Patterns for Programming in Parallel, Pedagogically

Matthew C. Jadud
Franklin W. Olin
College of Engineering
Needham, MA, 02492
matt@transterpreter.org

Jon Simpson
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF
jon@transterpreter.org

Christian L. Jacobsen
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF
christian@transterpreter.org

ABSTRACT

Pipeline, *Delta*, and *Black Hole* are three simple patterns used in concurrent software design. We recently presented these and other patterns for parallelism at a nine-hour workshop for professional embedded systems developers. By grounding these patterns in the context of robotic control on the LEGO Mindstorms, we provided an engaging and enjoyable educational experience for our "students," and reaffirmed that small, powerful languages have a place in education for beginners and experts alike.

Categories and Subject Descriptors

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

General Terms

Human Factors, Languages

Keywords

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

1. INTRODUCTION

For the last twenty years, concurrent software design has been part of the curriculum at the University of Kent in Canterbury, England. Over the past five years, we have experimented with introducing a sequence of laboratories regarding concurrent design focused on the problem of robotic control[4]. We chose robotic control for the simple reason that every robotic control system is fundamentally a concurrent system. A robot must always do three things "at the same time," meaning it must read from its inputs (Sense), compute over those inputs (Think), and control motors and other actuators (Act). In the programming language occam-π [7], this can be modeled as three processes that execute concurrently, flowing data from one process to the next (Figure 1).

SIGCSE'08, March 12–15, 2008, Portland, Oregon, USA.

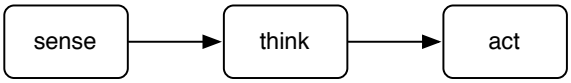

Figure 1: A robot's most basic process network.

We recently had the opportunity to distill material from our classroom at the University of Kent into a three-day, nine-hour workshop for professional embedded systems developers working for an IT solutions provider in Vienna, Austria. The workshop, titled "Patterns for Concurrency," compressed the essential aspects of software design in a concurrent language like occam-π, while retaining the hands-on nature of our semester-long robotics experience. This was remarkably challenging for a number of reasons. Only having nine hours of total contact time was one concern, and the fear that we might be seen as "wasting the developer's time" was another. We were being offered 25% of a work week for each of 10 full-time developers, which represents a substantial amount of time to take away from a group's normal development activities.

We chose to maximize the amount of hands-on programming that the developers would engage in, and developed a series of exercises that exemplify patterns for parallel software development that they could work through in pairs. Furthermore, we decided to keep the hands-on nature of the labs we had developed at Kent, and have all of their software development target the LEGO Mindstorms robotics kit.

The use of robots was a risky proposition. First, transporting 10 LEGO Mindstorms kits from Canterbury, England to Vienna, Austria presented logistical problems. Second, while we hoped the professional developers would appreciate working with real hardware (as opposed to a simulator), we were concerned that they might consider the LEGO robots "toys," and as a result think that we were not introducing powerful concepts that they could use in their own practice. And lastly, we were afraid that the use of a relatively unknown language like occam-π would turn them off even further.

When asked what they would keep from the workshop, half of the developers explicitly commented that they would *definitely* keep the LEGO Mindstorms: they thought they were fun (and sometimes funny) to work with, and—as embedded systems developers—they always preferred working with a real hardware platform whenever possible. When asked what they would change, they made constructive suggestions, but also acknowledged the time factor; one of the

participants said "I would add some bags of time, big bags." Lastly, when asked what they would throw away, they consistently said words to the effect of "nothing."

What follows is a three-paragraph introduction to the programming language occam-π, and our motivation for presenting concurrent and parallel software design on a small, commercial off-the-shelf robotics platform like the LEGO Mindstorms. To give a sense for how we structured the exercises the developers worked through, we share three of the patterns presented on the LEGO: *Pipeline*, *Delta*, and *Black Hole*. We then close with a brief discussion of how this material scales beyond these simple patterns presented here, and some final comments from the developers regarding their experiences learning a new programming language and paradigm on new hardware in three short days.

2. occam-π IN THREE PARAGRAPHS

The producer/consumer pattern is simple in design and infinitely complex and varied in execution. It is at the heart of the communications between independent processors in embedded systems (in the form of I2C and SPI protocols), inter-process communications on typical desktop computers, and interactions between machines on the Internet. In languages like occam-π, this is a naturally occurring pattern, and can be instantiated in parallel with only a few lines of code. In occam-π, a process that does nothing but produce the number forty-two would look like:

```
PROC producer (CHAN INT ch!)
  WHILE TRUE
    ch ! 42
:
```

The corresponding consumer would look like:

```
PROC consumer (CHAN INT ch?)
  INT tmp:
  WHILE TRUE
    ch ? tmp
:
```

The producer, in an infinite loop, outputs the value `42` on the channel `ch`. Output in occam-π is indicated by the 'bang' operator, `!`. This is a blocking communication: execution of this process blocks (and deschedules) until the corresponding read happens in the consumer. In the consumer, a blocking read takes place on the channel `ch` until the channel is ready, at which point the value (`42` in this case) is read into the variable `tmp`. A read on a channel is indicated by the `?`. Both of these symbols come down to us from Hoare's Communicating Sequential Processes (CSP) algebra[2].

The producer/consumer processes are instantiated in a third process, `main`:

```
PROC main ()
  CHAN INT ch:
  PAR
    producer(ch!)
    consumer(ch?)
:
```

In occam-π, indentation indicates block scope. In the process `main`, the **PAR**allel process indicates that the processes `producer` and `consumer` should be executed in parallel, and the **PAR** will exit when all of its child processes have exited. In this particular case, the process `main` never exits, because its two children forever schedule and deschedule, communicating the value `42` over the channel connecting them. It is up to the occam-π compiler and runtime to make sure that this concurrency (on a single processor) or parallelism (on multiple processors) takes place correctly.

These three processes taken together represent a complete occam-π program, and completely implement the producer/consumer pattern.

2.1 Why occam-π on the LEGO?

Put simply, we think that introducing concurrency on a small robotics platform is **authentic**, **constructive**, and **fun**[3].

Introducing concurrent software design in the context of a robot is **authentic** for two reasons. First, robots need to juggle the reading of sensors, computing over those sensor values, and control of motors and actuators continuously. Second, toy robots like the Mindstorms, as well as commercially available robots like iRobot Roomba, must deal with exactly the same problems; regardless of scale, this fundamental need for concurrency does not go away.

We attempt to remain true to the notion of **constructivism**; in particular, we try to structure our exercises so that students must learn on their own some critical aspect of the language or a pattern for designing concurrent solutions to control problems as part of a larger problem[5]. By leaving room for the learner to direct part of their own learning in this process, we hope that the lessons learned are that much more powerful.

Lastly, we believe strongly that the learning activities we give our students should be **fun**. Creative professionals in the real world strive to find interesting and engaging projects that they can *enjoy* working on. We are always working to provide similarly challenging, yet enjoyable tasks that let students express not only their abilities, but also their passion for learning new and interesting things about computing.

3. THREE PARALLEL PATTERNS

Pipeline, *Delta*, and *Black Hole* are the first three patterns for parallel software design that we presented to the developers during the workshop. Each of the patterns was presented in much the same way: a single sheet of paper for each pattern with a process diagram, explanation, and code sample was provided to the developers. In addition, one or more coding challenges were put to them, in which they were expected to employ the pattern as part (or all) of the solution to a robotic control problem. Working in pairs, they would discuss their way through exploring the language (we worked to minimize lecture and maximize the time we spent as "guides on the side"), sometimes with one member of the team exploring the language while the other looked up a detail of the LEGO API or a language feature in the documentation provided.

3.1 Pipeline

The pipeline is a fundamental concurrent pattern in any process-oriented language. Data comes in at one end of the pipeline, is manipulated by one or more processes, and output at the other end. It can easily and safely be extended at the start, in the middle, or the end of the pipe. This pattern is typically represented by a sequence of one or more processes (Figure 2).

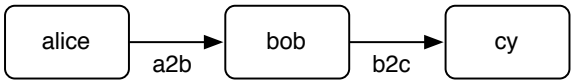

Figure 2: The Pipeline pattern.

This pattern is easily represented in occam-π. For each pair of processes in the pipeline, a channel is declared and used to connect the two processes together. Here is a three-stage pipeline with processes `alice`, `bob`, and `cy`:

```
CHAN INT a2b, b2c:
PAR
  alice (a2b!)
  bob (a2b?, b2c!)
  cy( b2c?)
```

The processes at the end of the pipeline typically only have one channel end, as the pipeline fits into a larger process network. In our example, `alice` has only the output end of a channel, while `cy` has an input end. Processes in the middle of the pipe will have the input end of the channel connecting them to the process on their left, and the output end of a channel connecting them to the process on their right.

3.1.1 In the classroom

When teaching this pattern, we introduce it in both the large and in the small. In the large, we introduce the notion of a robot having three primary functions: to take inputs from the world, to process them in a "brain" of some sort, and to then output commands to its motors and other actuators. This pattern of Sense → Think → Act is a common, concurrent pattern in all robotic control applications (Figure 1).

In the small, we talk about specific tasks, like processing sensor data. Students might be reading from a light sensor (which reports an intensity between 0 and 100), and they might want to only send a message to the "Brain" process when that value is above some value (say, 50). This ends up looking like Figure 3.

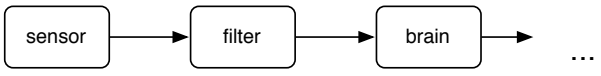

Figure 3: A typical (small) robot control pattern

3.2 Delta

The *Delta* pattern allows replication of data, facilitating multiple pieces of a network to process or react to it. A `delta` process effectively copies all input to two outgoing channels, introducing a one-stage buffer while doing so. A typical `delta` is shown in Figure 4.

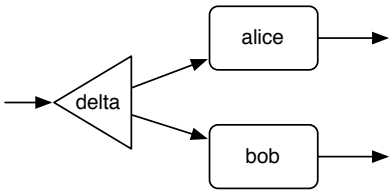

Figure 4: Delta

Due to occam-π's communications operators, writing a `delta` is straightforward. Three channels are declared, and values are read from the first of these channels. After reading a value from the input channel, that value is passed along on both of its output channels in parallel.

```
PROC delta (CHAN INT in?, out1!, out2!)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      PAR
        out1 ! x
        out2 ! x
:
```

3.2.1 In the classroom

On the robotics platform, the `delta` is introduced as a way for the process network to expand beyond a simple pipeline and allow components within the pipeline to maintain their simplicity, rather than doing multiple things in a single process. Taking the example of a pipeline control program from Figure 3, the student may wish to display the sensor value, as well as filtering and passing messages to the `brain` process. By using a `delta` process, a copy of the sensor reading can be used both by the `brain` and by the `screen` process in parallel.

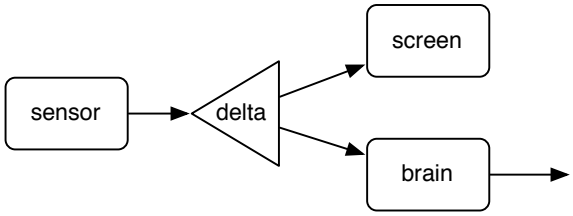

Figure 5: Delta in practice

The *Delta* pattern, at this stage in the learning process, can be seen as critical to maintaining the idea of many simple processes and promoting the idea of parallel composition, instead of adding complexity to existing processes.

3.3 Black Hole

The *Black Hole* pattern allows us to consume data from upstream and make it disappear into the void. In terms of plumbing, it is a "sink" or "drain": data goes in, and it never comes out. In a picture, it looks like Figure 6.

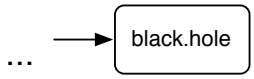

Figure 6: The Black Hole

We have already demonstrated the code for a `black.hole` process in this paper: the consumer on page 1 is a canonical black hole process. A `black.hole` process continuously reads from a channel into a temporary variable, and then does absolutely nothing with that value.

3.3.1 In the classroom

The *Black Hole* pattern may not seem obviously useful at first glance, nor may it seem like a "pattern," given that it is a pattern composed of a single process. As a program evolves, however, this pattern becomes immensely useful. Without the `black.hole` process, shrinking a process network (in particular, removing a `delta`) is a relatively complex endeavor. For example, if a programmer wants to remove the `screen` process from their network (building on the previous example), she must first remove the `delta` and the `screen` process, remove the channel declaration for the channel that connected them, and then re-wire the `sensor` and `filter` processes together. This pruning and rewiring of the process network is captured in Figure 7, part (a).

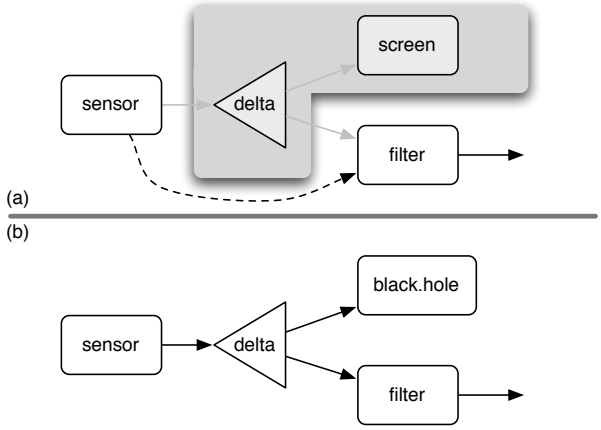

Figure 7: `black.hole`s are easier to create than pruning and rewiring a process network.

Alternatively, the programmer can simply remove the `screen` process, and instead insert the `black.hole`. This process consumes and throws away all the data flowing into it, which "plugs" the second branch in the `delta`, and prevents the `delta` from deadlocking because one of its outputs is not being read (Figure 7, part (b)). Therefore, it is not so much as a critical pattern in the initial design of a process network, but moreso a critical pattern to recognize as a process network *evolves*. Ultimately, a clean process network will not have any `black.hole` processes in it, but we teach our students how to use them so they can productively spend their time developing interesting programs, as opposed to getting bogged down in a continuous network refactoring process.

3.4 Bringing things together

The patterns for concurrency that we introduced are not intended to be used in isolation. occam-π processes are easily composed to produce larger networks of processes, as channels provide natural points to "plug together" one or more processes (or networks of processes). Therefore, each one of these patterns becomes a directly usable building block for writing larger and more interesting programs.

The developers worked towards building robots that were able to express a number of behaviors simultaneously. First, they worked towards building a line-following robot. Then, they added an additional light sensor that would alter the behavior of the robot based on the ambient light in the room. The critical thing about this addition was that they were encouraged to leave as much of their existing process network in place as possible while making these changes. This is actually a very achievable goal when programming in occam-π, as the language encourages a clean separation of concurrent concerns. In the world of robotic control, this layered architecture is generally referred to as a *behavioral control architecture* or the *subsumption architecture*.

4. FROM APPLICATION TO REFLECTION

To close our workshop, we set aside an hour to reflect on what the developers had covered in the workshop. They had spent two intense days working with a new language and a new hardware platform implementing patterns for concurrency many of them thought they knew well. To put put their reflection in context, we encouraged them to consider a problem they knew well—the implementation of software for the control of telephones—and consider how they would do this familiar task in a new language and paradigm.

As they had just seen the patterns that make up the building blocks of concurrent and parallel software design, we presented some additional material to put it in a larger context. Our group has been experimenting with the use of the subsumption architecture for concurrent runtime design and robotic control[6]. Based on the work of Rodney Brooks, the subsumption architecture involves the layering of concurrent processes, each of which encapsulates some simple behavior, and through the composition and layering of those, more complex behaviors can be expressed[1].

In conjunction with the patterns they had seen, these ideas provided the developers with a powerful set of tools to think about and discuss how they develop embedded control software. A process they were intimately familiar with suddenly became new again, and as one developer said, "it is like flipping my mind upside down." Far from being frustrated, they found the process refreshing and challenging—they were taking new knowledge and using it as a lens to reflect on and rethink the design and implementation of software they knew well.

5. CONCLUSION

There are many differences between a professional embedded developer with 10 or more years experience and a first- or second-year undergraduate. For the professional, a three-hour workshop where they get to dive into a new language on a small, yellow robot was not only novel, but entertaining and fun. In taking material we originally developed for classroom use into that "real world," we have returned with some very encouraging feedback.

We asked all ten of the developers questions relating to their opinion of the workshop and what effects it was likely to have on them in the future. When asked to indicate **Y**es, **N**o, or **M**aybe to the statement "This workshop was a good use of my time," every one of the ten developers taking part in the workshop said **yes**, it was a good use of their time. Every one of the developers also said **yes**, they would recommend this workshop to their colleagues. This kind of unanimous endorsement was very encouraging.

Eight of the ten developers said that the workshop provided them with new perspectives that will influence their professional practice; two reported *maybe*. The same eight also said they would experiment further with the tools and materials provided, and that they would also continue reading in the supporting papers and documentation that we had collated and presented to them over the course of the three days; the same two said *maybe*.

This experience has encouraged us to continue to develop our ideas and materials regarding the teaching of concurrency. We believe that the grounding of these ideas in a real-world context (like the LEGO Mindstorms) is a critical part of motivating the need for concurrent software design. And the developers' reactions to the LEGO were so encouraging that we are loathe to give up the tactile and playful interactions that this platform encourages in the classroom. In the end, it's just too much fun for everyone involved... and that seems to be important.

Acknowledgements

The work presented here builds upon countless years of efforts by many. Thanks are due to Damian Dimmich for helping with the delivery of the workshop itself, as well as his ongoing contributions to the software that makes this kind of work possible. Thanks also to Ralph Miarka for hosting our workshop, and to Peter Welch and the rest of the Concurrency Research Group at Kent for their ongoing efforts in maintaining and improving occam-π.

Obtaining the software

If you are interested in exploring concurrency on small robots like the LEGO Mindstorms, we suggest you check out both occam-π (`www.occam-pi.org`) and the Transterpreter project (`www.transterpreter.org`). From the Transterpreter site, you can download a complete development environment for occam-π programs that allows you to write occam-π programs that run on computers running Windows, Mac OSX, or Linux in addition to the LEGO Mindstorms RCX. The Transterpreter is an actively developed open-source project made available under the GPL.

6. REFERENCES

[1] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[2] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Upper Saddle River, NJ, USA, 1985. ISBN: 0-13-153271-5.

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

[4] C. L. Jacobsen and M. C. Jadud. Concurrency, robotics and robodeb. *2007 AAAI Spring Symposium on Robots and Robot Venues: Resources for AI Education*, 2007.

[5] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, New York, 1980.

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

[7] P. Welch and F. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, Apr. 2005.