Communicating Process Architectures 2007
Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch
IOS Press, 2007

A Native Transterpreter for the LEGO Mindstorms RCX

Jonathan SIMPSON, Christian L. JACOBSEN and Matthew C. JADUD

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NZ, England.

{jon , christian , matt} @transterpreter.org

Abstract. The LEGO Mindstorms RCX is a widely deployed educational robotics platform. This paper presents a concurrent operating environment for the Mindstorms RCX, implemented natively using occam-pi running on the Transterpreter virtual machine. A concurrent hardware abstraction layer aids both the developer of the operating system and facilitates the provision of process-oriented interfaces to the underlying hardware for students and hobbyists interested in small robotics platforms.

Introduction

At the University of Kent, we have access to over forty LEGO Mindstorms RCX robotics kits for use in teaching. Additionally, it is our experience through outreach to local secondary schools and competitions like the FIRST LEGO League[1] that the RCX is a widely available educational robotics platform. For this reason, we are interested in a fully-featured occam-π interface to the LEGO Mindstorms.

The Transterpreter, a portable runtime for occam-π programs, was originally developed to support teaching concurrent software design in occam2.1 on the Mindstorms[2]. In its original implementation, the Transterpreter ran on top of BrickOS, a POSIX-compliant operating system for the RCX[3]. However, as the Transterpreter has grown to support all of occam-π, it has used up all of the available space on the RCX. Given that the Transterpreter will no longer fit onto the RCX along with BrickOS, a new approach is required for a renewed port to the system.

To resolve the memory space issues, we can create a direct hardware interface for the Transterpreter that removes the underlying BrickOS operating system, freeing space to accommodate the now larger virtual machine. To achieve this, we can interact both with routines stored in the RCX's ROM as well as directly with memory-mapped hardware. While it was originally imagined that a C 'wrapper' would need to bind the virtual machine to a given hardware platform, we have discovered that much of this work can instead be done directly from occam-π, thus providing a concurrency-safe hardware abstraction layer.

1. Background

The LEGO Mindstorms Robotics Command eXplorer (RCX) is a widely available educational robotics platform. It takes the form of a large LEGO 'brick' containing a Renesas H8/300 processor running at 16MHz, 16KB of ROM, and 32KB of RAM shared by the firmware image and user programs. There are three input ports for connecting a variety of sensors, three output ports for motors, and an infra-red port used for uploading of firmware and programs. This infra-red port can also be used for communicating with other robots.

1.1. The Transterpreter

The Transterpreter is a virtual machine for occam-π written in ANSI C. At its inception, the virtual machine was designed to bring the occam2.1 programming language to the RCX as an engaging environment for teaching concurrency. The Transterpreter has platform-specific wrappers which link the portable core of the interpreter to the world around it[4]. In the case of the LEGO Mindstorms, a wrapper was originally written to interface with BrickOS[3]. However, there is limited memory space on the RCX, as shown in Figure 1. The choice of building on top of BrickOS was made because it was the quickest and easiest way to get the Transterpreter running on the LEGO Mindstorms; however, it proved impractical for running all but the smallest and simplest of programs.

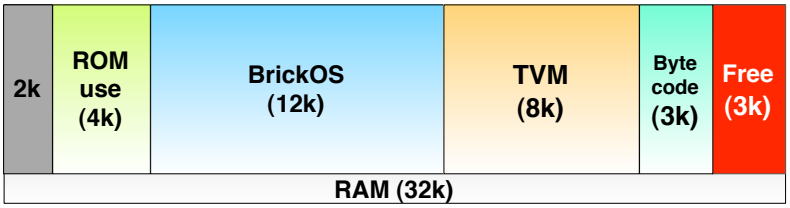


Figure 1. The memory distribution of the original Transterpreter RCX wrapper, using BrickOS.

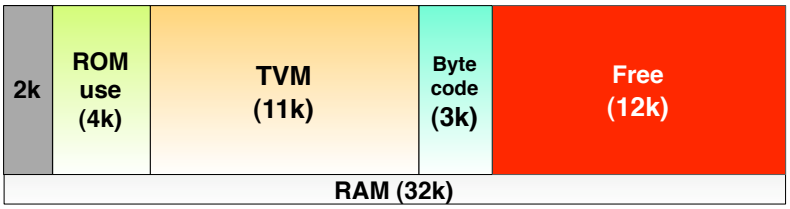


Figure 2. The memory distribution of the native Transterpreter RCX wrapper.

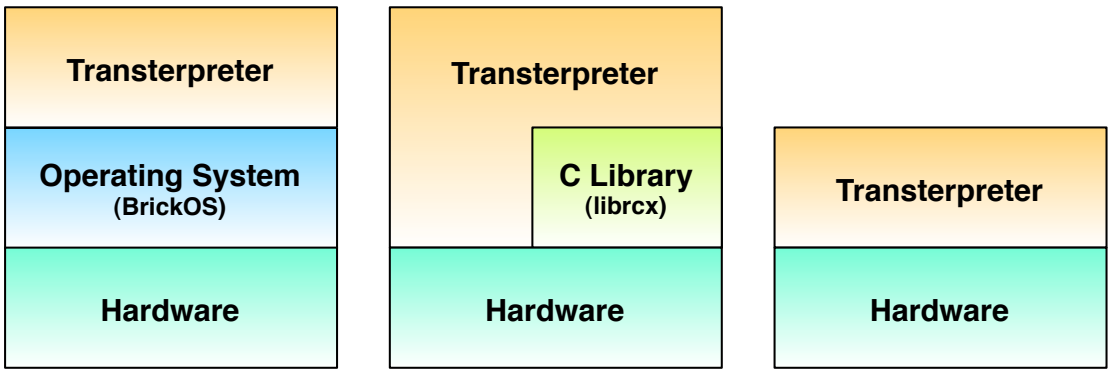
It should be noted that the remaining 3KB of memory space shown in Figure 1, left available after uploading the firmware and user program, was shared to meet the runtime needs of BrickOS, the Transterpreter, and the user's occam2.1 program. As a user's programs grew, this 3KB would be used both by the increased bytecode size of their program as well as a likely increase in memory usage for the larger program.

The Transterpreter virtual machine has grown to support the occam-π programming language[5], an extension of occam2.1 [6]. The extended occam-π feature set is extremely useful for concurrent robotics programming[7]. Unfortunately, supporting these extensions grew the compiled Transterpreter binary by 3KB, and as a result, running the Transterpreter on top of BrickOS is no longer a possibility. By running the Transterpreter natively on the RCX, as shown in Figure 2, we leave 12KB of free memory for the execution of user programs on the virtual machine.

2. Design Considerations

Our design goal is to implement a runtime and process interface for the LEGO Mindstorms RCX, and as such we must provide hardware interfaces to the occam-π programmer. When writing code to interface with the RCX hardware there are three main approaches which can be taken: running with an existing operating system providing the hardware interface, using

a C library to interface with ROM functions, or interfacing directly with ROM functions and memory from occam-π.



(a) Interfacing to hardware through an operating system.

(b) Access to ROM functionality through a C library.

(c) Native interaction with the hardware from occam-π.

Figure 3. Potential design choices for a new RCX port of the Transterpreter.

2.1. On Top of an Existing Operating System

Running on top of an existing operating system for the RCX was previously explored by running the Transterpreter on top of BrickOS (Figure 3(a)). This saved a great deal of work, as BrickOS exposed a high-level hardware API to the Transterpreter. However, this approach introduces additional storage and run-time memory space penalties, and is not practical given the current size of the virtual machine. In Figure 1 on the facing page, BrickOS is reported as occupying 12KB of space on the LEGO Mindstorms; this is *after* 7KB of unnecessary code had been removed from the operating system; reducing this further would be extremely challenging. To support occam-π on the Mindstorms, another approach must be taken, and a new hardware abstraction layer developed.

2.2. Through a C Library

The ROM image supplied with the LEGO Mindstorms contains the 'RCX executive' (Figure 4), which loads at power on and contains routines for interfacing with the hardware. These ROM routines are used by the standard LEGO firmware supplied with the Mindstorms robotics kit.

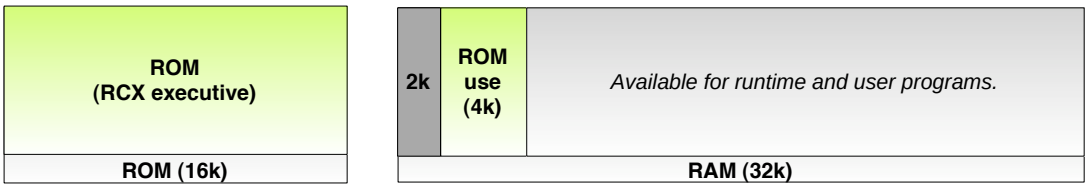


Figure 4. The RCX Executive with both ROM and RAM components, loaded at power-on.

These ROM routines can be exploited to give low-level control over the device without additional memory space penalties, as they are always present in the RCX ROM. However, these routines are not suitable for end-users to program against; they are far too low-level for everyday use.

`librcx` is a C library that wraps all of the available ROM routines, and provides C programmers with a slightly more usable interface to the hardware[8]. One possible approach to porting the Transterpreter to the LEGO Mindstorms would be to do all of the hardware abstraction in C, as shown in Figure 3(b) on the preceding page. The problem with this approach is that `librcx` was designed for use from C and not from a concurrent programming language. Any hardware abstraction layer written in C would not interact correctly with the occam-π scheduler, which could lead to race hazards (or worse), the likes of which are avoided if the abstraction layer is written in occam-π.

2.3. Native Interaction

At its core, `librcx` has five assembly code blocks, each of which calls a ROM routine accepting a specific number of parameters. By exposing these five function calls to occam-π, we can write virtually all of the operating system replacement code in without resorting to C, and leverage the concurrency primitives provided by occam-π (Figure 3(c) on the previous page). This also allows a process interface to the hardware to be exposed naturally, and the 'operating system' components to benefit from a safe, concurrent runtime.

By layering processes, some which provide low-level access to hardware and others that form a higher level API for programmers, we can offer different interfaces to different types of programmer. Novice users might work with the higher level processes, unaware that these processes hide details of the underlying functionality. More advanced users or the system programmer may wish to use the low-level processes to perform specific tasks or interact with the hardware more directly. We discuss this further in Section 4 on page 344.

3. A Concurrent Hardware Abstraction Layer

There is one simple reason for wanting to write as much of our code in occam-π as possible: safety. BrickOS[3] and LeJOS[9], two particularly prominent examples of third-party runtimes for the RCX, both use a time-slicing model of concurrency, where multiple 'tasks' are run on the system at the same time. This time-slicing model is then mapped to a threaded programming model for the user. This is a fundamentally unsafe paradigm to program in, regardless of how careful one is[10].

This would not be a problem, except that robotics programming naturally tends to involve multiple tasks running concurrently. For this reason, threading finds its way into all but the most trivial programs written for BrickOS or LeJOS. In "Viva la BrickOS," Hundersmarck et al. noted that that the default scheduling mechanisms in BrickOS are prone to priority inversion under heavy load[11]. By developing from the hardware up in occam-π, we protect both the operating system developer as well as the end-programmer from these kinds of basic concurrency problems, and strive to provide a safer environment for programming robots like the RCX.

3.1. Implementation Considerations

There are a number of implementation challenges that arise given that we have chosen to natively interface the Transterpreter with the RCX. occam-π provides two ways to access underlying hardware functionality: the Foreign Function Interface (FFI) and placement of variables at memory locations. When used correctly, both allow us to safely interact with the underlying hardware from occam-π.

3.1.1. The Foreign Function Interface

The RCX's ROM routines are made available through five core C functions, which we can access through occam-π's Foreign Function Interface mechanism[12]. Unfortunately, the

RCX hardware is big-endian, while the Transterpreter runs as a little-endian virtual machine. This means that considerable byte-swapping is required on values and addresses being passed back and forth between the occam-π and C, as can be seen in Listing 1.

```
void rcall_1 (int w*)
{
  __rcall1 (SwapTwoBytes(w[0]), SwapTwoBytes(w[1]));
}
```

Listing 1. `rcall_1`, a FFI call that passes its parameters to the RCX's ROM.

The five core calls to LEGO ROM routines, once provided to occam-π via the FFI, allow the majority of the ROM's functionality to be accessed. In cases where return values are required, such as when reading from a sensor, individual FFI calls must be written that marshal the values correctly to and from C (eg. swapping from big-endian to little-endian on their way back into occam-π). For example, the C function `rcall_1()` shown in Listing 1 can be accessed via the FFI from occam-π as shown in Listing 2.

```
-- ROM addresses for sensor access.
VAL [3]INT sensor.addr IS [#1000, #1001, #1002]:
-- Constants for system developer & user programming.
DATA TYPE SENSOR.NUM IS INT:
VAL SENSOR.NUM SENSOR.1 IS 0:
VAL SENSOR.NUM SENSOR.2 IS 1:
VAL SENSOR.NUM SENSOR.3 IS 2:

#PRAGMA EXTERNAL "PROC C.tvmspecial.1.rcall.1 (VAL INT addr, param) = 0"
INLINE PROC rcall.1 (VAL INT addr, param)
  C.tvmspecial.1.rcall.1 (add, param)
:

PROC sensor.active (VAL SENSOR.NUM sensor)
  rcall.1(#1946, sensor.addr[(INT sensor)])
:
```

Listing 2. `sensor.active` sets a sensor on the RCX 'active' through the occam-π FFI.

3.1.2. Variable Placement in Memory

occam-π supports the placement of variables at specific addresses in memory. As inputs and outputs on the RCX are memory-mapped, occam-π processes can be written that interface directly with the hardware by reading and writing to specific locations. Use of variable placement speeds up the system significantly, as the interpreter can read values directly rather than making calls into the RCX's ROM routines through C.

Endianness continues to be an issue when using variable placement with multi-byte variables, as values must again be byte-swapped due to the difference in endianness between hardware and virtual machine. Additionally, as functions of the RCX's ROM are being called, the firmware works with the same memory addresses and care must be taken not to disturb memory values that are in use by the ROM.

The use of variable placement to read the button values from the RCX, as shown in Listing 3 on the following page, is an example of how hardware interactions can be simplified and the number of calls through C to the ROM can be reduced. Additionally the memory

```
PROC run.pressed (CHAN BOOL pressed!)
  INITIAL INT port4.dr.addr IS #FFB7:
  [1]BYTE port4.dr:
  PLACE port4.dr AT port4.dr.addr:
  #PRAGMA DEFINED port4.dr
  WHILE TRUE
    IF
      -- Masking bit 2 of the byte value.
      (port4.dr[0] /\ #02) = 0
        out ! TRUE
      TRUE
        SKIP
:
```

Listing 3. `run.pressed` uses a variable placed in memory to read the 'Run' button state.

read operation can happen much more quickly than an equivalent FFI call and the necessary byte-swapping between occam-π and C that ensues. Endianness issues are avoided in this particular case, as the value of button presses are stored as individual bit flags in a `BYTE` value.

3.2. Advantages of Concurrency

By working with a concurrent language all the way from the hardware up there are advantages gained in both safety and simplicity. The LEGO Mindstorms RCX contains a segmented LCD display, including two segments used to draw a walking person on the screen (Figure 5). When debugging occam-π code running on the RCX it can be hard to tell if the runtime environment has crashed or deadlocked, as printing is frequently not possible once an error has occurred.

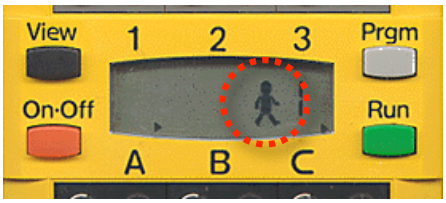


Figure 5. The 'walking figure' on the LCD display of the RCX

By running the `debug.man` process in parallel with other code being tested (like the process `foo()`, shown in Listing 4 on the facing page), it is possible to see that the VM is running, executing instructions and scheduling correctly. Using threading to get the same effect from a C program would have introduced additional complexity, whereas in occam-π it is natural to use concurrency for developing and debugging programs on what is otherwise a "black box" system.

4. Toward a Process Interface

Our goal is to have a complete, process-oriented interface to the LEGO Mindstorms RCX. This involves developing a hierarchy of processes, starting with an API for programmers to use down through direct access to the hardware. Looking just at input, and particularly the LEGO light sensor, we can see the stacking of one process on top of another to provide a concurrent interface to the underlying, sequential hardware. The occam-π code for

```
#INCLUDE "LCD.occ"

PROC debug.man ()
  WHILE TRUE
    SEQ
      -- Sleeping causes us to deschedule
      sleep (500 * MILLIS)
      lcd.set.segment (LCD.STANDING)

      sleep (500 * MILLIS)
      lcd.set.segment (LCD.WALKING)
:

PROC main (CHAN BYTE kyb?, scr!, err!)
  PAR
    debug.man()
    foo()
:
```

Listing 4. The `debug.man` process helps detect VM faults

`light.sensor` is shown in listing 5. This process provides a simple and logical end user interface for reading values from a light sensor, connected to one of the input ports on the RCX.

The `light.sensor` process abstracts across a more generic `sensor` process. Each type of sensor for the LEGO Mindstorms has its own read mode, and may be active or passive. Hiding these details from the end user lets them develop programs in terms of the robotics hardware sitting in front of them, rather than generic interfaces. Layering the processes in this way also means that more advanced programmers can use the `sensor` process directly, as they may have created their own 'homebrew' sensors for the RCX and want to have explicit control over the combination of parameters used to set up the sensor.

```
PROC light.sensor (VAL SENSOR.NUM num,
                   VAL INT delay,
                   CHAN SENSOR.VAL out!)
  CHAN SENSOR.VAL values:
  PAR
    sensor(num, delay, SENSOR.LIGHT, SENSOR.MODE.PERCENT, values!)
    SENSOR.VAL value:
    WHILE TRUE
      SEQ
        values ? value
        out ! value
:
```

Listing 5. The `light.sensor` process abstracts over a generic `sensor` process.

5. Leveraging occam-π: A small example

The most challenging part of robotic control—the scheduling and interleaving of sensor readings, computation over that data, and the control of one or more actuators—is handled transparently when developing programs for the RCX in occam-π running on the Transterpreter. While the example show here is simple, it provides a taste for the kinds of things that are possible when we target a concurrent programming language at a small robotics platform like the RCX.

Figure 6 illustrates a process network where each sensor on the LEGO Mindstorms communicates to a `work` process, which performs a calculation over the sensor data and then sends commands on to the associated `motor` process. Specifically, if the light sensor is reading a particularly light reading (a value greater than 512), the motor is set to spin forwards; otherwise, it is set to spin backwards.

Listing 6 provides the code for this network, and demonstrates the use of a replicated `PAR` for initializing these nine concurrent processes. Furthermore, it illustrates a few aspects of the concurrent API provided for interfacing the LEGO Mindstorms. Types have been defined for all sensor and motor channels: sensors communicate `SENSOR.VAL`s, while motors expect to receive `MOTOR.CMD`s, a tagged protocol that encourages the programmer to be clear about whether a motor is running in a forward or backwards direction. This helps keep our programs semantically clear, and let the type checker help make sure programs are correct. Additionally, the `light.sensor` process allows the programmer to determine how often the sensor will be sampled; in this example, we are sampling the three sensors once every one, two, and three seconds (respectively).

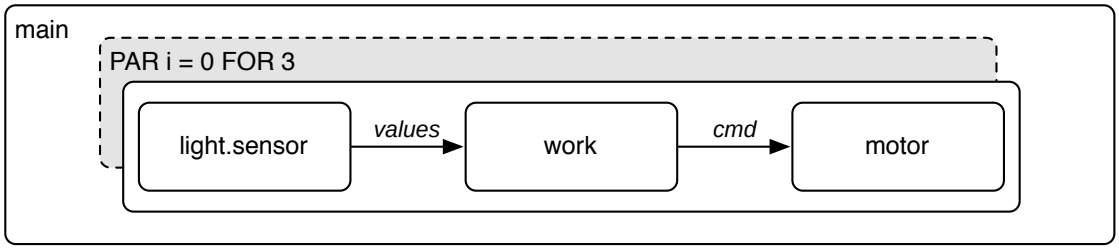


Figure 6. A process network connecting sensors to motors.

This small example does not illustrate any of the more advanced features of occam-π: `SHARED` channels, `MOBILE` data, `BARRIER`s, and so on. It does demonstrate, however, that we can quickly and easily set up many concurrent tasks and execute them directly on the LEGO Mindstorms. As our code grows more complex (as described in [7]), the benefits of a concurrent language and runtime for robotics becomes more apparent.

6. Conclusions and Future Work

Our initial goal was to resuscitate the LEGO Mindstorms RCX as a full-featured platform for occam-π robotics. To achieve this, we had to explore and overcome a number of challenges in developing a new wrapper for the Transterpreter and creating a concurrent, process-oriented interface for the RCX's functionality. However, a great deal more work is required before we have a platform that is casually usable by a robotics hobbyist or novice programmer.

The porting of the virtual machine and development of a concurrent hardware abstraction layer is only the first step towards providing a generally usable occam-π robotics environment. On top of the hardware abstraction layer, we need to write a small operating system

```
#INCLUDE "Sensors.occ"
#INCLUDE "Motors.occ"
#INCLUDE "common.occ"

PROC work (CHAN SENSOR.VAL in?, CHAN MOTOR.CMD out!)
  SENSOR.VAL x:
  WHILE TRUE
    SEQ
      in ? x
      IF
        x > 512
          out ! forward; 5
        TRUE
          out ! backward; 5
:

PROC main ()
  [3]CHAN SENSOR.VAL values:
  [3]CHAN MOTOR.CMD cmd:
  PAR i = 0 FOR 3
    PAR
      light.sensor(i, ((i + 1) * SECONDS), values[i]!)
      work(values[i]?, cmd[i]!)
      motor(i, cmd[i]?)
:
```

Listing 6. A sample program that maps sensor values to motor speeds in parallel.

or monitor that will run along side user programs and provide a basic user interface for the RCX. For example, there are four buttons on the RCX: On-Off, View, Prgm, and Run. At the least, we need to allow users to turn the brick on and off as well as start and stop their programs. The monitor would also need to handle the upload of new programs; the RCX maintains its memory state while powered down, and therefore it is possible to keep the runtime and monitor on the RCX, while the user might upload new bytecode to be executed. This saves the user from the slow and tedious process of uploading a complete virtual machine every time they change their program.

Even with a simple operating system running along side user programs, there is still more work to be done to provide a usable robotics programming environment. Currently, we provide a simplified IDE for programming in occam-π on Mac OSX, Windows, and Linux platforms. This IDE, based on JEdit[1], is extensible through plugins. Our old plugin must be updated to support the uploading of our new Transterpreter-based firmware to the RCX, as well as the compilation of programs for running in this 16-bit environment. This is not hard, but handling the inevitable errors that will occur (failed uploads over IR, and so on) and reporting them to the user in a clear and meaningful manner is subtle, but critical work. We say "critical" because the success of a language is determined as much by the quality of its end-user tools as well as the quality and expressive power of the language itself.

With a usable programming environment in place, we would then like to develop a set of introductory programming exercises using our process-oriented interface to the LEGO Mindstorms. We believe the RCX is an excellent vehicle for teaching and learning about concurrency. While the existing API is already clearly documented, additional materials are absolutely necessary to support novice learners encountering concurrent robotics programming in occam-π for the first time.

[1] http://www.jedit.org/

In this vein, we are ultimately interested in the combination or creation of a visual process layout tool like gCSP[13], POPExplorer[14], or LOVE[15] that supports our process-oriented interface to the RCX. The semantics of occam-π nicely lend themselves to visualization, and a toolbox of pre-written occam-π processes to enable graphical, concurrent robotics programming feel like a natural combination. This could potentially offer an environment where novices could begin exploring concurrency without having to (initially) write any occam-π code at all. In the long run, our goal is to reduce the cost of entry for new programmers to explore occam-π in problem spaces that naturally lend themselves to process-oriented solutions.

Acknowledgements

Many people continue to contribute to the Transterpreter project in many ways. David C. Wood was kind enough to supervise this work as a final year project at the University of Kent. The University of Kent Computing Laboratory and Peter Welch have provided support for hardware and travel for presenting our work. Damian Dimmich continues to develop a native big-endian Transterpreter, and Adam Sampson contributed excellent code that has considerably reduced the size of (and increased the speed of) the core interpreter. Matthew Jadud was supported during this time by the EPSRC-funded DIAS project.

References

[1] FIRST LEGO League. http://www.firstlegoleague.org/.
[2] Christian L. Jacobsen and Matthew 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.
[3] brickOS™Homepage. http://brickos.sourceforge.net/.
[4] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In *Communicating Process Architectures 2004*, pages 99–107, 2004.
[5] P.H. Welch and F.R.M. Barnes. Communicating Mobile Processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.
[6] INMOS Limited. *occam2 Reference Manual*. Prentice Hall, 1984. ISBN: 0-13-629312-3.
[7] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. Mobile Robot Control: The Subsumption Architecture and occam-pi. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 225–236. IOS Press, September 2006.
[8] Kekoa Proudfoot. librcx. http://graphics.stanford.edu/ kekoa/rcx/tools.html, 1998.
[9] LeJOS: Java for LEGO Mindstorms. http://lejos.sourceforge.net.
[10] Hans-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*, pages 261–268, New York, NY, USA, 2005. ACM Press.
[11] Christopher Hundersmarck, Charles Mancinelli, and Michael Martelli. Viva la brickOS. *Journal of Computing Sciences in Colleges*, 19(5):305–307, 2004.
[12] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent at Canterbury, August 1998.
[13] Jan F. Broenink, Marcel A. Groothuis, and Geert K. Liet. gCSP occam Code Generation for RMoX. In *Communicating Process Architectures 2005*, pages –, sep 2005.
[14] Christan L. Jacobsen. *A Portable Runtime for Concurrency Research and Application*. Doctoral thesis, University of Kent, 2007.
[15] Adam Sampson. LOVE. https://www.cs.kent.ac.uk/research/groups/sys/wiki/LOVE, 2006.