Concurrent Event-driven Programming in occam-π for the Arduino

Christian L. JACOBSEN [a], Matthew C. JADUD [b],
Omer KILIC [c] and Adam T. SAMPSON [d]
[a] *Department of Computer Science, University of Copenhagen, Denmark*
[b] *Department of Computer Science , Allegheny College, PA, USA*
[c] *School of Engineering and Digital Arts, University of Kent, UK*
[d] *Institute of Arts, Media and Computer Games, University of Abertay Dundee, UK*
{christian , matt , omer , adam} @concurrency.cc

Abstract. The success of the Arduino platform has made embedded programming widely accessible. The Arduino has seen many uses, for example in rapid prototyping, hobby projects, and in art installations. Arduino users are often not experienced embedded programmers however, and writing correct software for embedded devices can be challenging. This is especially true if the software needs to use interrupts in order to interface with attached devices. Insight and careful discipline are required to avoid introducing race hazards when using interrupt routines. Instead of programming the Arduino in C or C++ as is the custom, we propose using occam-π as a language as that can help the user manage the concurrency introduced when using interrupts and help in the creation of modular, well-designed programs. This paper will introduce the Arduino, the software that enables us to run occam-π on it, and a case study of an environmental sensor used in an Environmental Science course.

Keywords. Transterpreter, occam-pi, Arduino, embedded systems, interrupts, sensing.

Introduction

It is easy to run into basic issues regarding concurrency when programming embedded hardware, and often the use of interrupts is required to handle internal and external events. Interrupts introduce concurrency; concurrency, when coupled with shared state, easily introduces race hazards. When coupled with the non-deterministic nature of interrupt-driven systems, race hazards can be challenging for experienced programmers to diagnose, and rapidly become very difficult for novice programmers to fix.

Traditionally, the novice embedded systems developer would be a student of engineering or computing: an individual committed to learning the *how* and *why* of their problems. The Arduino, a low-cost (less than $30), open hardware platform used by more than 150,000 artists and makers for exploring interactive art, e-textiles, and robotics. It's release has radically changed the (traditional) demographics of the embedded systems world, and despite the (likely) non-technical background of users attracted to the Arduino, it presents them with the same challenge as the budding electrical engineer: how do you make a single processor do two or more things at the same time?

In the past two years, we have focused our development efforts on bringing occam-π to the Arduino platform. occam-π's use of *processes* and *channels* (from Hoare's CSP [1]) provide powerful abstractions for expressing parallel notions ("blink an LED while turning a motor") as well as managing random events in the world ("wait for pin 7 to go high while doing something else"). In addition, processes are an encapsulated abstraction for hardware,

and channels provide well-defined interfaces that allow for the design of systems that (1) mirror the structure of the hardware they control and (2) allow for easy substitution when, for example, an EEPROM module is replaced with an SD card or some other form of data storage.

Our work demonstrates the feasibility of running a virtual machine on an embedded platform with as little as 32KB of space for code and 2KB of RAM, while performing sufficiently well to allow for the development of interesting software and hardware. While there have been other runtime and language efforts targeting devices this size (for example, TinyOS [2], Mantis [3], Contiki [4]), these projects typically target very small research communities. Our goal in porting to the Arduino is to support the diverse and growing community of makers exploring embedded systems by helping them use occam-π to manage the concurrency inherent in their hardware/software systems. Using our tools we have explored several interesting problems that required careful handling of interrupts and real time concerns, including an environmental sensor for monitoring energy and room usage on a college campus (as demonstrated in this paper) and the real time control of an unmanned aerial vehicle, as demonstrated in [5].

1. The Arduino

The Arduino is described as "an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments." [6] In this regard, the Arduino is not just a piece of hardware, but rather an ecosystem consisting of hardware, software, documentation, and above all, its community. The *concurrency.cc board*, which we discuss in Section 1.3, is our own derivative of the Arduino's open hardware design. In total, it is estimated that at the time of writing, over 150,000 Arduino (or Arduino compatible) devices have been shipped to users worldwide.

The software officially supported for programming the Arduino is a custom integrated development environment (IDE) based on the Processing IDE [7], and a set of libraries based on Wiring [8]. Both of these projects are "sister projects" to the Arduino project. The minimalistic Arduino IDE provides support for editing, syntax highlighting, compilation, and uploading of code to a device. Programming is done in C++ and the Wiring libraries provide functions for interacting with the Arduino and a wide variety of sensors, motors, and an endless variety of storage and communication devices.

1.1. The Arduino Community

The Arduino's single greatest asset at this point is not the choice of microcontroller but the community itself. It's large number of enthusiastic developers, users, and merchants make it easy to get started with the Arduino.

The Arduino project's core development team is quite small, and while there is little contribution to the core from the global community, there is a large "external" community developing libraries and examples for the platform. While this code does not typically make it into the distribution, it finds its way onto many websites and into repositories all over the world. Both code and circuitry examples can be found to support users in exploring the control of LCD displays, data storage peripherals, and sensors and motors of all sorts.

The user community helps perpetuate the popularity of the platform. Enthusiastic users "tweet" about their creations (or make creations that tweet), write blog posts, or even generate videos of their creations for sharing on sites like YouTube and Vimeo. These enthusiasts—often artists, makers, and hobbyists with no formal background in computing or electronics—

are keen to help other newcomers to the community, recommending resources and solutions when a new explorer gets stuck.

1.2. occam-π and the `concurrency.cc` *Community*

Although occam is an old language, it has a tiny user community; hence why we have chosen a large and vibrant community of makers and learners in our most recent porting efforts. To grow in this community, we have taken a number of steps—but it will take time and persistence to see the value of these efforts. First, we chose a URL for our project that we hoped would be representative and memorable: `concurrency.cc`. The `.cc` country code was chosen to match that used by the Arduino project (`arduino.cc`) as well as reflect a commitment to open hardware and software (a la the Creative Commons[1]). Mailing lists, open repositories, and easy-to-use bug trackers are not adequate to attract new end-users: we needed to automate the building of packages containing an IDE and toolchain that could be easily installed on all major operating systems.

However, we know the occam-π project, and our efforts to grow our community of users on the Arduino, are hampered by numerous issues. Poor documentation remains our worst enemy: there are few resources for the occam-π programming language available online, and our own documentation efforts are slowed by a lack of contributors. Until it is easy for people to download tools, read (or watch) examples, and implement those examples successfully on their own—and have resources available to let them continue exploring—we expect that it will be difficult to significantly grow the occam-π user community. (We look at resources like the "How to tell if a FLOSS project is doomed to FAIL"[9] and Bacon's "The Art of Community: Building the New Age of Participation"[10] as guides down the long and challenging road to attracting and retaining users in an open, participatory framework.)

1.3. Arduino Hardware and the `concurrency.cc` *Board*

The most popular Arduino boards, the Uno and Mega, are both based on the megaAVR series of processors by Atmel[2]. The specifications for the Uno and Mega can be seen in Table 1 along with those for the LilyPad Arduino, an official 3rd party Arduino variant, and the *concurrency.cc board*, which is Arduino compatible. The megaAVR processors used on these boards are typical embedded microcontrollers with a modest amount of flash and RAM. They all have general purpose (digital) input-output capability, as well as 10-bit analog-to-digital (ADC) conversion hardware, pulse-width modulation hardware (PWM), and support a variety of common embedded protocols (UART, SPI, TWI).

Table 1. Common Arduino configurations

Board	MCU	Flash	SRAM	MHz	UART	ADC	PWM	GPIO
Uno	ATmega328	32 KB	2 KB	16	1	6	6	14
Mega	ATmega2560	256 KB	8 KB	16	4	16	14	54
LilyPad	ATmega328	32 KB	2 KB	8	1	6	6	14
c.cc	ATmega328	32 KB	2 KB	16	1	6	6	14

The standard Arduino board (the Arduino Uno) has three status LEDs (power, serial transmit and receive) as well as one LED that can be controlled via one of the processors pins. This LED can be used when initially working with the board in order to ensure that everything is working correctly: write a program to continuously blink the LED, compile

[1] `http://creativecommons.org/`

[2] Atmel's megaAVR product line: `http://www.atmel.com/dyn/products/devices.asp?category_id=163&family_id=607&subfamily_id=760`

and upload it. Success is evident. When using the Arduino environment blinking this LED is sufficient as a *getting started* exercise, but when using a concurrent language we would like to be able to easily illustrate the concurrent nature of our programs. To do this, we often blink several LEDs in parallel, a feat that cannot be accomplished without attaching further LEDs to the standard Arduino board.

For this reason, demonstrating concurrency on a standard Arduino in a classroom or workshop environment involves connecting several devices (eg. LEDs) to the board using jumper wires that easily fall out during experimentation and use. To remedy this, we have developed an Arduino variant that we call the the *concurrency.cc board* (Figure 1), or "c.cc board" for short. The c.cc board is an Arduino derivative developed by the third author that incorporates a number of features a standard Arduino does not.

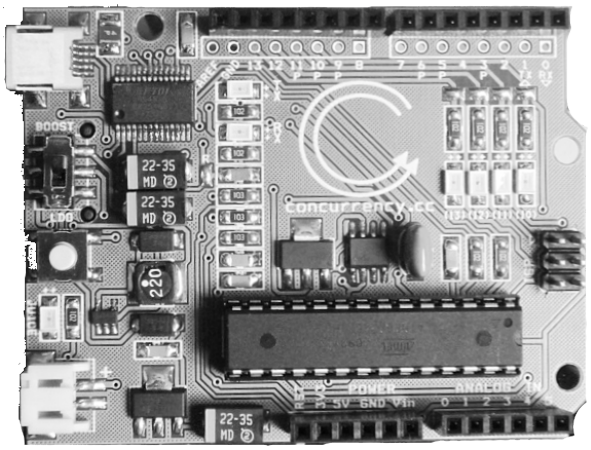

Figure 1. The concurrency.cc board.

First, the c.cc board incorporates a boost converter circuit that allows it to run off low-voltage sources like single AA batteries. Second, it uses a JST connector (as opposed to a larger barrel jack), meaning high energy density lithium polymer batteries can be plugged directly into the board. Third, a mini-USB plug is used, which is now common on many small electronic devices. Finally, four LEDs are designed directly into the board, allowing basic demonstrations of concurrency without the need for an external circuit. These features permit the use of readily available power sources (AA batteries) in teaching environments with an integrated "display" of four LEDs that students can use to see multiple outputs in parallel without needing to construct a separate, error-prone circuit.

1.3.1. Blinking Four LEDs

A first project on any embedded platform is to blink an LED, which demonstrates that code has been uploaded to the processor and that one or more registers that affect the external state can be manipulated. When programming with a concurrent programming language, blinking four LEDs independently should be no harder than blinking one. To blink one LED, we might write the program in Listing 1.

```
1  #INCLUDE "plumbing.module"
2
3  PROC main ()
4    blink (13, 500)
5  :
```

Listing 1. Blinking the built-in LED on and off at a rate of 500ms.

To blink four LEDs at different rates, we would use a **PAR** and four instances of the `blink()` procedure with different output pins and toggle rates (Listing 2).

```
1  #INCLUDE "plumbing.module"
2
3  PROC main ()
4    PAR
5      blink (13, 500)
6      blink (12, 400)
7      blink (11, 300)
8      blink (10, 200)
9  :
```

Listing 2. Blinking four LEDs in parallel at different rates.

For comparison, we have included a C++ program that blinks four LEDs at different rates (Listing 3). It follows the pattern of all programs written in the Arduino environment, which involves implementing both a `setup()` and a `loop()` separately. The former is run once, the latter is run repeatedly until power is removed.

```
1  boolean state[4] = {false, false, false, false};
2  unsigned long prev = 0;
3
4  void setup () {
5    for (int i = 0; i < 4; i++)
6      pinMode(10+i, OUTPUT);
7  }
8
9  void toggle (int pin) {
10   state[pin - 10] = !state[pin - 10];
11   digitalWrite(pin, state[pin - 10]);
12 }
13
14 void loop () {
15   unsigned long time = millis();
16   if (time != prev) {
17     if ((time % 500) == 0) { toggle(13); }
18     if ((time % 400) == 0) { toggle(12); }
19     if ((time % 300) == 0) { toggle(11); }
20     if ((time % 200) == 0) { toggle(10); }
21     prev = time;
22   }
```

Listing 3. Blinking four LEDs at different rates in C++.

Even though the problem is "simple," the second author still made several errors in writing this code. The first mistake was made in implementing `toggle()`. To index into the array `state` (which holds the current state of the four Arduino pins), the value 13 was subtracted to obtain an index instead of 10; this code executed and produced very odd behavior, whereas an equivalent occam-π program would have crashed at runtime. To debug this, a print statement was added to `toggle()` that output the values in the `state` array. Once this was fixed, the serial printing was removed—at which point, the program ceased to function correctly. The reason was that the `loop()` procedure was running *too quickly*, and as a result multiple readings (and therefore multiple pin toggles) were taking place in sub-millisecond timeframes. As a fix, the conditional on line 16 was added.

While we believe there is much more work to be done to support occam-π on the Arduino, we also believe that blinking four LEDs concurrently should be easy. An appar-

ently simple problem ("blink four LEDs at different rates") is not a program that a novice programmer—an enthusiastic artist or maker—can tackle without running afoul of either the language (C++) or the complexities of managing both state and hardware timing.

2. Implementation

The execution of the occam-π language on the Arduino is made possible by the Transterpreter virtual machine [11]. This is the same virtual machine that is used on desktop-class hardware, and it has also been used in the past on the LEGO Mindstorms RCX [12] and the Surveyor SRV-1 [13] mobile robotics platform. The Transterpreter on the Arduino uses a megaAVR specific *wrapper*[3], supporting the ATMega328 and larger processors. The smaller processors in the megaAVR range do not have enough flash (minimum 32KB) or RAM (minimum 2KB) required by the virtual machine. Program execution is facilitated by uploading the virtual machine to the Arduino's flash memory alongside occam-π bytecode, which can be uploaded separately.

This section will deal with specific aspects of the implementation that pertain to the megaAVR family of processors, particularly the implementation of interrupts as well as sleep modes. Details of other aspects of the virtual machine can be found in the papers referenced above.

2.1. General Architecture

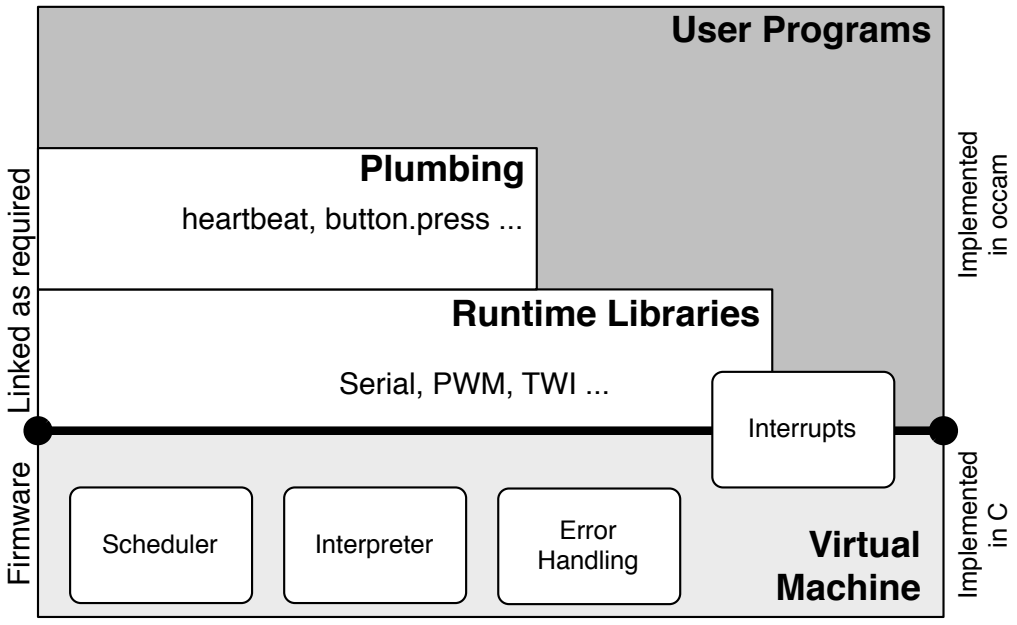

Figure 2. The structure of the software.

Figure 2 depicts the occam-π software stack for the Arduino. Underlying the runtime system is the virtual machine and the portable bytecode interpreter. The virtual machine provides a number of services besides just interpreting bytecode: it loads, parses and checks bytecode located at a specific address, schedules processes, provides a clock for use by both the virtual machine and user programs, and provides error handling (which catches errors and

[3]The *wrapper* contains the platform specific portions of the virtual machine.

attempts to print a useful error message on the serial port). The only service that is exposed directly to the user however, is the `wait.for.interrupt` routine, which is used by both libraries and user code to wait for a particular interrupt to fire.

The runtime libraries, the Plumbing libraries[4], and the user program are compiled into a monolithic bytecode file. Dead-code elimination ensures that unused portions of the libraries or user program are removed from the generated bytecode, and the symbol and debugging information is stripped before the code is uploaded to the device. These measures keep the bytecode files small enough to fit in the limited amount of flash memory available.

The runtime library provides functions for interacting with the serial port, changing the state of individual pins, or using other features of the chip, such as PWM or TWI. This functionality is implemented entirely in occam-π, which has direct access to interrupts and memory (required for manipulating control registers). The Plumbing library is a high level library, and provides the interface we expect most programmers to use. Plumbing provides a process oriented programming interface which lets the user "plumb" together processes such as `heartbeat`, `blink`, `button.press`, `pin.toggle`[5]. The user program can mix the services of the higher level Plumbing library with other libraries as desired. Full access to memory and interrupts is provided to user code, and therefore the user can write code or libraries which interact directly with the hardware.

2.2. Interrupts

The megaAVR range used on the Arduino boards support a wide range of both internal and external interrupt sources. Examples of internal interrupts are those generated by the UART module (serial communication) and the timer. External interrupts are generated in response to a change in the state of one of the processors pins: a pulse from a rotary encoder on a servo, an infrared sensor, or a mechanical switch are all examples of possible external interrupts.

While in simple cases it is possible to poll instead of using interrupts, this style of programming has drawbacks and limitations. For example, if a program needs to count very short pulses from a rotary encoder, it may be hard to ensure that the polling occurs "often enough." Put another way, the polling must be frequent enough to guarantee that a pulse is not missed. As the complexity of a program increases, it becomes harder to ensure that polling can provide the desired resolution. Thus, the use of interrupts can ensure that a program can deal with short signals without complicating its timing logic with frequent polling.

The use of interrupts may also provide lower latency between the signal occurring and the program becoming aware of the signal. This may be useful in situations where a signal must be acknowledged in some way, for example during communication with a device. Avoiding polling also allows the processor to enter one of several low power modes, conserving power while still being able to wake from external or internal interrupts, for example a signal generated by a switch or an internal source such as serial communication or timeout.

2.2.1. Interrupts on the megaAVR Processors

The ATmega328p processor, which is the one most commonly used in Arduino branded and derivative hardware[6], allows all I/O pins to act as interrupt sources. However, only two pins have their own dedicated interrupt vector (the *external interrupts*) and the remaining pins are multiplexed over three further interrupt vectors (the *pin change interrupts*) with a maximum

[4]Full source available for download from `http://projects.cs.kent.ac.uk/projects/kroc/trac/browser/kroc/trunk/tvm/arduino/occam/include/plumbing.module?rev=7082`.

[5]The online (Creative Commons licensed) book *Plumbing for the Arduino* [14] introduces these components and the ways they can be connected to form complete programs.

[6]Other ATmega processors are also used but they are generally similar, varying in amounts of flash, RAM, I/O pins, or internal devices.

of eight pins to an interrupt request. The *external interrupts* can be configured for different levels and edges whereas the *pin change interrupts* are activated by any change in any of the interrupt requests' corresponding pins (each of the multiplexed pins on a *pin change interrupt* can be individually enabled or disabled). An ATmega processor can also generate a number of *internal interrupts* from the UART (serial port), TWI (the Two Wire Interface bus), ADC (analog to digital converter), a number of timers, and other devices.

2.2.2. Interrupts in the Virtual Machine

The Transterpreter for the megaAVR supports all of the interrupts available on the processor: the *external interrupts*, the *internal interrupts*, as well as the *pin change interrupts*, which must use some occam-π support code to demultiplexes the individual pins.

The interrupt support relies on the virtual machine's inner and outer loops. The inner loop is the *run loop* proper, part of the portable virtual machine, which fetches and dispatches bytecodes. The outer loop repeatedly executes the inner loop, while dealing with any platform specific or exceptional tasks that may arise during execution of the inner loop. If an platform specific event occurs, such as an interrupt being raised, the inner loop will terminate, returning control to the outer loop, which can then take the appropriate action.

In order to wire the interrupts into the virtual machine, the processor's interrupt vectors are set up to point to a simple interrupt service routine (ISR). This ISR performs two actions: it sets a flag to indicate to the inner loop that an interrupt has occurred, and it also updates an internal structure to indicate *which* interrupt fired. When the interrupt service routine has finished executing, the inner loop will resume execution and will eventually inspect the status flag, seeing that an interrupt has fired. When the inner loop is at a safe rescheduling point, it will return control to the outer loop. The outer loop can then handle the interrupt condition.

Internally, the virtual machine keeps track of all the available interrupts at all times, as the VM does not know in advance which interrupts will be required by the user program. For each interrupt, the virtual machine tracks (1) whether it has fired, (2) when it fired, and (3) whether a process is waiting for that interrupt. This is implemented using two 16-bit words per interrupt: one holds the identifier of a waiting process (or a value indicating that no process is waiting) and the other holds the time the interrupt fired (or the lowest possible time value to indicate that it has not yet fired). This structure uses up a considerable amount of RAM on the smaller megaAVR parts. For example, on the ATmega328p 12 interrupts are monitored resulting in a structure taking up 24 16-bit words (48 bytes). This table uses up close to 2.5% of the available 2KB RAM on the ATmega328p. It is for this reason that the support for demultiplexing the *pin change interrupts* are not included in the virtual machine (which would provide better performance but at the cost of higher RAM usage). Applications which need to demultiplex *pin change interrupts* can include the relevant occam-π runtime support code.

2.2.3. Interrupts in occam-π

Casual users of occam-π on the Arduino need not be aware that the underlying system makes extensive use of interrupts. In fact, users of the Plumbing library are unlikely to ever realise that interrupts exist! Other users might need to use the interrupt facilities provided by the virtual machine in order to write interfaces to external devices. However, working with interrupts should provide no great surprises to the user, as the underlying mechanisms of the interrupt system closely match the event based semantics of occam-π.

The semantics of the interrupt system provided by the virtual machine is like that of a channel communication. To demonstrate this, we will use a fictional channel type: `CHAN INTR`, which represents an interrupt channel carrying integer value corresponding to the time the interrupt fired. An interrupt channel can be constructed using this (fictional) type: `CHAN INTR int0:`, which a process can now use to wait for an interrupt on interrupt

vector `int0`. Using this syntax, a process would simply perform a read from the channel (`interrupt ? time.fired`) in order to wait for an interrupt. Interrupts, like channel communications, block the reading process until the interrupt (or channel) 'fires.' When the interrupt has fired, the process can continue and will have received the time the interrupt fired into the variable `time.fired`.

For reasons of implementation simplicity and performance, waiting on an interrupt has not been implemented as a channel communication. Instead, the interrupt mechanism is implemented using procedure call interface

```
wait.for.interrupt (VAL INT interrupt.number, INT time.fired)
```

This procedure call has the exact same semantics as the channel communication shown above: the process calling `wait.for.interrupt` sleeps until the interrupt fires. When it has fired, the process is resumed and receives the time the interrupt fired in the pass-py-reference parameter `time.fired`. If the interrupt fired before a call to `wait.for.interrupt`, `wait.for.interrupt` will return immediately supplying the time the interrupt fired.

2.3. Interrupts and Low Power

Traditionally, interrupt handlers are supposed to be very short, simple programs. When a hardware interrupt fires, state is saved so that the processor can return to its current point of execution, a pointer is looked up in the appropriate register, and the processor jumps to the interrupt handling routine. Embedded systems developers are taught to handle the interrupt as quickly as possible, perhaps by reading a value and storing it in a global variable. Control is then returned to the central control loop.

Lifting interrupts into the virtual machine has a cost. At the least, the firing of the interrupt must be acknowledged[7]. If a process is waiting on the interrupt, the workspace of the waiting process is updated, and then an interrupt is raised within the virtual machine. This allows the cooperative scheduler to then begin executing occam-π code after the call to `wait.for.interrupt`.

Ideally, a "concurrency aware" runtime should put the processor into a low power state when all of the processes it is executing are waiting for an internal communication, a timer event, or an external interrupt. To sleep the ATmega family of processors, one simply issues a single assembly instruction: `sleep`. For testing, this functionality was tested within the runtime as part of a branch[8].

2.3.1. Polling for Interrupts

Table 2. Average polling latencies for occam-π code (N=18) and standard deviation.

	occam (σ)
Poll w/o powersave	0.2267 ms (0.0429 ms)
Poll w/ powersave	0.2212 ms (0.0421 ms)

First, to ascertain that the changes made to the virtual machine did not impact its execution of code in the general case, we wrote a program that polled continuously for interrupts without any of the rescheduling that is common in most occam-π programs. This meant that the scheduler would never have a chance to execute, and the performance of a firmware with or without powersaving enabled should run the same. This is the case shown in Table 2: neither runtime performs significantly better when the userspace program is continuously polling.

[7] `http://projects.cs.kent.ac.uk/projects/kroc/trac/browser/kroc/trunk/tvm/arduino/interrupts.c?rev=7130`

[8] `http://projects.cs.kent.ac.uk/projects/kroc/trac/browser/kroc/branches/avr-sleep`

2.3.2. Interrupt Latency

Second, we compared the time it takes for the an occam-π program to respond to interrupts as opposed to a program written in C. It is already known from prior work that sequential bytecode executes 100x to 1000x slower than native code. To measure interrupt latency, one Arduino was used to generate digital events that triggered the external interrupts of a second Arduino. Then, a logic analyzer[9] measured the time that it takes for the second Arduino to wake and toggle pin 13 (the build-in LED). Table 3 shows that the handling of interrupts in occam-π is roughly 100x slower than a C program that does the same thing.

Table 3. Average interrupt handling latencies for occam-π and C code (N=18) and standard deviation.

	occam (σ)	C (σ)
Interrupt w/o powersave	0.1425 ms (0.0036 ms)	0.0013 ms (0.0000 ms)
Interrupt w/ powersave	2.4485 ms (0.0210 ms)	2.3339 ms (0.0233 ms)

Of interest is the second row of Table 3. This shows how long it takes to handle an interrupt when the processor is placed into a power-saving sleep mode. As can be seen, it does not matter whether the interrupt handling code is written in C or occam-π; it takes more than 2ms to wake from sleep and begin executing code. The occam-π code is approximately 0.11ms slower than the C, which is (again) in keeping with our previous measurements and the Transterpreter's known performance on interpreting sequential code.

The occam-π code does differ from the C code in one critical way: the virtual machine will not preempt a running process when an interrupt occurs due to the cooperative scheduling used by occam-π. The virtual machine must instead wait until it reaches a safe point (a rescheduling point) in the code before it can deschedule the current process and reschedule another. This could, in theory, mean that there could be no upper bound on the interrupt latency in an occam-π program. In practice this is not often an issue, and when it is, the compiler has an option for emitting more reschedule points, and it is possible to manually insert reschedule points in the code.

2.3.3. Interrupts: Practical Implications

The relatively poor performance of interrupt handling in the Transterpreter has a practical implication for programmers using our tools: we are limited as to how much information we can process using interrupts. For example, a serial communications handler written in occam-π will not be able to process characters at a baud rate of much more than 300bps. (While higher rates might be possible, it is unlikely much additional work could be done in-between the receipt of individual characters.)

That said, not all interrupt-driven applications are high performance or require microsecond response times. It is often the case that we need to respond to an interrupt in a sub-millisecond (but not sub-*micro*second) timeframe. In these cases, where the interrupt represents a clock tick or a sensor crossing a threshold, we can respond in more than adequate time while simultaneously helping the programmer deal with the traditional complexity of interrupt-driven programming. In the next section we discuss an environmental sensor that falls exactly into this category.

3. Case Study: A Room Usage Monitor

Being able to handle interrupt-driven sources in a simple and reliable manner is motivated by real-world need. Many sensors and devices that might be used with an Arduino change state

[9] `http://www.saleae.com/logic/`

Figure 3. Sensor exterior with motion and light sensors noted.

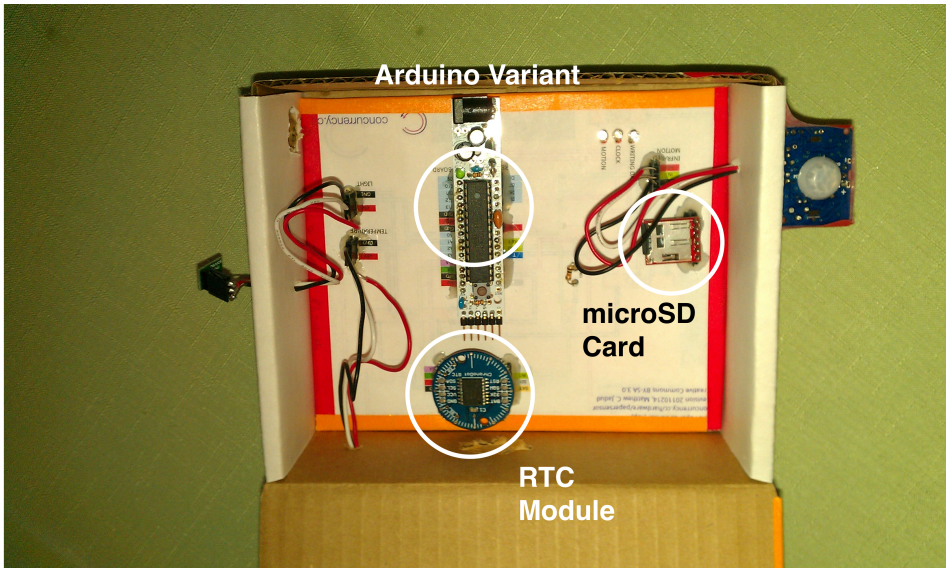

Figure 4. Sensor interior with Arduino, microSD card, and RTC noted.

in response to events in the world. For an Arduino to detect that change of state, one must either busywait or we can wait for an interrupt—which means we allow other processes to execute while waiting for sensor input from the world.

As an example of a recent, real-world use of interrupts on the Arduino, we share a case regarding the recent design and development of a sensor that was built and deployed by undergraduates enrolled in *ES210: Research Methods* at Allegheny College as part of their studies in Environmental Science (Figures 3, 4). The students wanted to determine what kind of energy waste was taking place in classrooms on campus, and the sensors would help them determine when (1) the lights were on and (2) there was no one in the room. We were given two weeks to research the components, prototype the sensor, and develop kits that the students could assemble as part of their laboratory sessions.

3.1. Sensor Design and Implementation

The room usage sensor had to be able to detect the light level in a room as well as detect when the room was occupied. Commercial off-the-shelf solutions in this space cost at least $100 to $150, and were all closed or "opaque" solutions, meaning that the students would have little say regarding how the device should function. For the sensors to be useful to the students, our design needed to satisfy a number of hardware and software requirements.

Measure Light. Record ambient light level, ideally able to distinguish between "daylight + lights" vs. "just daylight."

Fixed-cycle Measurements. Record light levels on a fixed interval (eg. every minute).

Movement-based Measurements. Record light levels when the room is occupied, throttling measurements (eg. no more than one motion-based event every 2 minutes).

Accurate Timing. All measurements should be stamped with an accurate timestamp.

Easy Assembly. ES students can be assumed to have no prior electronics background of any sort, yet they must be able to assemble/solder the entire sensor themselves.

Minimize Budget. Build sensors for $50 each or less..

Maximize Reusability. Sensors need to be re-usable on a component-by-component basis for future use in classroom contexts.

Easily Analyzed Data. Students need to easily be able to extract and analyze data using commonly available tools (eg. Open Office or Google Spreadsheets).

Our final bill of materials for each sensor came to approximately $75 per sensor. Each node is capable of the accurate measurement of temperature and light (the latter on a logarithmic scale in the same range as the human eye), has an extremely accurate real-time clock (or RTC, ± 2 seconds/year), can detect motion using a common passive infra-red motion sensing module, and uses a FAT-formatted, microSD card for data storage, allowing students to easily extract data from their sensor nodes at the end of the experiment. All major components are modular: the sensing components, RTC, microSD, and microcontroller are all easily removed from the node and incorporated in other designs, meaning that the total "lost" or "sunk" cost per node is under $5.

3.2. Sensor Control

The sensor was developed using the Plumbing library for occam-π on the Arduino[10]. There are two interrupts from the outside world: the RTC (which triggers an interrupt once every minute) and the IR motion sensor (which can trigger an interrupt once every five seconds). There are two analog sensors (temperature and light intensity), and one device attached to the serial output line (the microSD logger).

Whereas a solution written in C++ would likely need to leverage some kind of global state to pass sensor data from an interrupt routine (perhaps triggered by the motion sensor) into a control loop, our firmware has one process in the network that listens to each interrupt that we might receive from the outside. Both `real.clock` and `motion` use `digital.input` (defined in the Plumbing library) to wait on interrupts from the RTC and passive IR sensor. These processes then signal other processes that serve to throttle the rate at which either clock-based or motion-based events are logged (`n.minute.ticker`). When enough minutes have gone by to trigger a clock-based reading, or we have waited enough minutes to register another motion event, then a `SIGNAL` is generated to the `get.type` process.

[10]Complete source code for the sensor can be found online on GitHub: `https://github.com/jadudm/Paper-ES-Sensor/tree/57b3e14e6922a84d2d6f4a8d3c1034528ea5fcb5`.

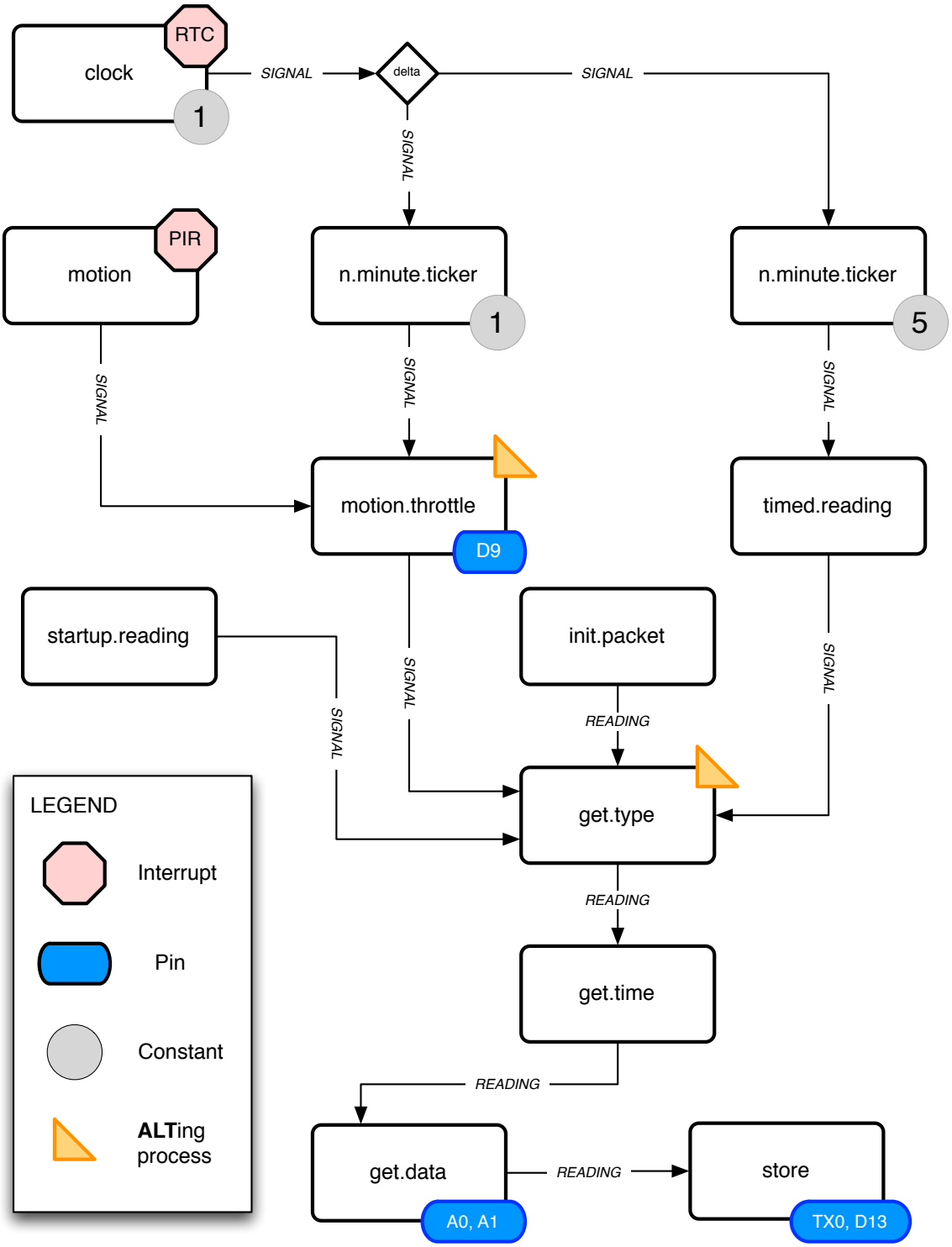

Figure 5. A process network for monitoring room usage.

The `get.type` process sits in our core pipeline, holding a largely uninitialized `READING` record. It is the only process in the core that contains an `ALT`, meaning we have isolated the non-determinism/potential randomness in our core data gathering pipeline into one location only. `get.type` watches three input channels, and depending on which fires, it populates the `READING` record with a flag indicating whether this was an initial reading taken at sensor startup (which can only happen once), a reading triggered by the clock, or a reading triggered by motion.

Once we have tagged the record, it is populated with the current time (which we request from our RTC), the current temperature and light level (which is an instantaneous reading taken from the temperature and light sensors), and finally this data is serialized as plain text out to the microSD logging module.

3.3. Development Process

The sensor control code was developed incrementally over a period of six days, from January 1, 2011 through January 6, 2011. At the start of the process, we had no experience with the RTC, the microSD (which was not part of the initial design), or the sensor platform itself. At the time that software development commenced, the hardware had not yet been fully designed: the circuit existed only as a prototyped on a breadboard. Our choice of occam-π as a language for embedded development made it possible to be co-developing hardware and software simultaneously without concerns that the fundamental architecture of our control software might be unsound, or (for that matter) that we might introduce critical bugs along the way.

The first commit[11] explored only the real time clock:

```
1  PROC main ()
2    SEQ
3      serial.setup(TX0, 57600)
4      zero.clock(FALSE)
5
6      CHAN SIGNAL s:
7      CHAN [3]INT time:
8      CHAN INT light:
9      PAR
10       current.time (s!, time!)
11       adc (A0, VCC, s?, light!)
12       display (time?, light?)
13 :
```

Listing 4. Blinking four LEDs in parallel at different rates.

This three process network (`current.time`, `adc`, and `display`) only aspired to transmit the current time and light level back to the developer over a serial link. By the end of the first day of development, explorations were underway to store data to a 256KB EEPROM[12]. This low-cost integrated circuit was originally intended as the destination for the sensor's data; as was discovered in testing, it would prove to be difficult for the student researchers to easily extract their data (violating a design goal for the project), which is a fundamental step in their research.

The utility of a process-oriented decomposition came when we replaced our data storage medium—originally an EEPROM, later a microSD card—and we were able to easily avoid fundamental changes in the process network. A single process called `store` accepted

[11] `https://github.com/jadudm/Paper-ES-Sensor/blob/f0fa1603ce/chronodot-and-ambi.occ`
[12] `https://github.com/jadudm/Paper-ES-Sensor/blob/099cec7610/firmware.occ`

a READING structure and serialized it out to the EEPROM. When we switched to a microSD card (which makes it possible for students to easily read and process the data they have collected), we instead a different physical protocol to access the storage medium, but our process network did not change. Instead, we inserted a new process that read from a channel of type READING and handled the low-level details correctly.

This modular, type-safe approach to embedded programming allowed us to develop a useful system leveraging the Plumbing libraries quickly. Specifically, we found that the abstractions that have been developed with the intention of making complex tasks simple (like waiting patiently for an interrupt from the outside world) work incredibly well. After several weeks of deployment on multiple sensors, our efforts paid off: the students were able to determine that classrooms sat idle as much as 47% of the time (with lights on), which across campus amounts to a substantial energy loss. This information will be used to inform decisions about future renovations on campus (regarding building automation), and the project itself sets the stage for future collaborations at the intersection of computing and environmental science within the institution.

4. Conclusion and Future Work

We see two important lines along with our future work should proceed: that which focuses on the community, and that which continues to explore fundamental issues regarding the implementation of concurrent and parallel programming languages in embedded contexts.

4.1. Growing and Supporting Community

Our porting of occam-π to the Arduino platform is an extension of previous work involving the successful use of the Transterpreter on a variety of small robotics platforms in educational contexts. What differentiates this work from previous efforts is our improved handling and abstraction over fundamentally complex aspects of hardware/software interaction and the large number of enthusiastic users in the Arduino community. In order to introduce this community to occam-π, we have begun work on a small, Creative Commons licensed book titled *Plumbing for the Arduino*. This book introduces the occam-π programming language and Plumbing libraries through a series of exercises grounded fully on the Arduino platform. Like all open projects, the book is a work in progress, but has been successfully used by members of the community who (1) have little programming experience or (2) no occam-π programming experience to become productive explorers and, in some cases, contributors to our ongoing efforts.

We consistently use process diagrams in the book to introduce Plumbing and the architecture of the occam-π programs. These diagrams translate, in a straightforward fashion, into occam-π code. Given this correspondence between diagrams and code there have been several attempts to create visual programming environments for occam-π. In [15], we describe a number of these efforts and our own ongoing efforts towards developing an interactive visual programming interface for a dataflow language. Currently, we are working on integrating the Plumbing library into our visual tool with the hope that we might introduce many of the concepts of dataflow programming without having to wrestle with the syntax of occam-π (or any other language for that matter). We will be investigating whether we can create a large and diverse enough set of components that occam-π can be easily used to generate sensors, like the one presented in this paper, for sensor prototyping applications. This would ultimately enable scientists to rapidly prototype low-cost sensors based on the Arduino platform, and program them visually using the occam-π language.

4.2. Implementing Concurrency

It is important to note that the Plumbing libraries do not guarantee the programmer protection from race hazards at a low level. For example, it is possible for two processes to claim that they are responsible for setting the hardware state of pin 13; one process might try to turn it off while another might try to turn it on—a classic race. Our runtime currently does not provide a mechanism for tracking these kinds of resources, but it could (at the expense of some of the already limited RAM resources on the Arduino).

In the same spirit, we cannot currently protect the user from attempting to wait in multiple places on the same interrupt. On one hand, this might be useful: if a single pin goes high, we could want multiple (different) processes to wake up and begin executing. That said, there are interrupts for which this could be bad: if two processes were to attach to the serial receive interrupt, we would (again) have a race, where one process might get the first character, than the other process the second... or, one might starve the other. In the case of the former example, we could simply use our existing implementation of `wait.for.interrupt` and enable the use of `BARRIER`s on the Arduino—the process waiting for the interrupt could enroll on the `BARRIER`, and any processes wanting to synchronize on that event would also enroll on the same barrier. It is the second case for which we need protection, however: we do not generally want multiple processes to be able to respond to the same interrupt.

Executing a bytecode interpreter on a processor executing at 16 MHz is, sometimes, a challenge. For example, we cannot write a pure-occam-π implementation of a serial receiver, as large programs with lots of parallel processes introduce long delays between opportunities for the serial handler to execute. Either we run occam-π on an Arduino with a faster processor (which does not exist), or we might look to other ways to speed up our runtime. There have been, in the past, explorations that seek to transform occam-π into either native C programs directly from the bytecode [16], generate C from a new compiler [17], or leverage existing frameworks like the Low Level Virtual Machine (LLVM) project [18,19]. The first would require a great deal more work—and would be unique to our toolchain. The second requires (at the least) updates to the virtual machine's scheduler API. The third would require developing an entire backend for LLVM targeting the megaAVR series of processors. While improved performance would be *nice*, it has not yet become critical, and therefore we acknowledge the potential need, but have not yet run into a situation where the Transterpreter on the Arduino has been completely inadequate.

5. Acknowledgements

This work was supported in part by the Department of Computer Science and the Department of Environmental Science at Allegheny College, as well as a grant from the Institute for Personal Robotics (`http://roboteducation.org/`). We wish to especially thank those students in the Spring 2011 offering of *ES210: Research Methods* at Allegheny College for their efforts and willingness to explore across disciplinary boundaries.

References

[1] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[2] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[3] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10:563–579, August 2005.

[4] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Ian Armstrong, Matthew Jadud, Michael Pirrone-Brusse, and Anthong Smith. The Flying Gator: Towards Aerial Robotics in occam-π. In Peter Welch, Adam Sampson, Fred Barnes, Jan Pedersen, Jan Broenink, and Jon Kerridge, editors, *Communicating Process Architectures 2011*, volume 68 of *Concurrent Systems Engineering Series*, pages 329–340, Amsterdam, June 2011. IOS Press.

[6] Massimo Banzi, David Cuartielles, Tom Igoe, and Gianluca Martinoand David Mellis. The Arduino. `http://www.arduino.cc/`, February 2011.

[7] Ben Fry and Casey Reas. Processing. `http://processing.org/`.

[8] Hernando Barragán. Wiring. `http://wiring.org.co/`.

[9] Tom 'spot' Callaway. How to tell if a FLOSS project is doomed to FAIL. `https://www.theopensourceway.org/wiki/How_to_tell_if_a_FLOSS_project_is_doomed_to_FAIL`, 2009.

[10] Jono Bacon. *The Art of Community. Building the New Age of Participation.* O'Reilly Media, 2009.

[11] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 99–106, Amsterdam, September 2004. IOS Press.

[12] Jonathan Simpson, Christian Jacobsen, and Matthew C. Jadud. A Native Transterpreter for the LEGO Mindstorms RCX. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 339–348, Amsterdam, July 2007. IOS Press.

[13] Matthew Jadud, Christian L. Jacobsen, Jon Simpson, and Carl G. Ritson. Safe parallelism for behavioral control. In *2008 IEEE Conference on Technologies for Practical Robot Applications*, pages 137–142. IEEE, November 2008.

[14] Matthew Jadud, Christian Jacobsen, and Adam Sampson. Plumbing for the arduino. `http://concurrency.cc/book/`.

[15] Jonathan Simpson and Christian L. Jacobsen. Visual process-oriented programming for robotics. In *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, pages 365–380, Amsterdam, September 2008. IOS Press.

[16] Christian L. Jacobsen, Damian J. Dimmich, and Matthew C. Jadud. Native Code Generation Using the Transterpreter. In P. Welch, J. Kerridge, and F. Barnes, editors, *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering*, pages 269–280, Amsterdam, September 2006. IOS Press.

[17] Adam T. Sampson and Neil C. C. Brown. Tock: One year on, September 2008. Fringe presentation at Communicating Process Architectures 2008.

[18] Carl G. Ritson. Translating etc to llvm assembly. In *CPA*, pages 145–158, 2009.

[19] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, pages 75–88, Washington, DC, USA, 2004. IEEE Computer Society.