards, a property enforced by the compiler.

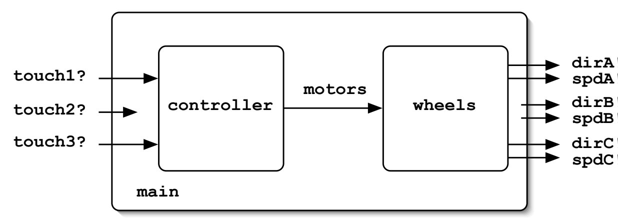

Figure 1: A simple process diagram

```
 1 #INCLUDE "legolib.inc"
 2 PROTOCOL Motors IS BYTE; BYTE; BYTE; BYTE:
 3 VAL INT backupTime IS 1000:
 4
 5 PROC main ()
 6   CHAN Motors motors:
 7   PAR
 8     controller (touch1?, touch3?, motors!)
 9     wheels (motors?, dirA!, spdA!,
10                      dirC!, spdC!)
11 :
```

Figure 2: `main` gets things started

Figure 1 is a simple process diagram that we might develop with our students to represent a collection of processes on a Mindstorms—just two in this example.

There is a `controller` process, containing the control logic, and a `wheels` process, used to mediate communication with the motors. There are also three external input channels, one for each touch sensor, and six external output channels, that directly drive actuators on the wheel motors. These external channels are provided by our *legolib*—see figure 2 (line 1). There is also one internal channel, `motors`, carrying information between the two processes.

Figure 2 lists the top-down presentation of this system. We start by specifying the `main` process that declares the internal channel, and creates and starts the two concurrent sub-processes. Of interest is the declaration of the `Motors` protocol (line 2), which declares that all communications over a channel of type `Motors` will involve sending four bytes: left motor direction, right motor direction, left motor speed, right motor speed; the channel variable `motors` is locally declared in the procedure `main` on line 6.On line 7, we spawn two processes in parallel using the `PAR` construct: the controlling process, and the process that drives the wheels.

In creating the `controller` and `wheels` processes, we see that each have been passed a different end of the `motors` channel. The end of the channel passed to each process is signified by a '?' suffix (for reading) or '!' suffix (for writing). We will use this `motors` channel to set the direction and speed of the two motors attached to the LEGO Mindstorms. Messages passed set the direction and speed of the two motors connected, as programmed by the `wheels PROC` (figure 3).

```
 1 PROC wheels (CHAN Motors moto?,
 2      CHAN BYTE dirA!, spdA!, dirC!, spdC!)
 3   WHILE TRUE
 4     BYTE dLeft, sLeft, dRight, sRight:
 5     SEQ
 6       moto ? dLeft; sLeft; dRight; sRight
 7       PAR
 8         dirA ! dLeft
 9         spdA ! sLeft
10         dirC ! dRight
11         spdC ! sRight
12 :
```

Figure 3: `wheels` handles motor control

A common idiom in occam-π programs is for each process to contain an infinite loop (line 3); in a language that is inherently concurrent, this is not a problem. We see on line 1 the parameter `moto?`, which is our motor control channel; the other end of this channel is connected to the `controller` process (this fact is not relevant to the design and implementation of this process however). In `SEQ`uence (line 5), we read in four bytes from the `moto` channel (line 6), and then in `PAR`allel we set the direction and speed of motors attached to ports A and C on the Mindstorms by the relevant channel communications.

```
 1 PROC controller (CHAN BOOL touchFront?,
 2      touchBack?, CHAN Motors moto!)
 3   WHILE TRUE
 4     BOOL touched:
 5     TIMER clock:
 6     INT curTime:
 7     SEQ
 8       moto ! FWD; FULL; FWD; FULL
 9       touchFront ? touched
10       moto ! BWD; HALF; BWD; FULL
11       clock ? curTime
12       ALT
13         touchBack ? touched
14           SKIP
15         clock ? AFTER curTime PLUS backupTime
16           SKIP
17 :
```

Figure 4: `controller` is the interesting bit

In figure 4 we see the `controller` process; this `PROC`ess does all of the "work" in our example. On lines 1-2, are the parameters—in this case, two input channel ends from the touch sensors and the output end of the `motors` channel. On line 3 is the idiomatic infinite loop, followed by three local variable declarations. The `clock` defined on line 5 has a special type, a `TIMER`. This is treated as a read-only channel from which the current system time (in milliseconds for this platform) is always available.

We begin by moving forward (line 8); we accomplish this by sending library-defined constants for direction and speed down the channel `motors`, which given its `PROTOCOL` must take four bytes of information. Sending data down a channel is accomplished via the syntax

$$< channel > \texttt{!} < valexp >$$

where the result of the value expression on the right-hand side is sent down the channel (as long as the compile-time type check passes).

In the next line, we take advantage of the fact that occam-π channels are synchronized; the `controller` process will block on line 9 until the `touchFront` channel becomes ready with a value—meaning the touch sensor has been pressed. Here, we see the occam-π syntax for reading from a channel,

$$< channel > \texttt{?} < variable >$$

where a value is read from the channel into the variable provided.

Once the `touchFront` channel is triggered, line 10 tells the motors to run in reverse (at different speeds, so we pivot), and the interesting part of the program begins.

The `ALT` construct (line 12) provides for passive waiting for one of two or more events to become ready; it allows the expression of the non-deterministic behavior of a process. In this case, we are waiting for either the back touch sensor (`touchBack`, line 13) or the `clock` (line 15). In particular, we are watching to see if a number of milliseconds equal to the variable `backupTime` (line 3, figure 2) have elapsed since the time this process first read the clock on line 11.

If either the touch sensor or the time-out on the clock becomes ready for communication, we do the same thing in both cases: we drop out of the `ALT` with the `SKIP` instruction (a no-op). This takes us up to the top of the loop, where we once again set the motors moving in a forward direction, and begin the process all over again.

For information, the source-code for the above occam-π program, is 42 lines long. A Java solution, programmed by an expert colleague, using a standard OO event-driven paradigm, occupies 165 lines.

5. CONCLUSIONS AND FUTURE WORK

The Mindstorms is an excellent introductory platform for introducing students to the issues involved in developing concurrent and parallel programs. It is an authentic application of these ideas, as even small robots have big problems dealing with the immediacy and concurrency of the real world.

Our plans for future work are driven by technological, pedagogic and research concerns. The Transterpreter will continue to grow until it supports the full extent of occam-π. Additionally, we look forward to experimenting with the Transterpreter as a platform for exploring grid computing in the context of dynamic clusters—another natural application of occam-π. We have begun collecting resources for teaching and programming with occam-π at `www.transterpreter.org`, intended for use by instructors and students interested in our work. Lastly, more research of this nature regarding the use of robotics for motivation and concurrency in the curriculum is absolutely necessary.

6. ACKNOWLEDGMENTS

We would like to thank David Barnes for his comments and feedback on an early draft of this paper. In addition, thanks to Peter Welch at Kent and Mark Smith of Colby College for their insights during the revision process.

7. REFERENCES

[1] G. Andrews and R. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Publishing Company, Inc., 1993.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[3] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts. *Handel-C Language Reference Guide*. Oxford University Computing Laboratory, August 1996.

[4] F. R. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.

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

[6] D. Baum and R. Zurcher. *Dave Baum's Definitive Guide to Lego Mindstorms*. APress L. P., 1999.

[7] R. D. Beer, H. J. Chiel, and R. F. Drushel. Using autonomous robotics to teach science and engineering. *Commun. ACM*, 42(6):85–92, 1999.

[8] S. Carr, J. Mayo, and C.-K. Shene. Threadmentor: a pedagogical tool for multithreaded programming. *J. Educ. Resour. Comput.*, 3(1):1–30, 2003.

[9] S. Clarke. Measuring API usability. *Dr. Dobbs Journal*, May 2004.

[10] J. C. Cunha and J. Lourenco. An integrated course on parallel and distributed processing. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 217–221. ACM Press, 1998.

[11] B. Fagin. An Ada interface to LEGO Mindstorms. *Ada Lett.*, XX(3):20–40, 2000.

[12] B. Fagin. Ada/mindstorms 3.0: A computational environment for introductory robotics and programming. *IEEE Robotics and Automation Magazine*, 10(2):19– 24, June 2003.

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 Users Guide and Reference manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1994.

[14] R. Hempel. The pbForth Home Page, 2004. `http://www.hempeldesigngroup.com/lego/pbForth`.

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

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

[17] M. C. Jadud. Teamstorms as a theory of instruction. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference*, volume 1, 2000.

[18] L. Jin and L. Yang. A laboratory for teaching parallel computing on parallel structures. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 71–75. ACM Press, 1995.

[19] F. Klassner. Enhancing lisp instruction with RCXLisp and robotics. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 214–218. ACM Press, 2004.

[20] B. L. Kurtz, H. Cai, C. Plock, and X. Chen. A concurrency simulator designed for sophomore-level instruction. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 237–241. ACM Press, 1998.

[21] F. G. Martin. *Circuits to Control: Learning Engineering by Designing LEGO Robots*. PhD thesis, Massachusetts Institute of Technology, 1994.

[22] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[23] M. Noga. BrickOS for the LEGO Mindstorms. `http://freshmeat.net/projects/brickos/`.

[24] Y. Pan. An innovative course in parallel computing. *Journal of STEM Education*, 4(0), 2003.

[25] P. B. Petre M. Using robotics to motivate 'back door' learning. *Education and Information Technologies*, 9(2):147–158, 2004.

[26] Pitsco LEGO Dacta. The ROBOLAB system, 2000. `http://www.pitsco-legodacta.com/Products/robolab.htm`.

[27] M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Micorworlds*. MIT Press, 1994.

[28] S. Robbins. Starving philosophers: experimentation with monitor synchronization. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 317–321. ACM Press, 2001.

[29] P. V. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[30] L. Stein. Challenging the computational metaphor: Implications for how we think, 1999.

[31] The OpenMP Architecture Review Board. OpenMP Specifications version 2.0. `http://www.openmp.org/drupal/mp-documents/cspec20.pdf`.

[32] D. Wood and P. Welch. The Kent Retargetable occam Compiler. In B. O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166. IOS Press, Netherlands, 1996. ISBN: 90-5199-261-0.