A Concurrent Programming Language for Arduino and Educational Robotics

Ricardo Moran, Matías Teragni, Gonzalo Zabala

Centro de Altos Estudios en Tecnología Informática,
Facultad de Tecnología Informática,
Universidad Abierta Interamericana, Av. Montes de Oca 745, Ciudad Autónoma de Buenos Aires, República Argentina
(+54 11) 4301-5323; 4301-5240; 4301-5248
{Ricardo.Moran, Matias.Teragni, Gonzalo.Zabala}@uai.edu.ar

Abstract. Arduino is currently one of the most popular platforms for educational robotics due to its low cost and large amount of available resources online. The Arduino software library provides an abstraction layer over the hardware details, making it possible for novices to build interesting projects. However, its lack of support for concurrency makes some educational robotics projects difficult. In this paper, we explore different approaches to solve this problem and we propose the implementation of a concurrent programming language supported by a virtual machine running on the Arduino board.

Keywords: education, robotics, programming language, virtual machine, Arduino, concurrency

1 Introduction

Arduino has become one of the most popular platforms for building electronic projects, especially among hobbyists, artists, designers, and people just starting with electronics. The Arduino software library and integrated development environment (IDE) provide an abstraction layer over the hardware details that makes it possible to build interesting projects without a complete understanding of more advanced microcontroller concepts such as interrupts, ports, registers, timers, and such. At the same time, this abstraction layer can be bypassed to access advanced features if the user needs them. These characteristics make the Arduino platform suitable for both beginners and experts.

However, there is one aspect in which the Arduino language lacks proper abstractions: concurrency. For all but the simplest projects, the `setup()` and `loop()` program structure [1] proposed by Arduino is not expressive enough. Even moderately complex problems require some sort of simultaneous task execution.

Furthermore, most educational robotics projects require the implementation of a device that performs two or more simultaneous tasks [2]. This poses a limitation on the type of educational projects that can be carried out, especially if the teaching subject is not robotics or programming itself.

2 Solution

We propose the implementation of a concurrent programming language [3] supported by a virtual machine running on the Arduino. We call this language UziScript and we expect it to become a suitable compilation target for visual programming environments such as Physical Etoys [4], Scratch for Arduino [5], and Ardublock [6], among others.

Given that the main purpose of this programming language is educational, it was designed based on the following principles:

- Simplicity: It should be easy to reason about the virtual machine and understand how it performs its job.
- Abstraction: the language should 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 student.
- 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: the toolchain must provide mechanisms for error handling and step by step code execution. Without debugging tools, the process of fixing bugs can be frustrating for an inexperienced user.

3 Related work

Even though the Arduino libraries don't provide any concurrency abstraction by default, third-party libraries are available that attempt to address this issue.

SCoop [7] is a library that implements cooperative multitasking. It provides the user a set of objects, functions and macros that essentially allow to specify multiple `loop()` functions with independent intervals. Each task has its own stack to hold the values of its computations, and instead of blocking it uses the reserved keyword "`yield`" to let another task use the processor. Some utility functions that simplify task creation are provided, as well as a `sleep()` function that works as a replacement for the Arduino `delay()` function.

ArduinoProcessScheduler [8] is another library that provides similar functionality. Unlike SCoop, this library forces the object-oriented paradigm into the user's sketch. Each task must be defined in its own class, which must inherit from `Process`, and the task's behavior must be implemented by overriding the `service()` method.

Other libraries worth mentioning are ArduinoThread [9] and everytime [10]. These two do not implement truly independent tasks, instead they provide facilities to schedule the execution of procedures at desired intervals. Each time a task is executed it will run to completion. The main difference between the two is that everytime

implementation is fully based on compile-time macros, which makes it impossible to truly dynamically create new processes.

A lot of other libraries exist but even though these libraries are useful, they all exhibit the same flaw: they require the user to understand the execution model provided by the library to use it effectively. If the programmer fails to understand the strengths and limitations of the library, they may be punished with potentially hard to debug errors.

Forcing the users to write code in a particular style is difficult (if not impossible) with just a library. In contrast, a virtual machine approach has the advantage of providing an isolation layer between the guest and host systems. The developer of the virtual machine is responsible for deciding which native abstractions to expose to the programmer and it has entire control over the way in which these abstractions are presented.

We can find several implementations of high-level languages and virtual machines for the Arduino platform. Most of them are based on preexistent general-purpose programming languages such as Java [11], Scheme [12] or Python [13]. A more interesting example is the Transterpreter project [14], a virtual machine designed to exploit concurrency on embedded systems. This virtual machine runs occam-pi programs on several platforms, one of which is Arduino [15]. Occam-pi is a variant of the occam programming language [16], especially designed to write concurrent programs based on the communicating sequential processes (CSP) process algebra [17]. Occam-pi has a rich set of runtime libraries that provide functions for interacting with Arduino features such as the serial port, PWM and TWI. Regarding performance, the execution of bytecodes has been reported be 100 to 1000 times slower than the execution of native code.

4 Description of the language

UziScript syntax[1] is based on C, which is familiar to most programmers including Arduino developers. The "`task`" keyword has been added to represent behavior that can be executed periodically at a configurable rate. For example, the following code will declare a task that will toggle the LED on pin 13 every second.

```
task blink() running 1/s { toggle(D13); }
```

UziScript does not require any type declarations, so to distinguish a function from a procedure two new keywords are introduced: "`func`" and "`proc`".

```
func isOn(pin) { return read(pin) > 0.5; }

proc toggle(pin) {
  if isOn(pin) { turnOff(pin); }
  else { turnOn(pin); }
}
```

[1] For the complete grammar, see: https://richom.github.com/Uzi/cacic2017/uzi.pegjs

A program can have any number of tasks, and each task can be defined with a different interval as well as a different starting state, which can be either "`running`" or "`stopped`". Additionally, special tasks that need to be run just once can be defined using the keyword "`once`" after the task name. This is especially useful to initialize variables and can be used as a substitute to the Arduino `setup()` function. The execution of each task at the correct time is performed automatically by the virtual machine but the user can invoke certain primitives to start, stop, pause, or resume a given task. Each task execution is independent, it has its own stack, and it shares memory with other tasks through specially defined global variables. This design allows users to write sequential programs in Arduino's usual style and make them run concurrently without being concerned about the processor scheduling. Primitive instructions like `delay()` are provided to allow the user to block the executing task for a given amount of time without affecting the rest.

The "`import`" keyword allows to include external libraries, which can extend the primitive functionality provided by the language. For now, only a library to interact with DC motors is implemented. We are currently working on implementing libraries to provide process synchronization. Even though these are just experiments, we have working examples of recursive locks and synchronous channels[2].

5 Implementation

The UziScript firmware is implemented as a regular Arduino sketch that can be uploaded using the Arduino IDE. We initially target the Arduino UNO because it is the most popular model amongst beginners but we plan to support other models in the future. All the code (including the firmware, compilation tools, and libraries) is open source and it is available online[3].

Internally, the firmware implements a stack-based high-level language virtual machine that uses a decode and dispatch bytecode interpreter [18] to execute UziScript programs. This implementation was chosen mainly because of its simplicity. Since the purpose of this language is educational, performance is not currently considered a high priority.

The stack and global variables are the only memory available to the user program. There is no heap or dynamic memory allocation. This allows for simpler virtual machine code and very compact object code [19]. The instruction set is designed to be compact. Almost all the instructions can be encoded using one byte for both the opcode and its arguments and just a few special instructions (such as branches) require an extra byte[4].

Apart from the virtual machine, the firmware includes a monitor program that allows to interact with a computer through the serial port. Periodically, this monitor program

[2] These libraries can be found at: https://github.com/RichoM/Uzi/tree/master/uzi/libraries
[3] Source code at: https://github.com/RichoM/Uzi
[4] The instruction set specification can be found at: https://richom.github.io/Uzi/ISA

will send the status of the Arduino and receive commands, allowing the host computer to fully control the virtual machine.

By having these two programs running on the Arduino we can provide an interactive programming experience with a short feedback loop without sacrificing autonomy. Moreover, the monitor program permits the implementation of debugging tools that allow the user to stop the execution of any task, inspect the value of all the variables, explore the call stack, and execute instructions step by step.

6 Validation

We wrote three versions of the same program in order to compare the resulting code. The first version is written in standard Arduino code without using any concurrency library. The second version uses the SCoop library mentioned above. The third version is written in UziScript.

The chosen problem was inspired by the rules of the Robocup Junior Rescue competition [20]. However, the requirements have been slightly simplified. The objective of the challenge is to make a line follower robot [21] that can detect small patches of color representing "victims". The line is made with standard electrical (insulating) tape, 1 - 2 cm wide. The victims are placed at random places along the circuit and every time the robot detects a victim it must stop and play an alarm to notify its finding. The number of beeps must be equal to the total number of victims found so far.

This challenge was chosen because it is representative of the problems a student learning robotics might face and its solution requires at least two concurrent tasks: following the line and detecting the victims. While each task on its own is trivial to solve, performing them at the same time can be challenging for a novice.

The same robot will be used for all versions of the program. This robot will be composed of an Arduino UNO, two DC motors, a single line sensor on its front, and a buzzer. The motors will be placed in a way that allows the robot to move forward, backwards and rotate. The line sensor will be made from an LED and a photoresistor. While not optimal, this type of sensor is inexpensive and easy to build.

For reasons of brevity and clarity, only extracts of the actual code will be included in this paper (the full source code will be made available online).

Arduino version of the Robot Rescue line follower[5]:

```
int victims, lastVictimTime, state;

void loop() { followLine(); detectVictims(); }

void followLine() {
  switch (state) {
    case 0:
      moveRight(); if (isOnBlack()) state = 1;
```

[5] Full source code at: https://richom.github.io/Uzi/cacic2017/line_follower.ino

```
      break;
    case 1:
      moveRight(); if (isOnWhite()) state = 2;
      break;
    case 2:
      moveLeft(); if (isOnBlack()) state = 3;
      break;
    case 3:
      moveLeft(); if (isOnWhite()) state = 0;
      break;
  }
}

void detectVictims() {
  if (millis() - lastVictimTime < 100) return;
  if (isOnVictim()) {
    stopMotors(); victims++;
    for (int i = 0; i < victims; i++) {
      digitalWrite(buzzer, HIGH); delay(500);
      digitalWrite(buzzer, LOW); delay(500);
    }
    lastVictimTime = millis();
  }
}
```

Although the two tasks have been encapsulated into separate procedures ("`followLine`" and "`detectVictims`"), the execution of each procedure can interfere with the other. For this reason, the Arduino solution is implemented as a state machine. The `state` variable keeps track of the line follower state in order to know which direction to move at each step. Similarly, the `lastVictimTime` variable is needed to keep track of the last time in which a victim was found and avoid detecting the same victim more than once.

SCoop version of the Robot Rescue line follower[6]:

```
int victims;

void setup() {
  pinMode(sensor, INPUT); pinMode(buzzer, OUTPUT);
  mySCoop.start();
}

void loop() { yield(); }

defineTaskLoop(followLine) {
  while (!isOnBlack()) { moveRight(); yield(); }
```

[6] Full source code at: https://richom.github.io/Uzi/cacic2017/line_follower.SCoop.ino

```
  while (!isOnWhite()) { moveRight(); yield(); }
  while (!isOnBlack()) { moveLeft(); yield(); }
  while (!isOnWhite()) { moveLeft(); yield(); }
}

defineTaskLoop(detectVictims) {
  if (isOnVictim()) {
    followLine.pause();
    leftMotor.stop(); rightMotor.stop();
    victims++;
    for (int i = 0; i < victims; i++) {
      digitalWrite(buzzer, HIGH); sleep(500);
      digitalWrite(buzzer, LOW); sleep(500);
    }
    followLine.resume();
    sleep(100);
  }
}
```

By allowing the user to express concurrency using the `defineTaskLoop` macro, SCoop encourages a clearer separation of responsibilities and simpler code. The `followLine` task, in contrast to the Arduino version, can now be expressed as a straightforward sequential procedure that can be paused and resumed as needed. Avoiding multiple detections of the same victims can be achieved using a simple `sleep(100)` instruction after resuming the `followLine` task. Since the `sleep()` function only blocks the currently executing task, the follow line procedure will keep running while the victim detection does not execute for 100 ms.
Even though the execution of each task is independent it is still possible for a task to affect another (as demonstrated by the `detectVictims` task) but this interaction must be declared explicitly while in the Arduino code any blocking operation might inadvertently break the behavior of another procedure.
This makes the SCoop solution more robust and scalable as the number of concurrent tasks increases. However, the abstraction presented by SCoop is leaked as soon as the user is forced to call the `yield` function on every loop. Failing to meet this requirement can result in incorrect behavior that could be hard to reproduce, hard to diagnose if the user does not understand the cooperative multitasking model used by SCoop, and hard to fix considering the limited support for debugging in the Arduino IDE.

UziScript version of the Robot Rescue line follower[7]:

```
var victims = 0;

task followLine() running {
  until isOnBlack() { moveRight(); }
  until isOnWhite() { moveRight(); }
  until isOnBlack() { moveLeft(); }
```

[7] Full source code at: https://richom.github.io/Uzi/cacic2017/line_follower.uzi

```
  until isOnWhite() { moveLeft(); }
}

task detectVictims() running {
  until isOnVictim();
  pause followLine; stopMotors();
  victims = victims + 1;
  repeat victims {
    turnOn(buzzer); delay(500);
    turnOff(buzzer); delay(500);
  }
  start followLine; delay(100);
}
```

UziScript version shares the same benefits as the SCoop version but it doesn't suffer from its weaknesses. By letting the virtual machine handle the scheduling without any involvement from the user, UziScript solution is both more robust and easier to use by a novice. Furthermore, the use of a virtual machine with debugging capabilities makes it easier for the user to diagnose and fix any incorrect behavior due to a concurrency bug.

While simple, this example shows how the Arduino code is more fragile and can introduce complexity as the program grows. Adding an extra concurrent task without disturbing the current behavior could be challenging for a novice, and the resulting code would be more complicated.

7 Future work

This project is still a work in progress. A lot of improvements can be made to address some of its current limitations. For instance, the only supported data type is 32-bit floating point. It would be desirable to support some other numeric types as well as aggregate data types such as arrays and strings. Also, the language does not support any dynamic memory allocation. It does not support objects or higher order functions. While the usefulness of these features for this particular language is not fully decided yet, we believe they could be integrated smoothly to some degree.

Performance can also be a problem. Our early measurements show the UziScript virtual machine to be 6 to 10 times slower than native Arduino code. While this performance might be good enough for certain applications, we believe there is still room for improvement.

Another potential problem is the program size. Considering that Arduino boards have very limited memory[8], it is desirable that the virtual machine code and the user programs are as compact as possible. The instruction set, although designed to be compact, can be optimized further. And the current compiler does not perform any kind of optimization nor code compression.

[8] For the Arduino UNO: 32 KB Flash, 2 KB SRAM, and 1 KB EEPROM

8 Conclusion

In this paper, we have analyzed the importance of concurrency in programming languages for educational robotics. We showed how Arduino, one of the most popular platforms lacks support for concurrency and we reviewed different approaches to solve this problem.

As an alternative solution, we proposed a concurrent programming language (called UziScript) supported by a virtual machine running on the Arduino. We described the implementation of the language and the virtual machine, and we tested its expressiveness by writing the same program using both standard Arduino code and UziScript. We conclude that the lack of concurrency in the Arduino language forces the user to be extra cautious when writing independent procedures to avoid accidental interactions between them. When the problem requires several simultaneous tasks, the resulting code tends to be complicated and difficult to extend. Using a concurrency library helps but it is not enough to solve the issue and can introduce problems if the user is not aware of how the scheduling works.

Concurrency is hard and not having proper support in the language makes it even harder, especially for beginners. UziScript's simple model for concurrency allows to express simultaneous behavior in a straightforward way. We believe this language has the potential to become an ideal environment for educational robotics.

References

[1] Arduino, "Arduino Playground - Structure," [Online]. Available: http://playground.arduino.cc/ArduinoNotebookTraduccion/Structure. [Accessed 23 Julio 2017].

[2] M. Resnick, "MultiLogo: A Study of Children and Concurrent Programming," Interactive Learning Environments, vol. 1, no. 3, 1990.

[3] G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," ACM Computing Surveys (CSUR), vol. 15, no. 1, pp. 3-43, March 1983.

[4] Grupo de Investigación en Robótica Autónoma del CAETI (GIRA), "Physical Etoys," 2010. [Online]. Available: http://tecnodacta.com.ar/gira/projects/physical-etoys/. [Accessed 15 Junio 2017].

[5] Citilab, "About S4A," 2015. [Online]. Available: http://s4a.cat/. [Accessed 15 Junio 2017].

[6] "Ardublock | A Graphical Programming Language for Arduino," [Online]. Available: http://blog.ardublock.com/. [Accessed 15 Junio 2017].

[7] F. Oudert, "fabriceo/SCoop: Simple Cooperative Scheduler for Arduino and Teensy ARM and AVR," 13 January 2013. [Online]. Available: https://github.com/fabriceo/SCoop. [Accessed 23 July 2017].

[8] A. Wisner, "wizard97/ArduinoProcessScheduler: An Arduino object oriented process scheduler designed to replace them all," 15 January 2017. [Online]. Available: https://github.com/wizard97/ArduinoProcessScheduler. [Accessed 23 July 2017].

[9] I. Seidel, "ivanseidel/ArduinoThread: A simple way to run Threads on Arduino," 15 May 2017. [Online]. Available: https://github.com/ivanseidel/ArduinoThread. [Accessed 23 July 2017].

[10] K. Fessel, "fesselk/everytime: A easy to use library for periodic code execution.," 2 February 2017. [Online]. Available: https://github.com/fesselk/everytime. [Accessed 23 July 2017].

[11] G. Bob, "HaikuVM: a small JAVA VM for microcontrollers," 2017. [Online]. Available: http://haiku-vm.sourceforge.net/. [Accessed 15 Junio 2017].

[12] R. Suchocki and S. Kalvala, "Microscheme: Functional programming for the Arduino," in Scheme and Functional Programming Workshop, Washington, D.C., 2014.

[13] "PyMite - Python Wiki," 2014. [Online]. Available: https://wiki.python.org/moin/PyMite. [Accessed 15 Junio 2017].

[14] C. L. Jacobsen and M. C. Jadud, "The Transterpreter: A Transputer Interpreter," Communicating Process Architectures 2004, vol. 62, pp. 182-196, 2004.

[15] C. L. Jacobsen, M. C. Jadud, O. Kilic and A. T. Sampson, "Concurrent event-driven programming in occam-π for the Arduino," Concurrent Systems Engineering Series, vol. 68, pp. 177-193, 2011.

[16] M. Elizabeth and C. Hull, "Occam-A programming language for multiprocessor systems," Computer Languages, vol. 12, no. 1, pp. 27-37, 1987.

[17] A. W. Roscoe and C. A. R. Hoare, "The laws of Occam programming," Theoretical Computer Science, vol. 60, no. 2, pp. 177 - 229, 1988 .

[18] J. Smith and R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes, San Francisco, CA: Morgan Kaufmann Publishers Inc., 2005.

[19] T. Luerweg, "Stack based programming paradigms," in Seminar Concepts of Programming Languages (CoPL), 2015.

[20] RoboCup, "RoboCup Junior - rescue," [Online]. Available: http://rcj.robocup.org/rescue.html. [Accessed 23 July 2017].

[21] M. Pakdaman and M. M. Sanaatiyan, "Design and Implementation of Line Follower Robot," 2009 Second International Conference on Computer and Electrical Engineering, pp. 585-590, 2009.