Safe Parallelism for Robotic Control

Matthew C. Jadud
Allegheny College
Meadville, PA 16335, USA
matt@transterpreter.org

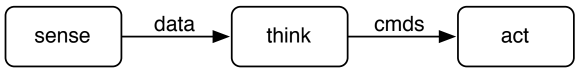
Christian L. Jacobsen, Carl G. Ritson, Jonathan Simpson
University of Kent
Canterbury, Kent CT2 7NF, UK
{christian, carl, jon}@transterpreter.org

***Abstract*—During the Spring 2008 semester at Olin College, we introduced the programming language occam-pi to undergraduates as part of their first course in robotics. Students were able to explore image processing and autonomous behavioral control in a parallel programming language on a small mobile robotics platform with just two weeks of tutorial instruction. Our experiences to date suggest that the language and tools we have developed allow the concise expression of complex robotic control systems, and enable the integration of events from the environment in a consistent and safe model for parallel control that is directly expressed in software.**

I. Introduction

As part of their first course in robotics we introduced Olin students to the programming language occam-pi, a programming language which directly implements a message-passing model of concurrency. Using this language on the Surveyor SRV-1 (a small mobile robotics platform), students explored the basics of behavioral control, vision processing, and interfacing with hardware. This was made possible through a combination of features exposed by the programming language occam-pi and the underlying virtual machine we have developed, the Transterpreter[1], [2].

occam-pi is a descendant of the programming language occam, developed in the early 1980's for the Transputer[3]. This small language is formally grounded in Hoare's Communicating Sequential Processes algebra, and provides explicit constructs for specifying processes that the programmer wishes to see run in parallel[4]. For example, we might claim that a small robot must sense, think, and act, and that data flows from the robots sensor inputs to its "brain," which then sends messages to its actuators. This network of three parallel processes might be depicted as:



This image looks sequential. In occam-pi, the programmer would explicitly indicate that these three processes are executing in PARallel, and are connected by channels that carry messages between them:

```
1 CHAN data, cmds:
2 PAR
3   sense(data!)
4   think(data?, cmds!)
5   act(cmds?)
```

In this short piece of code, we see several abstractions provided by the occam-pi programming language. Line 1 declares two channels (think "virtual wires") that will carry information between the processes `data` and `cmd`. Line 2 declares a `PAR` block: everything indented underneath the `PAR` will execute in parallel. Each of the instantiated processes under the `PAR` (lines 3-5) hold one end of a channel. The `!` indicates that the process holds the output, or sending end of a channel, and the `?` indicates that the process holds the input, or receiving end of the channel. When writing occam-pi programs, the intention is that there is a close correspondence between the code and its associated network diagram.

The message-passing model of computation is naturally captured in a language like occam-pi, which has explicit support for parallelism. In languages like C, Java, and Python, threads are the most common mechanism by which concurrency is introduced. As Edward Lee argues in *The Problem with Threads*, introducing threads "throws away" determinism and predictability in the execution of concurrent and parallel programs[5]. This makes it very difficult for the programmer to reason about the execution of their code. Likewise, Boehm discusses in his 2004 paper *Threads Cannot be Implemented as a Library* that relegating threads to libraries has serious implications for correctness[6]. When your threads come from a library, optimizing compilers can introduce errors in a multi-threaded/multi-core regime simply because it is impossible for the compiler to analyze the concurrency in your code. As will be discussed in Section III, the occam-pi compiler can not only reason about the correctness of our parallel algorithms, it eliminates entire classes of hazard traditionally associated with programs in this space.

Our rationale for introducing Olin students to occam-pi stemmed from our desire to have students explore parallel models for robotic control. Our primary goal was for them to not only be exposed to behavioral models of robotic control (e.g. subsumption architectures), but to express them in code without becoming lost in control variables and state machines or semaphores and threads. In a two week laboratory, students working in teams of two were able to explore the beginnings of vision processing and subsumptive control. For their final projects, some students went on to navigate mazes and integrate new hardware into their robotics platforms. From an educational perspective, we are confident these explorations would not have been possible in the same timeframe had we given our students a C compiler and a JTAG adapter.

In this paper, we will present the technology that made these learning experiences possible. It is not intended as an evaluation of the classroom environment; instead, we will use our students' learning experiences to motivate a reflection on the programming language occam-pi and the architecture of our bytecode virtual machine. We will begin by describing the hardware used by our students and the architecture of our virtual machine (Section II), followed by how this combination enabled our students to explore parallel architectures for robotic control (Section III). In Section IV, we close with application areas and research directions that we wish to explore further.

II. Hardware and Architecture

The students in the Spring 2008 instance of Robotics at Olin College worked through a sequence of five laboratories over the course of the semester, and chose one of these laboratories for deeper, independent investigation as part of a final project. The laboratory we focus on here was designed to expose them to the rudiments of vision processing and the subsumption architecture as articulated by Brooks[7].

A. The Surveyor SRV-1

For this laboratory, we made use of the Surveyor Corporation's SRV-1 mobile robotics platform[8]. This small robot (5" x 4" x 3", Figure 1) has a 500MHz ADSP-BF537 Blackfin processor, 32MB of RAM, 4MB of Flash, a 1280 x 1024 pixel camera, and an 802.11g/b radio. Students would write and compile compile programs on their laptops and upload bytecode over this wireless link. Once their programs were executing, they could send and receive text as well as receive images from the Surveyor using a plugin we developed for JEdit, an open-source editor written in Java[9].

The Surveyor ships with an open-source firmware that responds to single- and multi-byte commands sent over the wireless link, allowing users to tele-operate their robot out-of-the-box. In addition, the Surveyor supports the upload and interpretation of a C-like language, and very recently added a small Lisp interpreter. However, the firmware uses polling for all of its hardware interactions—serial interaction with the radio is, for example, handled through busy-waiting on the RX/TX buffers. Given that we explicitly wanted to expose our students to a clear model of concurrency, the C and Lisp interpreters that ship with the Surveyor provided inadequate starting points for our explorations.

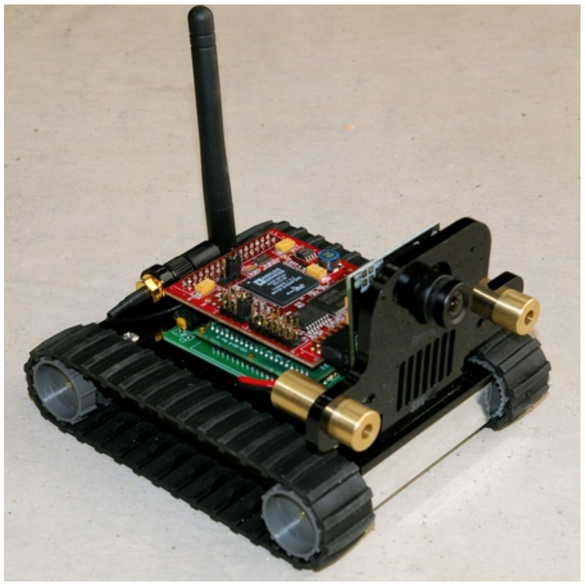


Fig. 1. The Surveyor SRV-1

B. The Transterpreter Virtual Machine

The Transterpreter Virtual Machine (TVM) was developed as a bytecode interpreter for the Extended Transputer Code, an evolution of the bytecode originally executed by the INMOS Transputer. Its small size allows it to execute compiled occam-pi programs on 16-bit (or larger) platforms with as little as 16K of flash and 1K of RAM. The interesting features of our virtual machine architecture are discussed in the context of the Surveyor, but can be safely generalized to any embedded system. The Transterpreter is free software licensed under the LGPL[10].

The TVM is implemented as a fully re-entrant library that allows us to run multiple virtual machine instances concurrently on single-core architectures, and in parallel in multi-core contexts. For the Surveyor platform, this meant that we could completely manage the hardware in one virtual machine and execute user code in a second. Each virtual machine instance has its own registers and run-queues, and can be scheduled cooperatively or pre-emptively. The algorithm for scheduling between the virtual machines is defined by the developer, and the default round-robin algorithm provided is typically sufficient.

C. A Concurrent Firmware in occam-pi

The majority of our firmware is written in occam-pi, with only the minimal work carried out in C to bind the virtual machine to the underlying hardware. On power-up, the processor and peripherals are initialized in C, and then the virtual machine begins interpreting the bytecode representing the firmware. This firmware uses occam-pi channels to provide an interface to the underlying hardware, enabling parallel-safe access to the radio, motors, and other devices attached to the Blackfin. After setting up the camera and other peripherals, the firmware then waits for user bytecode to be transmitted over

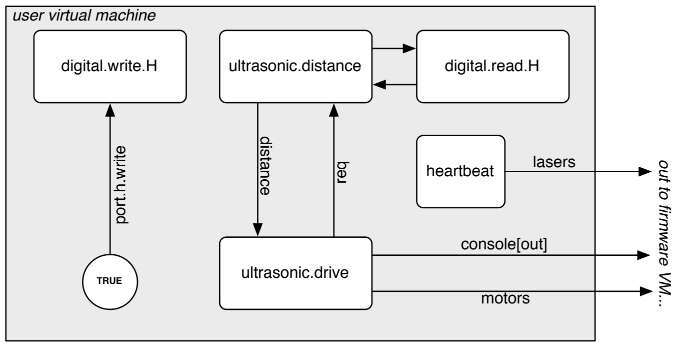


Fig. 2. The process network represented by a student's code.

the radio link (Figure 3). When bytecode representing a user's program is received, a second virtual machine is initialized in RAM, and the firmware hands several channel ends off to the user's application. The user's program is then executed alongside the firmware.

One of the distinguishing features of the occam-pi programming language is that communication between parallel processes[1] happens over well-defined *channels*. These channels provide unbuffered, point-to-point, synchronous communications between any two processes. Processes and channels come to us directly from the Communicating Sequential Processes algebra, and are the primary means of abstraction in the occam-pi programming language.

III. Safe Parallelism for the Beginning Roboticist

The students taking Robotics during the Spring 2008 semester may have had an introduction to Software Engineering in Python, and had probably written a small amount of C. The students were pursuing courses of study that focused on mechanical, biological, systems, and electrical engineering. Put simply, the students enrolled in Robotics had varied programming background, and had many diverse interests in the subject of robotics.

The two-week laboratory based on the Surveyor was intended to introduce students to the basics of vision processing (to demonstrate some of the challenges), and to introduce them to behavioral notions of robotic control in the form of a two-layer subsumption architecture. As part of their preparation for this lab, they had readings from both *Embedded Robotics* by Bräunl as well as Brooks's original technical report on the subsumption architecture[11]. The latter was particularly interesting, because the choice of occam-pi as an implementation language allowed students to directly express in code multi-process networks like those Brooks describes and diagrams in his paper.

Although not subsumptive in nature, what follows is an example derived from an exploration carried out by several students in adding new sensors to the Surveyor. It will provide a context for discussing how the features of our language choice and architecture enable interesting explorations of parallel control, even for the relatively novice programmer.

[1]Think "fibers" or "tiny lightweight threads", not OS processes.

Figure 2 is a diagrammatic representation approximating the work of one pair of students who wanted to extend their robot's capabilities mid-semester. To start, twhey wanted to add an ultrasonic rangefinder; in the first instance, it would act as a non-contact "bump" sensor. This exploration was particularly interesting for two reasons. First, while not a "model solution," it is an authentic representation of what students were able to accomplish with the language after a very brief period of instruction. And second, their solution is interesting because they were able to program the underlying hardware directly from occam-pi, without requiring any changes to the interpreter or firmware.

A. Process Networks

When the students read Brooks's AI memo, they were presented with diagrams much like Figure 2 representing a collection of processes executing in parallel. They were encouraged to use the same kind of visual notation to discuss and design solutions with each-other. Each box in these network diagrams represents an occam-pi process executing in parallel with all the processes around it. The point-to-point, directional channels in occam-pi are indicated by arrows.

In Figure 2, the process network shows that the students created a `heartbeat` process to communicates with the lasers on the robot, and it does not communicate with any of the other processes in their program. The `digital.read.H`, `ultrasonic.distance`, and `ultrasonic.drive` processes communicate with each-other, and only the last process actually communicates with the firmware. The last pair of processes are used to communicate a `TRUE`, or high, value to the pin that powered the ultrasonic rangefinder.

In a picture, we have the entire six-process parallel network representing the student's architecture for a mobile robot with an ultrasonic "bump" sensor.

B. Incremental Development

The first process the students developed from Figure 2 was the `heartbeat` process. It gave them an immediate, visual indication as to whether their robot was running. Every half-second, they would toggle the lasers on or off.

As they added their sensor, they were able to plug together processes to hold a pin high, thus powering their sensor. Lastly, they developed a small network of processes to take readings, processes them, and then drive the motors appropriately.

C. Process Isolation

Every process in occam-pi is an isolated entity. It is this feature of the language that allowed the students to develop their process network incrementally. They could code with the confidence that the process `ultrasonic.drive` would not change the behavior of their `heartbeat` process, or any other process in their network.

In occam-pi, it is impossible for one process to manipulate the state or behavior of another process. The only way for

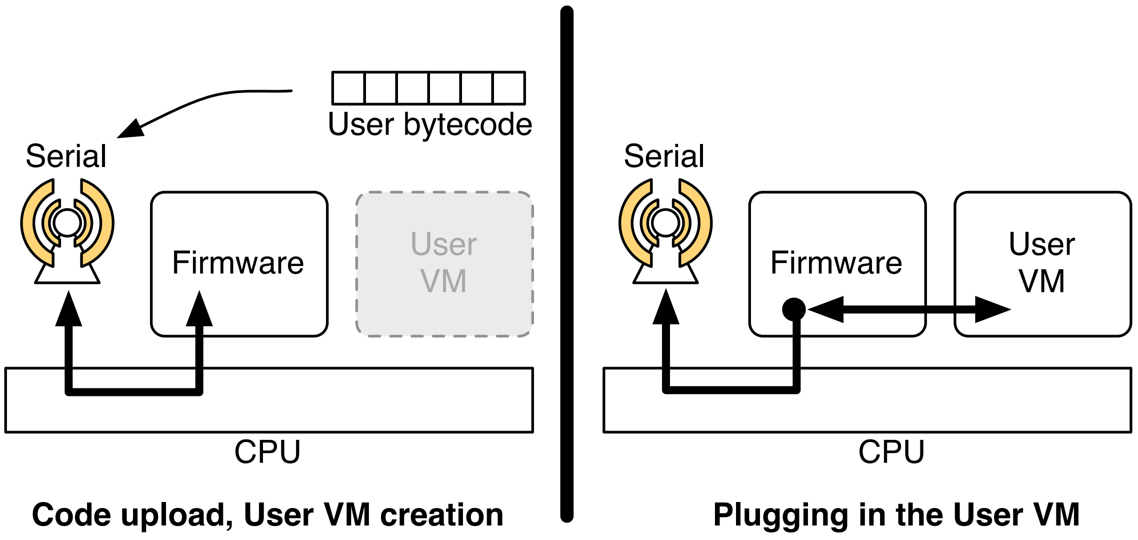


Fig. 3. A dual-VM architecture; channels from the firmware to usercode-VM connected at runtime.

processes to interact with each-other is through their defined channel interface. Any occam-pi program that successfully compiles is guaranteed to be free of basic race hazards (e.g. read-after-write, write-after-write, etc.). It is impossible for one process to read or write directly to or from another processes' internal state.

D. Channels, Synchronization, and Extension

Channels play three critical roles in a process network. First, they carry data between communicating processes: this might be boolean values and integers or structured records with many elements. The act of communication synchronizes the two processes, allowing the programmer to reason about where these two processes are in relation to each-other. Lastly, they provide natural points of extension within a program.

Our firmware relies heavily on the abstractive power of channels. When waiting for user bytecode, the firmware interacts directly with the serial port. Once user code as been uploaded, the connection to the serial port is handed off, allowing the user to send commands to the robot and receive back text and images. From the user's perspective, this is a direct connection to the hardware; however, the firmware actually intercepts all incoming communications, watching for the character '`!`'. The "bang" is used as a kill switch; if the user sends this character, the firmware virtual machine halts the user virtual machine, frees its memory, and begins waiting again for bytecode.

E. Interrupts as Channels

Interrupt handling can be disruptive to the design of embedded firmwares, and forces the developer to think non-deterministically about how their program will execute. In the case of our virtual machine, we can lift these interrupts out of C and into occam-pi in a manner that is consistent with the semantics of the language. This allows us to take what was unmanaged randomness in our code and turn it into a managed part of our firmware's architecture.

Students in Robotics used the `console` channel to send and receive characters from their laptops all semester long. At no point did they need to know that the UART was interrupt-driven. When a character was ready on the input buffer, it was placed at the end of a channel that existed half in C and half in occam-pi. A flag was then set in the VM that the channel was "ready," and the next time through the scheduler, the associated occam-pi process could handle the waiting character, just like it would process data waiting on any other channel.

From the user's point of view, this channel could be read from like any other channel. When no data is ready, the user's process waits. When a character becomes ready, the communications rendezvous completes, and the user's code proceeds normally. This is exactly what would happen with a channel defined entirely between two occam-pi processes; this is what we mean when we say that the interrupt is handled in a manner that is consistent with the semantics of the language.

F. Sleep

Related to interrupt handling is the issue of powering down the Blackfin to conserve power. The nature of the language and run-time environment allow us to detect when nothing is running, and to drop the processor into a low-power, or "idle" mode. When an interrupt occurs (either a peripheral interrupt or a timer interrupt for processes sleeping on the timer queue), we wake back up and begin processing.

The benefit of this approach is that the occam-pi developer does not need to be concerned with power management in most cases, as it comes "for free" in the runtime. Any difficulty related to sleeping the processor can be directly integrated with the run-time environment in a manner that is consistent (and safe) in light of arbitrary user code. However, the benefit of this in terms of power savings is completely application dependent.

G. The Compiler

The occam-pi compiler plays a critical role in all of these language features. One example of how the compiler protects the programmer is by looking for aliased variables or channels between processes (to make sure no parallel writes to one location take place). Another is that it checks to see that two processes are not attempting to "send" down opposite ends of a channel.

No compiler or interpreter for C, Java, or Python is able to provide these kinds of checks as easily. As our goal was to introduce students to parallel approaches to robotic control, these languages were poor choices for teaching and implementation.

H. Deadlock Detection

It is conceivably possible that the compiler could do deadlock *prediction*, but it is NP-hard[12]. At runtime, the problem of deadlock *detection* in our virtual machine is of order O(1). The virtual machine maintains a list of processes that are ready to be executed; when there are no processes on the queue, there are either no processes in the system, or the processes that exist must be waiting on a channel communication. In either case, no process can spontaneously migrate to the run-queue, and we must be in a state of deadlock.

In practice, we must do some work. In particular, we must exclude firmware processes from our check; this is easy, because each VM has its own run-queue. Slightly trickier is that we must check to see that none of the user's processes are waiting on a channel that is tied to a hardware interrupt. For example, a user's code may appear to be deadlocked, but one of the processes are waiting on a communication from the serial port, an interrupt-driven sensor, or some other peripheral. The user's VM will have a run-queue of length zero, but in truth, it is simply waiting for a random, external event—at which point, the network will "come alive" again.

While this sounds complex, it is very straight-forward in practice. The result, from the student's perspective, was that they would sometimes have their programs exit with the error "Deadlock!", their VM would shut down, and the firmware would begin waiting for a new program. In C, their code would simply stop responding. In occam-pi, with bytecode running on a pair of virtual machines, we can detect this run-time error, report it to the developer, and they can then proceed to reason through their process network to fix the problem. This provided many excellent "teachable moments" in the Robotics course—and the students were able to overcome the problems in their code in minutes, not hours.

IV. Conclusion

During the Spring semester of 2008, we introduced two dozen students to the notion that robotic control is an inherently parallel problem. Through our choice of tools, we were able to present a model of parallelism that was consistently implemented in the language they were using to express their designs. There are many models of parallelism beyond synchronous message-passing. However, we believe there is great value in our students experiencing one simple model of parallelism in an engaging context.

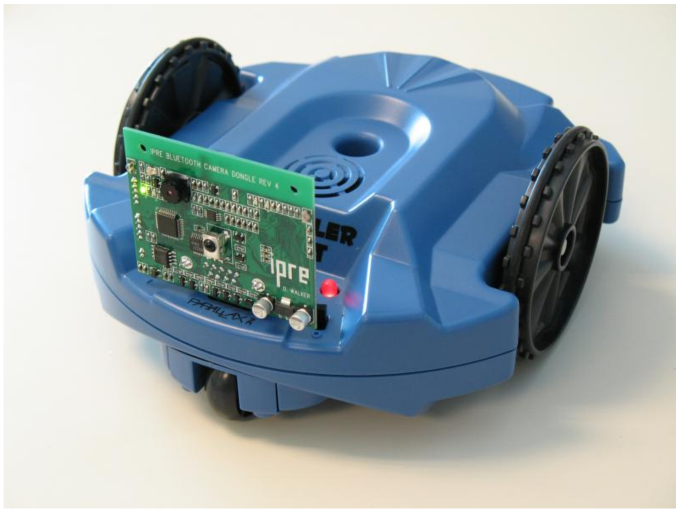


Fig. 4. The IPRE Fluke/Scribbler.

To this end, we have three directions for future work and exploration.

A. Distribution and Documentation

We believe our tools have pedagogic value. For this reason, we have attempted to make them available on a variety of robotic platforms that are available COTS for educational use. The LEGO Mindstorms RCX is one of the first robotic platforms in this space that we targeted. The IPRE Fluke/Scribbler represents our most recent target for a fully supported port[13].

The Institute for Personal Robotics in Education (IPRE) produces the Fluke, an LPC2106-based board that rides on top of the commercial, off-the-shelf Scribbler platform (Figure 4). The Fluke has a small camera, Bluetooth, 8Mb of flash, RS232, SPI/I2C, three infra-red rangefinders, and three user-accessible LEDs. Our port to the LPC2106 (designed by NXP around the ARM7TDMI core) leverages a great deal of the code developed for the Blackfin, as our runtime environment is largely platform agnostic. As a result, developer time in porting from one platform to another is minimized, and we benefit from a robust, well-tested software stack.

With the current IPRE firmware, the Fluke runs in a "tethered" mode, with commands being sent back-and-forth from a Python program running on a laptop or desktop host. In porting to the IPRE Fluke/Scribbler, we will be providing for autonomy where the existing firmware does not. Furthermore, we are creating another low-cost and widely available platform for educators, students, and hobbyists to explore robotic control in a concurrent regime. We are using this effort as an opportunity to improve our internal documentation and update the teaching materials we have developed to date.

B. Performance and Multi-core

The Transterpreter is a small, portable virtual machine for executing occam-pi bytecode. We have done little to optimize the performance of the runtime, but know that with enhanced compiler support, we could compete with C in terms of efficiency and execution speed[14].

Systems at all scales are concerned with efficiency of execution and power consumption. Our explorations so far have begun to demonstrate that a concurrency-aware runtime is able to intelligently safely sleep a processor without user intervention. Likewise, the language and virtual machine should allow developers to take advantage of heterogeneous cores (like on the Cell Broadband Engine[15]) or homogeneous cores without requiring any additional effort on their part. When a developer says `PAR`, it should be possible for them to execute their program concurrently on a single processor system, and for their code to execute in true parallel on a multi-core system. Efficiently handling rich sensor data like images and large LIDARs in a distributed context should feel natural in a parallel, message passing language like occam-pi.

C. Usable Languages for Parallelism

Ultimately, we face a problem of culture. "But Ada already does this!" "Why can't you just use threads?" No-one wants to use a new language (regardless of how and what it can express), especially when they already know how to get things to work with the tools they already have. And as long as the majority of shipping systems are single-core solutions, it is unlikely that tools like ours will gain serious traction outside of education.

As we move to an increasingly multi-core future, it will be critical to have tools that allow us to simply and safely express both soft- and hard-real-time parallel solutions to common problems. The design of these languages and their runtime environments will require us to rethink traditional boundaries between hardware, software, and language design. The focus needs to be on the kinds of programmers who will be using the language or tool, the kinds of programming tasks they will engage in, and the resiliency and safety we desire in their final product. A cultural change in the design and implementation of languages, and a blurring of boundaries between disciplines is needed to address these interesting and open problems in software performance, safety, and reliability.

ACKNOWLEDGMENTS

The teaching and work carried out at Franklin W. Olin College took place as part of Matthew Jadud's visit during the 2007-2008 academic year. David Barrett, Gill Pratt, and the students enrolled in *ENGR:3390 Robotics* are all to be recognized for their support and entrepreneurial spirit in tackling these new ideas with gusto and good will.

At the University of Kent, Peter Welch, David Wood, Fred Barnes, and Damian Dimmich have contributed greatly to the development of the tools described, both in terms of code and excellent conversation and debate.

REFERENCES

[1] P. H. Welch and F. R. M. Barnes, "Communicating Mobile Processes: introducing occam-pi," in *25 Years of CSP*, ser. Lecture Notes in Computer Science, vol. 3525. Springer Verlag, April 2005, pp. 175–210.

[2] C. L. Jacobsen and M. C. Jadud, "The Transterpreter: A Transputer Interpreter," in *Communicating Process Architectures 2004*, ser. Concurrent Systems Engineering Series, D. I. R. East, P. D. Duce, D. M. Green, J. M. R. Martin, and P. P. H. Welch, Eds., vol. 62. IOS Press, Amsterdam, September 2004, pp. 99–106. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2004/2004

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

[4] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

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

[7] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, March 1986.

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

[9] "JEdit," 2008, http://www.jedit.org/. [Online]. Available: http://www.jedit.org/

[10] "The Transterpreter," 2008, http://www.transterpreter.org/. [Online]. Available: http://www.transterpreter.org/

[11] R. A. Brooks, "A robust layered control system for a mobile robot," MIT, Cambridge, MA, USA, Tech. Rep., 1985.

[12] P. Ladkin and B. Simons, "Compile-time analysis of communicating processes," in *ICS '92: Proceedings of the 6th international conference on Supercomputing*. New York, NY, USA: ACM, 1992, pp. 248–259.

[13] "The Institute for Personal Robotics in Education," 2008, http://www.roboteducation.org/. [Online]. Available: http://www.roboteducation.org/

[14] C. L. Jacobsen, D. J. Dimmich, and M. C. Jadud, "Native Code Generation Using the Transterpreter," in *Communicating Process Architectures 2006*, P. Welch, J. Kerridge, and F. Barnes, Eds. Amsterdam, The Netherlands: IOS Press, September 2006, pp. 269–280.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.