On the Design and Implementation of a Virtual Machine for Arduino

Gonzalo Zabala, Ricardo Moran, Matías Teragni and Sebastián Blanco

Abstract Arduino has become one of the most popular platforms for building electronic projects, especially among novices. In the last years countless tools, environments, and programming languages have been developed to support Arduino. One of these is Physical Etoys, a visual programming platform for robots developed by the authors. Physical Etoys supports compiling programs into the arduino. For this to work, a Smalltalk to C++ translator has been built. Although it has been very useful, this translator has brought a new set of issues. In this paper we will discuss some of these problems and how we decided to overcome them by developing a simple virtual machine that will be used as the base for the new Physical Etoys.

Keywords Arduino · Programming language · Virtual machine · Concurrency · Physical etoys

1 Introduction

Since the emergence of the Arduino board, the world has seen a significant increase in the amount of people without technical training (artists, designers, hobbyists) that have started to explore the world of microcontroller programming. The educational field has not been exempt of this trend. Following the movement that promotes

G. Zabala (✉) · R. Moran · M. Teragni · S. Blanco
Universidad Abierta Interamericana, Buenos Aires, Argentina
e-mail: gonzalo.zabala@uai.edu.ar

R. Moran
e-mail: ricardo.moran@uai.edu.ar

M. Teragni
e-mail: matias.teragni@uai.edu.ar

S. Blanco
e-mail: sebastian.blanco@uai.edu.ar

M. Merdan et al. (eds.), *Robotics in Education*, Advances in Intelligent Systems and Computing 457, DOI 10.1007/978-3-319-42975-5_19

teaching programming and computer science in schools, robotics offers a highly motivational medium to introduce concepts from several disciplines. Furthermore, the amount of knowledge necessary to carry out small to medium size robotic projects is increasingly smaller. The Arduino board, being open hardware and low cost, invites students from all over the world to embark on the adventure of educational robotics.

The Arduino platform provides a simplified environment (based on the C++ programming language) in which most of the advanced microcontroller concepts are hidden away from the user. However, this environment is still too complex for some of the most inexperienced users, especially young children.

One of the main issues we found while teaching robotics to high school students is the limited support for concurrency in the Arduino programming language. Robotics competitions and exercises usually require the coordination of several concurrent tasks to achieve a goal. For instance, a common competition is the robot-sumo, in which two robots try to push each other out of a circle. This competition involves performing two simultaneous tasks: finding the opponent, and avoiding being pushed out of the circle. Other competitions involve more complex tasks. Most of the students participating in this kind of activities struggle to specify the concurrent behavior and as a result their robots perform poorly. The experience, thus, becomes frustrating instead of fun and engaging.

For these reasons, and taking advantage of the fact that Arduino is open source, there have been multiple attempts to provide a programming environment more suitable for beginners. One of these attempts is Physical Etoys (PE), an extension of Etoys [1] designed to provide children the tools to easily program different hardware platforms, including Arduino. Etoys is a media-rich authoring environment and visual programming system that allows children of all ages to create simulations and small videogames using a tile-based scripting system. Up to the creation of PE, Etoys only allowed to manipulate virtual objects, such as drawings on the screen. By using PE, a child can interact as easily with the virtual world provided by Etoys as with the real world through any of the supported robotic kits.

Etoys (and by extension, PE) provides a very simple concurrency model in which all of the user scripts run at configurable intervals inside an infinite loop. Although execution is entirely single-threaded, from the user perspective all the scripts are running simultaneously. This simple model is usually good enough for most students.

However, PE required the Arduino board to be constantly connected to the computer, which is not allowed in some robotics competitions. In order to address this issue PE can now also compile its scripts and upload them to the Arduino using a special programming mode. This feature involves a great deal of complexity that beginners are not usually able to handle. And since the PE firmware cannot coexist with the compiled programs, once the scripts are uploaded to the Arduino all the benefits provided by PE interactive environment are lost.

Moreover, this “compiled” mode has severe technical issues. In order for the compilation to work, PE first translates the scripts (which are automatically generated Smalltalk code) to C++, then it uses avr-gcc to compile the C++ sources

into machine code, and finally it uploads the machine code to the Arduino board using avrdude. This process is highly complex and slow. It forces the PE distribution to include the AVR tools, increasing its size up to 5 times (from 47 to 231 Mb in its last version). Furthermore, the AVR tools are platform dependent, which makes it difficult to provide a single cross-platform distribution for PE.

Even though the ability to choose the programming mode depending on the kind of project being carried out has distinguished PE from other similar projects (such as Scratch for Arduino [2] or Minibloq [3]) the above-mentioned problems require a different approach.

The following requirements have to be met:

1. The script execution must be performed directly on the Arduino board without the need for interaction with the computer.
2. If the arduino happens to be connected to the computer, all the interactivity features provided by PE must be preserved.
3. The user should not be required to specify which programming mode to use (either compiled or interactive).

With these objectives in mind, it was decided to implement a virtual machine that could interpret the bytecode of a very simple programming language. This virtual machine (which was called Uzi) would be uploaded to the Arduino board once. The computer can then send individual instructions or entire programs for the Arduino to run by communicating with the virtual machine through the USB port.

In this paper we will discuss the design and implementation of the Uzi virtual machine, and we will compare it with other similar technologies in order to highlight the benefits and limitations of this solution.

2 Related Work

The development of virtual machines and high-level languages for small microcontrollers is not new. There have been a lot of attempts to provide a different programming environment for Arduino. Most of them are based on pre-existent general purpose programming languages such as Java, Scheme or Python.

HaikuVM is one of such attempts [4]. It is a Java VM based on leJOS [5] that runs on Arduino. Its compiler analyzes the Java source code in order to generate a C program that contains the user program (stored in Flash memory as a set of C structs) and the virtual machine that will interpret it. The user must then use the Arduino toolset to upload the program to the board. This implementation has benefits, such as the low memory usage by storing the user program in the Flash memory alongside the VM, but it needs the arduino tools to compile and upload the programs. The fact that it outputs C code allows the compiler to easily introduce special constructs that let the user inline C code, thus allowing him to choose the level of abstraction required for the problem at hand. HaikuVM supports almost all of Java semantics, including garbage collection, threads, and exceptions, but it lacks

support for reflection, object finalization, weak references, and type information for arrays. In order to efficiently use the available memory in Arduino, the compiler performs a static program analysis that allows it to discard unused classes and thus generate more compact programs. Regarding performance, some benchmarks show an execution speed of "about 55k java opcodes per second on 8 MHz AVR ~ATmega8" [4].

Ocamm-pi is a variant of the Ocamm programming language [6] that supports several platforms, including Arduino [7]. Ocamm is especially designed to write concurrent programs, which are difficult to express using the Arduino language. It requires a board with at least 32 KB of space for code and 2 KB of RAM, so the smallest Arduino boards supported are the ones that use the ATmega328 chip. Similarly to HaikuVM, the occam-pi bytecodes are stored in flash memory alongside the virtual machine. However, unlike HaikuVM, the bytecode can be uploaded separately. Another similarity ocamm-pi has with HaikuVM is the static program analysis that allows it to eliminate dead-code and generate compact programs. This process is not only performed on user-generated code but also on ocamm-pi libraries. Ocamm-pi has a rich set of runtime libraries that provide functions for interacting with Arduino features such as the serial port, PWM and TWI. Most of these libraries are entirely implemented in occam-pi. This is possible thanks to interrupts and memory being accessible from occam-pi code, allowing the development of low-level libraries directly in occam-pi. However, handling interrupts using occam-pi code has a performance cost that limits the amount of information that can be processed. For example, handling serial communication in occam-pi can only process characters at a baud rate of at most 300 bps. Regarding performance, the execution of bytecodes has been reported be 100–1000 times slower than the execution of native code.

Splish [8] is an interesting project because instead of providing only a virtual machine it also provides a visual programming environment, much like PE. All the instructions and programming constructs are represented as icons that can be interconnected to define the program flow. The programs built using this visual environment are then compiled into an object code for a stack virtual machine designed specifically for this language. Uploading the compiled programs into the Arduino board is done via USB. The Splish firmware includes a monitor program that is in charge of the communication between the board and the computer; it listens to the Serial port for commands to execute and periodically sends back status information. This allows the computer to monitor the state of the Arduino pins and the execution of the programs. It is designed with debugging facilities in mind, even if that has a negative impact on the performance. If the Arduino board is connected to the PC, it can run programs in "debug mode", allowing step by step execution.

PyMite [9] (also known as python-on-a-chip) is a Python interpreter for 8-bit and larger microcontrollers. It can execute a subset of Python bytecodes and it supports almost all of Python's most important data types (such as 32-bit signed integers, Strings, Tuples, Lists, and Dictionaries) and some advanced features such as generators, classes, and decorators. It allows writing native code by marking a Python function with a special keyword and writing the C code in the function's

documentation string, thus making it easy to develop low level libraries. It supports several platforms, but since it requires at least 64 KB of program memory and 4 KB of RAM, Arduino boards smaller than the MEGA are not supported.

The Scheme programming language has several implementations designed for small microcontroller based embedded systems. Two of the most interesting ones are Microscheme [10] and PICOBIT [11], which are very different in their approach, even though they both are implementations of the same programming language. Microscheme targets the 8-bit ATmega chips used by most Arduino boards, while PICOBIT targets the Microchip PIC18 family of microcontrollers. Microscheme differs from PICOBIT in that it uses direct compilation instead of a virtual machine. Its compiler, written in C, generates AVR assembly code which is in turn assembled and uploaded to the board using the avr-gcc/avrdude toolchain. PICOBIT instead provides a Scheme virtual machine written in portable C that, although being currently implemented for the PIC18 microcontrollers, could be ported to any platform that has a C compiler. Another characteristic worth mentioning of the PICOBIT approach is that it does not only provide a custom Scheme compiler and virtual machine but also a custom C compiler designed specifically for developing virtual machines. This C compiler takes advantage of the patterns commonly found in the implementation of virtual machines and it performs a set of optimizations that result in a significant reduction of the generated code. Both implementations support different subsets of Scheme.

3 Design Principles

The main goal of this project is to provide a tool that a visual programming environment such as PE could use to compile and run its programs.

Given that PE has an educational purpose, Uzi was designed based on the following principles:

- Simplicity: It should be easy to reason about the virtual machine and how it does its job.
- Abstraction: It is the responsibility of the Uzi language to provide high-level functions that hide away some of the details regarding both beginner and advanced microcontroller concepts (such as timers, interruptions, concurrency, pin modes, and such). These concepts can later be introduced at a pace compatible with the needs of the user.
- Monitoring: It should be possible to monitor the state of the board while it is connected to the computer.
- Autonomy: The programs must be able to run without a computer connected to the board.
- Debugging: Uzi must provide mechanisms for error handling and step by step execution of the code. Without debugging tools, the process of fixing bugs can be frustrating for an inexperienced user.

4 Implementation

In order to simplify the translation of PE scripts to Uzi programs the execution model of the Uzi virtual machine was designed to be as similar as possible to the Etoys model. Like Etoys projects, an Uzi program can include several scripts that are executed concurrently. Each script runs forever in an implicit loop and its execution rate can be configured independently. This model simplifies the needed code to express concurrent tasks, as can be seen in the example section of the present paper.

Some of the Uzi tools reside on the computer while others run directly on the Arduino. The computer counts with all the necessary components to parse, compile, and transmit the programs to the Arduino board through the serial port. All these programs were developed using Squeak, an open source version of Smalltalk. The Arduino, on the other hand, includes all the software required to execute Uzi programs. These tools were written in C++ (Fig. 1).

The UziParser is responsible for parsing a string written in Uzi syntax and generating a parse tree. It was implemented using PetitParser [12], a parsing framework that allows you to define parsers using Smalltalk code. Although the Uzi syntax is heavily inspired by Smalltalk, it should not be confused with Smalltalk code. Uzi does not follow any of the Smalltalk semantics. It does not support objects nor late binding. It is a domain specific language which main purpose is to run PE scripts inside the Arduino board and it was designed to make the translation process as simple as possible.

An example script written in the Uzi language can be seen below. This small program blinks the LED on pin 13:

```
#blink13 ticking 1 / s [toggle: 13]
```

The UziCompiler is responsible for traversing the parse tree and generating a compiled program containing the bytecodes that the Uzi virtual machine will execute.

The UziEncoder is in charge of serializing the program to a custom binary format designed for Uzi. This format is designed to be as compact as possible

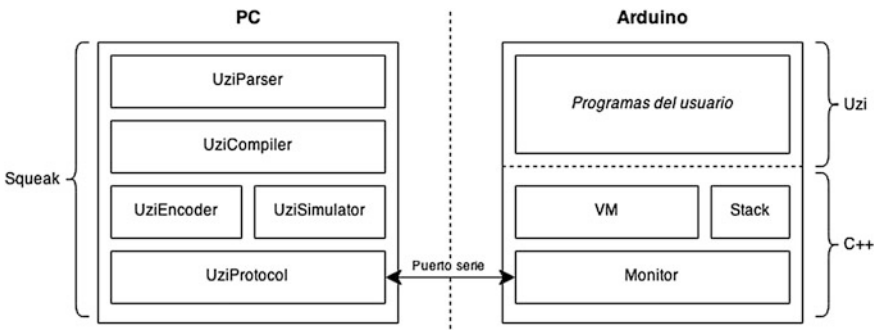

Fig. 1 Uzi architecture

because it will not only be used to transmit the program to the arduino but also to store it in the EEPROM, which has very limited space.

The UziSimulator is a Smalltalk implementation of the Uzi virtual machine. This tool allows us to run on the computer the exact same process that the Arduino will execute. This is currently useful to verify the implementation of new functionality before making the change in the actual virtual machine that will be uploaded to the board. In the future, the UziSimulator might also be used to add debugging features such as step by step execution.

The UziProtocol is the last tool on the PC side. It is used by the other components to communicate with the arduino. It can either send entire programs or specific commands that the arduino will execute. It also listens for arduino state updates. All the IO primitives implemented on the UziSimulator, for example, use the UziProtocol to actually perform the operation on the board.

On the arduino side, Uzi is installed as a firmware that contains both the Uzi virtual machine and also a small Monitor program that communicates with the UziProtocol through the serial port. The Monitor acts as a bridge between the virtual machine and the development tools. It listens on the serial port waiting for commands to execute and periodically sends data back to the computer. The commands that the Monitor understands include IO operations, executing a specific program, and storing a program in the EEPROM memory. The data that the Monitor sends includes the state of the pins and the state of the virtual machine (global variables, instruction pointer, stack, and current script).

The VM class is responsible for executing Uzi programs. It requires, essentially, two attributes: the instruction pointer (IP), an integer that refers to the next instruction to be executed; and a pointer to the stack. In each tick, the VM iterates over the entire list of scripts. For each script the VM knows the time it was last executed and its ticking rate. If the time since it was executed exceeds its ticking rate, the VM executes the script. Executing a script involves resetting the IP and executing each of the script's bytecodes one by one. The execution of a script must leave the stack exactly as it was before its execution started. The bytecode execution is handled by a simple switch statement. Since, as mentioned before, the Uzi compiler privileges small code size over execution speed, the Uzi instruction set was designed to use as little space as possible. Each instruction occupies one byte, where the most significant four bits are used to represent its operation code and the least significant 4 bits are used to specify its operand. Since 4 bits can only address a maximum of 16 values, a special instruction is used to extend a specific operation by using the next byte as its operand. The value 0xFF is used to mark the end of a script. Since only 4 bits are used to represent an operation code, the instruction set only includes the most common operations, such as handling the stack, accessing pins, calling primitives, and starting/stopping scripts. Other operations (such as arithmetic or logical) are implemented as primitives.

The stack has a fixed size of 100 elements. In case of stack overflow, the VM will stop execution immediately. The invalid state will be stored and transmitted by the Monitor to the host PC (if connected).

5 Example

The following example, although admittedly simple, is useful to show the differences between code written in the simplified C environment provided by Arduino (which we will call Arduino code), scripts built using the PE visual interface, and scripts written in the Uzi programming language.

This program performs four independent tasks:

1. It blinks a LED once per second (BLINK13).
2. It blinks another LED twice per second (BLINK12).
3. It turns on a third LED when a button is pressed (BUTTON).
4. It controls the brightness of a fourth LED with a potentiometer (DIMMER).

These simple tasks are performed concurrently, which is something difficult to express in the Arduino code. As it can be seen in the example below the Arduino code mixes the statements that perform the tasks with the code required to schedule them at the correct intervals. This makes the code's intention less obvious and, thus, harder to read and modify.

The Arduino conceptual model for the pins represents another issue. In order to read or write, it differentiates analog and digital pins, forcing the user to use different functions for different types of pins. The abstraction is not event correct: the function that "writes" a PWM wave is called analogWrite() even though it does not generate an analog wave and is not related to analog pins or the analogRead() function in any way [13]. Additionally, each pin can either be in one of two modes, which the user must explicitly specify: INPUT for reading, and OUTPUT for writing. A simpler model could restrict the operations that can be performed on a pin to simply "write" and "read", handling each specific case without exposing the details to the user. This model has its drawback but it would be simpler to understand for a beginner than the Arduino functions. Moreover, while digitalWrite () and digitalRead() functions work on the same range (either 0 or 1), analogWrite() accepts a value from 0 to 255 and analogRead() returns a value from 0 to 1023. This small difference forces the user to transform from one scale to the other when trying to use the input from an analog pin to output a PWM signal, as can be seen in the Arduino example code. Failing to do this can lead to incorrect behavior, which is difficult to debug for an inexperienced user.

Additionally, since you can't read the value of a pin configured as OUTPUT (without accessing the registers directly, at least), in order to blink the LEDs, the user is forced to store the state of the pins in a variable. This extra code adds complexity to the solution.

Handling each LED blink rate also requires extra code. Using the delay() function, which blocks the processor for a given amount of time, as it is a common practice in Arduino examples, is not allowed here because it would disrupt the execution of the other tasks (the Arduino board has only one microcontroller). Instead, the user is forced to call the millis() function and check on every tick if it is time to blink each led.

All these issues with the Arduino code greatly increases the complexity of an otherwise simple project.

```
boolean leds[] = { false, false };
unsigned long last = 0;

void setup() {
 for (int pin = 10; pin < 14; pin++)
   pinMode(pin, OUTPUT);
 pinMode(9, INPUT);
 pinMode(A1, INPUT);
}
 void loop() {
  unsigned long now = millis();
  if (now != last) {
    if (now % 1000 == 0) toggle(13); // BLINK13
    if (now % 500 == 0) toggle(12);  // BLINK12
    last = now;
  }
  digitalWrite(11, digitalRead(9)); // BUTTON
  analogWrite(10, analogRead(A1) / 4);  // DIMMER
 }

 void toggle(int pin) {
  leds[pin - 12] = !leds[pin - 12];
  digitalWrite(pin, leds[pin - 12] ? HIGH : LOW);
 }
```

In PE this same example is very different due to the fact that PE is a completely visual programming environment. First, the user needs to indicate which type of device is connected to each pin by clicking and dragging on icons. Then the user has to build each script by, again, clicking and dragging the different instructions. Each script belongs to an object and it runs concurrently with all the others. The concurrency is automatically handled by the PE scheduler, which simplifies describing the execution of concurrent tasks (Fig. 2). Although it cannot be seen in the figure, each script is configured to run at different rates: 1/s. for the first, 2/s for the second, and 100/s for the third and fourth scripts. Such configuration is much simpler to set up in PE than in the Arduino code. Each task is encapsulated into its own script, which simplifies reading and understanding the code. The visual interface presented by PE is easier for beginners to understand because it makes syntax errors impossible and it exposes the user to an object oriented API in which each graphical object represents a real object that the user can manipulate directly. Although the user does not have to specify each pin mode, it does have to tell PE which devices are connected to each pin, but doing it by clicking and dragging

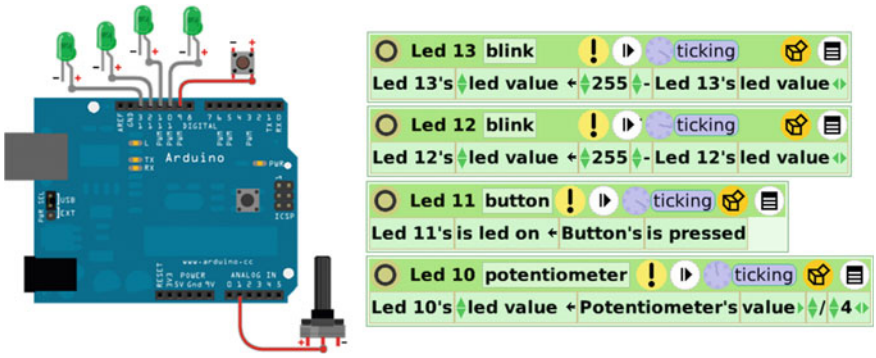

Fig. 2 Graphical representation of the arduino inside PE and its corresponding scripts

devices into their corresponding pins feels much more natural and intuitive than calling a function.

Finally, the Uzi program is the smallest of the three, with only four lines of code describing four scripts. Each script can be configured with its own ticking rate and the Uzi virtual machine will take care of executing it at the desired interval. If the user does not specify a ticking rate (as it is the case with the "button" and "dimmer" scripts, then the virtual machine will execute them on every tick). It is no longer necessary to remember the state of each pin in order to blink the LEDs, because Uzi handles it automatically when calling the "toggle:" primitive. There is no distinction between analog and digital pins, the only operation available is "write:value:" and "read:" (apart from others that can be built upon these two, such as "toggle:") and both accept values in the 0 to 1 range. This can be seen in lines 3 and 4, where the scripts have essentially the same statements but with different parameters. And finally, the user is not forced to specify each pin mode explicitly; Uzi configures the pins automatically.

```
#blink13 ticking 1 / s [toggle: D13]
#blink12 ticking 2 / s [toggle: D12]
#button ticking [write: D11 value: (read: D9)]
#dimmer ticking [write: D10 value: (read: A1)]
```

6 Limitations

Some of the design decisions that were taken during the implementation of Uzi resulted in limitations, performance being the most important. Using a virtual machine makes it nearly impossible to obtain the same performance that can be obtained using native code. Although no benchmarks have been run yet, we expect the performance to be at least 100x slower. For most of the programs we expect the users to write using PE this might not be a problem, but for others this might

impose a clear limitation. One of the solutions that is being considered is to automatically generate, apart from the Uzi bytecodes, a C++ program that could be uploaded to the board using the arduino software. This would allow the maximum efficiency for the cases when it is needed while maintaining the benefits of the virtual machine approach.

The small amount of memory available on Arduino boards poses a whole set of limitations. Currently, the EEPROM is being used exclusively for program storage, which means that the user cannot use it to store values from its programs. Ideally, the Monitor, VM and user programs would all be stored in Flash memory (which is much larger), but this is not implemented yet. Until then, no strings or arrays are supported because they occupy too much space.

The current Uzi implementation does not allow dynamic memory allocation. This design decision has several advantages, such as making the implementation of a garbage collector unnecessary or allowing static analysis to determine how much memory the program will need at compile time. However, it also restricts the type of programs that can be written using Uzi.

7 Future Work

Uzi is still a work in progress. Although most of its design is finished, only a small subset of all primitives is currently implemented, which allows to write only simple programs like the one described above. Once the implementation becomes stable, it will be integrated with PE so that visually scripted Etoys projects can be translated to Uzi bytecodes.

The Uzi language also requires better tooling. Although debugging is one of the project guiding principles, no debugger has been implemented yet. The development of an integrated development environment is planned for the future.

Even though Uzi is currently designed with a special focus on PE, it is of interest for the authors to evaluate its capabilities as an intermediate language in which different programming models could be implemented.

Finally, since the Uzi virtual machine is small and simple, porting the Uzi virtual machine to other educational robotics platforms (such as Lego Mindstorms Nxt or even PIC microcontrollers) is also of interest.

8 Conclusion

The design and implementation of Uzi, a virtual machine for Arduino, was described.

This virtual machine solves a specific problem encountered while using PE to teach robotics to high school students.

The advantages of Uzi over the traditional Arduino tools were exemplified by writing the same program in three different programming languages: the simplified C provided by Arduino, PE, and Uzi.

Given the advantages of Uzi over the traditional Arduino tools, its use for educational purposes is highly encouraged.

References

1. Freudenberg, B., Ohshima, Y., Wallace, S.: Etoys for one laptop per child. In: 7th International Conference on Creating, Connecting and Collaborating through Computing—C5 2009, Kyoto, pp. 57–64 (2009)
2. Citilab: Scratch for Arduino (2015). http://s4a.cat/
3. Rahul, R., Whitchurch, A., Rao, M.: An open source graphical robot programming environment in introductory programming curriculum for undergraduates. In: 2014 IEEE International Conference on MOOCs, Innovation and Technology in Education, IEEE MITE 2014, Patiala, pp. 96–100 (2014)
4. Bob Genom: HaikuVM: a small JAVA VM for microcontrollers (2014). http://haiku-vm.sourceforge.net/
5. Rao, A.: The application of LeJOS, Lego Mindstorms robotics, in an LMS environment to teach children Java programming and technology at an early age. In: 5th IEEE Integrated STEM Education Conference, ISEC 2015, pp. 121–122(2015)
6. Elizabeth, M., Hull, C.: Occam-a programming language for multiprocessor systems. Comput. Lang. **12**(1), 27–37 (1987)
7. Jacobsen, C.L., Jadud, M.C., Kilic, O., Sampson, A.T.: Concurrent event-driven programming in occam-π for the Arduino. Concurr. Syst. Eng. Ser. **68**, 177–193 (2011)
8. Kato, Y.: Splish: a visual programming environment for arduino to accelerate physical computing experiences. In: 8th International Conference on Creating, Connecting and Collaborating through Computing, C5 2010, La Jolla, CA, pp. 3–10 (2010)
9. Python (2014). https://wiki.python.org/moin/PyMite
10. Suchocki, R., Kalvala, S.: Microscheme: functional programming for the Arduino. In: Scheme and Functional Programming Workshop, Washington, D.C., pp. 21–29(2014)
11. St-Amour, V., Feeley, M.: PICOBIT: a compact scheme system for microcontrollers. In: 21st International Symposium on Implementation and Application of Functional Languages, IFL 2009, South Orange, NJ, pp. 1–17 (2010)
12. Bergel, A., et al.: PetitParser: Building Modular Parsers. In: Deep into Pharo, pp. 375–410 (2013)
13. Arduino—analogWrite() (2015). https://www.arduino.cc/en/Reference/AnalogWrite