ESPeciaL: An Embedded Systems Programming Language

Christopher Métrailler Pierre-André Mudry

University of Applied Sciences Western Switzerland
HES-SO Valais
Rte du Rawyl 47
1950 Sion, Switzerland
{christopher.metrailler, pierre-andre.mudry}@hevs.ch

Abstract

The advent of off-the-shelf programmable embedded systems such as Arduino enables people with little programming skills to interact with the real-world using sensors and actuators. In this paper, we propose a novel approach aimed at simplifying the programming of embedded systems based on the dataflow paradigm. Named ESPeciaL, this programming framework removes the need of low-level programming in C/C++, as the application is written by connecting blocks that produce and consume data. Thus, an embedded application can be described in terms of ready-to-use blocks that correspond to the various micro-controller peripherals and to program function (multiplexers, logic gates, etc.).

The user application itself is written as an embedded Scala DSL. From that code, the ESPeciaL compiler then generates the corresponding C++ code which can be tailored – using different back-ends – to match different embedded systems or a QEMU-based simulation environment. To demonstrate the validity of the approach, we present a typical embedded systems application implemented using ESPeciaL.

Categories and Subject Descriptors B.1.4 [*Hardware*]: Microprogram design aids—Languages and compilers, verification; C.3 [*Computer systems organization*]: Special-purpose and application-based systems—Real-time and embedded systems; D.1.7 [*Software*]: Programming Techniques—Visual programming

Keywords Domain specific languages (DSL), embedded systems, Scala

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SCALA'15, June 13-14, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3626-0/15/06. . . $15.00.
http://dx.doi.org/10.1145/

1. Introduction

The advent of easily programmable embedded systems such as *Arduino*, *MBed* or *Teensy* enables people with little programming skills to interact with the real-world using various sensors and actuators. The simplified programming languages proposed in those systems tightly follow an imperative approach in which no operating system is available. For instance, the toolchain for Arduino-compatible systems rely on the use of a reduced version of the C programming language where the user describes the application by filling the body of a function which is repeatedly called by the framework. In addition to this execution mechanism, this programming language provides simple to use functions (with no pointers for instance) for accessing all the peripherals available on the embedded system.

Largely based on the *Processing* [10] programming language, this approach has the advantage of hiding the complex details of setting-up the micro-controller, such as configuring the processor clock manager, initialize interrupt vectors, configure pins, etc.

This model of programming along with the provided peripherals library considerably simplify the programming of such systems. However, modeling the activity of a micro-controller with an infinitely-repeated loop does not completely capture the reality of embedded systems in which several hardware functions (such as timers or IO peripherals) work in parallel.

In this paper we introduce ESPeciaL (*Embedded Systems Programming Language*), a prototype embedded systems programming language. Implemented using Scala, ESPeciaL is intended to be a simple to use programming ecosystem in which the application is written by connecting blocks that produce and consume data. Thus, an embedded application can be described in terms of blocks from a library corresponding to the various micro-controller peripherals and to program functions (multiplexers, logic gates, etc.). From this description, ESPeciaL can then generate C++ code suitable for standard embedded compilers as well as simulation code aimed at a tailored version of QEMU (see `http://www.qemu.org`).

This paper is organized as follows: in the next section, we give an overview of existing research in the domain and situate the current research. Section 3 will then discuss the various elements of our programming framework. In section 4 we will demonstrate how our approach can be applied to standard embedded systems programming problems before we conclude our paper.

2. Related Work

Block-based visual programming languages such as *Scratch* [8] or *TurtleArt* [12] have been successfully used to teach the basics of programming. In the domain of embedded systems, the same idea of building programs by interconnecting blocks has been applied in several projects such as *Bitbloq* or *Modkit* [9], notably for education (see [1, 5]). One advantage of these languages is that they present the sequential nature of the code in a graphical form. However, those languages do not completely encompass the parallel and asynchronous execution of the code which is performed by special hardware functions that are present in the embedded system. Thus, as embedded code heavily relies on interrupts (for pin inputs, timers, etc.) or hardware blocks (for representing protocols, DMA transfers, etc.), using such programming aids in the context of education present some challenges, as it has been shown for instance in [5].

Dataflow Programming: dataflow and flow-based programming are two other close approaches to visually describe programs that have been used to model embedded systems for education [4, 13]. In both cases, applications are defined using black-boxes components connected as a graph to exchange data and information. In flow-based programming [6], nodes are constantly waiting for messages and data between the nodes are exchanged using asynchronous channels.

Dataflow on the contrary relies on a synchronous approach: this time, the graph is executed in a sequential order and the output of the node is computed when all the inputs have received a valid data. The produced result can then “flow” to the next node.

More complex models such as functional reactive programming exist [7], among others to explicitly include the notion of time. However, the dataflow model is simpler to implement, notably because of the scheduling of the operations, yet it still allow to model all the major components of embedded systems. For this reason it has been chosen as the execution model of our framework, as we will discuss in the next section.

3. The ESPeciaL Framework

ESPeciaL is composed of several components (see Fig. 1). Directly exposed to the end-user is the internal *domain specific language* (DSL). This is the language in which the dataflow graph corresponding to the application is written.

Generally speaking, a DSL extends the host language (here Scala) by adding new constructs that are specific to a given domain (in our case, embedded systems with a dataflow approach). Concretely, this translates in the DSL to *specific types* that encompass the requirements of embedded systems (for instance for bit-based operations) and also to the capability to *connect blocks* inputs and outputs together.

A second important feature of ESPeciaL is the component library containing ready-to-use blocks to build embedded applications with an high abstraction level. Using the description given in the DSL for the interconnected blocks corresponding to an application, a code generator is then able to generate the corresponding C++ code. This generated source file leverages a C++ hardware-abstraction layer (HAL) which enables the code to be compiled and run on different embedded systems or on a software emulator.

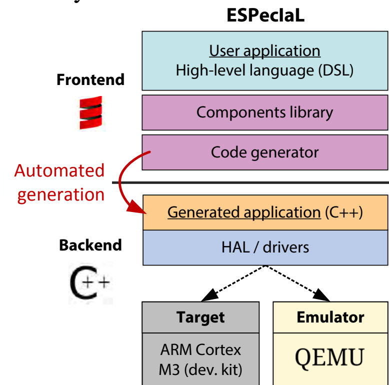

Figure 1. ESPeciaL architecture overview

3.1 Internal DSL Implementation

The DSL proposed mainly consists of two main components:

1. A block linking operator, `-->`, which serves to interconnect blocks outputs to other blocks inputs (available as class attributes), corresponding to hardware and to software behaviors. Block inputs and outputs are typed, an information which is used by the framework to prevent faulty connections.
2. New types specific to embedded systems, notably to support bit based operations, digital and analogue IOs, etc.

A very simple example application is shown in Listing 1, in which a pin of the processor is configured as a digital output to power-on a LED, using a constant block. Another LED blinks periodically.

```
val cst = Constant(bool(true))
val led = DigitalOutput(Pin('C', 3)) // GPIO init.
cst.out --> led.in // Connect and power on LED on pin C#12
Timer(500 ms).out --> DigitalOutput(Pin('C', 4)).in // Timer
```

Listing 1. Basic DSL application code

Several optimizations and checks are performed on the corresponding code tree. For instance, output ports can only be connected to input ports of other components. Moreover,

ports types must be compatible, which is achieved thanks to a specific data type which is transported through the connection. Connections types are checked when a connection is created and explicit errors are printed to help the user correct its code (i.e. for instance if the user tries to connect a `boolean` to an `int` port). In addition to error checking, the DSL also contains features like anonymous components instantiation, variadic constructors and implicit conversions which help to write concise applications in a natural way as depicted in the example 2, which corresponds to the majority function of three digital buttons.

```
val A = IO.btn1.out // Input buttons
val B = IO.btn2.out
val C = IO.btn3.out
val O = IO.led1.in // Output LED
(A & B | B & C | A & C) --> O // Majority function
```

Listing 2. Majority function

Component Library

ESPeciaL provides a library of ready-to-use blocks, presented in Fig. 2. Those blocks are grouped into three categories. The first type of components are *target-specific blocks*. They allow to access to micro-controller peripherals, like GPIOs, analog inputs (ADC), external interrupts or pulse width modulation (PWM) outputs. The second category model *generic components*, like logic gates or mathematical blocks. They can be used configured to use a generic number of inputs, a feature used for instance for the multiplexer block. Finally, the third category regroups components with a *fixed number of IOs*, like inverter gates, constant generation blocks or a PID regulator.

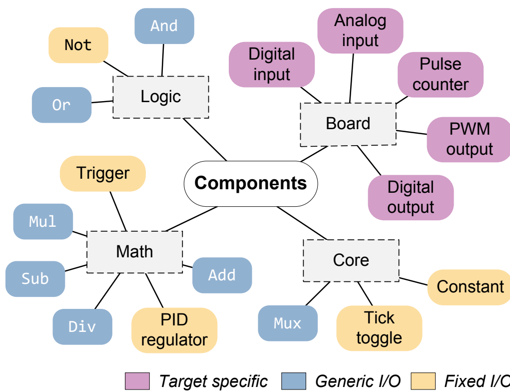

Figure 2. The component library

Because the project is still relatively young, only this limited number of blocks are currently available. However, the framework can be extended easily and new components can be added depending on the user needs.

3.2 Code Generation Pipeline

Once written using the DSL, the application block-diagram is stored in a direct acyclic graph (DAG) and transformed to a C++ code automatically thanks to the code generation pipeline presented in Fig. 3.

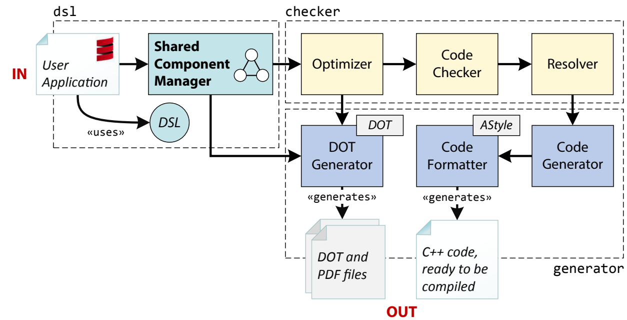

Figure 3. Overview of the code generation pipeline

In the application DAG, nodes of the graph are the components of the program and each arc represents a directed connection from an output to an input port of another node. Each arc is labeled with the data type of the connection (a signed/unsigned integer or a float value for instance). During this phase, error detection as well as optimizations are applied. First, by analyzing the application graph, connections errors and unconnected ports and components can be detected. Second, isolated nodes or paths of unused components can be detected and removed before generating the C++ code of the application.

From a code emission perspective, each component of the dataflow is responsible to generate its own C++, which corresponds to the low-level implementation of the block for the target. The resulting source file is a sequential program composed by codes fragments produced by each blocks which is generated step-by-step by incrementally adding the code of each block. The aggregation order is given by computing the topological sort of the DAG, which is done by the resolver phase of the pipeline. It is worth noting that to be able to transform the graph into a sequential program, the graph must be restricted to an acyclic graph because only this form allows a static scheduling, known at compilation time [3].

Overall, the generated C++ program is divided into several sections (file header, global definitions, functions declarations, main loop, etc.). These sections can be used or not, depending on the block functionality. Once the program is generated, it can then be compiled and executed on the emulator or on the target.

Software Execution Model

The output program is generated specifically for embedded systems in a “bare metal” configuration, i.e. without the use of an operating system.

To accommodate this lack of OS, a simple execution model has been chosen to support a wide range of embedded systems, including those with limited resources. The sequential application runs in a single and monolithic “thread”. The skeleton of the generated code, divided into sections, is

composed of an initialization function which initializes all program blocks and a main loop that executes the sequential application, based on the input-process-output model: first all program inputs are read, then the logic of the application is computed before output program values are updated. In this model, all inputs are read at the same time, and at the end of a cycle, all outputs are updated. In addition to this recurring loop, hardware interrupts are supported (for instance for buttons or timers). Thanks to that, it is for instance possible to guarantee timing constraints which would be otherwise complicated to implement.

This execution model is similar to the *Grafcet* execution model in a programmable logic controller [2], in which a program cycle corresponds to one iteration of the main loop. The code of each component is executed once, in a particular order (determined statically), to take into account the dependencies between them.

3.3 C++ Back-end

Embedded systems exist in many different flavors, ranging from very powerful 32-bits processors with FPU to 8-bits processors with no stack. In addition, each processor is derived into several models containing different kinds of peripherals. To limit the impact of this variety between those processors and to support multiple targets, an *Hardware Abstraction layer* (HAL) has been developed. This software layer, implemented in C++, provide a generic way to access micro-controller inputs, outputs and peripherals (see the target specific components in Fig. 2).

It standardizes the access and the control to different peripherals of the targeted micro-controllers. Therefore, the same application can be executed on different targets without modifications because all IOs and peripherals are accessed through this abstraction layer. For instance, GPIO are automatically configured by providing a port and pin number. After calling the initialization function, values can be read or written using the corresponding function, without using low-level code. To provide this level of abstraction and support multiple targets, a back-end library must be developed once for each hardware target. The work required to support a new hardware target is relatively limited, provided that the hardware can be programmed in C++ and supports interrupts. It mainly consists in providing the required code for the bare-metal initialization as well as writing wrapper functions around the peripheral access functions, the rest being take care of by the HAL. In the current state, two back-ends have been developed as we will discuss in the next section.

4. Experimental Setup

To demonstrate the validity of our approach, we implemented several sample applications. In this section, we will present two of them. The first one was mainly used to verify the basic functions of the framework. The second example is a standard regulation application to show the type and the complexity of programs that can be built with ESPeciaL. In both cases, the target used for the execution is a `STM32F103` ARM 32-bit Cortex-M3 processor. Compilation has been performed using the GNU ARM cross-compiler and we used *GDB* and *OpenOCD* to program and debug the code on the target, with the help of a generic JTAG adapter.

4.1 Digital Logic Application

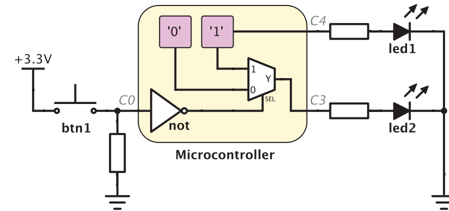

Figure 4. Digital logic block diagram

The first basic application shown in Fig. 4 demonstrates the usage of several ESPeciaL blocks to wire a basic digital function. The code written in the embedded DSL corresponding to this application is as follows:

```
val not = Not() // Not gate with `uint8` conversion
val mux = Mux2[bool]()
val cst1 = Constant(bool(true)).out

IO.btn1.out --> not.in

not.out --> mux.sel
!cst1 --> mux.in1
cst1 --> mux.in2

mux.out --> IO.led2.in
cst1 --> IO.led1.in
```

Listing 3. Digital logic sample code

After being processed by ESPeciaL, the DSL code is then translated to the following C++ code:

```
while(1) {
    // 1) Read inputs
    bool in_C0 = in_cmp02.get();

    // 2) Loop logic
    uint8_t out_cmp01 = !in_C0;
    uint8_t sel_cmp03 = out_cmp01;
    bool out_cmp03;

    if(sel_cmp03 == 0)
        out_cmp03 = false;
    else
        out_cmp03 = true;

    // 3) Update outputs
    out_cmp05.set(out_cmp03);
    out_cmp06.set(true);
}
```

Listing 4. Partial generated C++ code

4.2 Regulation Application

This demonstration application regulates the rotational speed of a computer fan. The target speed of the fan can be set by

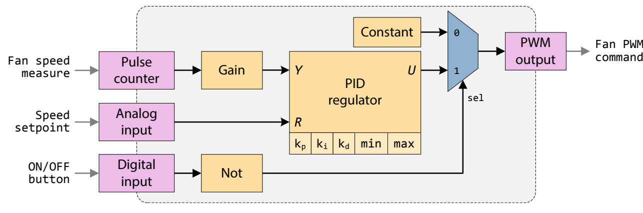

Figure 5. Regulation application block diagram

the user using a potentiometer connected to an analog input pin. A proportional-integral-derivative (PID) controller block, available in the component library, automatically adjusts the speed of the fan depending on the user setpoint and the effective speed of the fan. To achieve this, the speed of the fan is controlled by the duty-cycle of a pulse-width-modulated signal. The effective speed of the fan is measured using a pulse counter block, which captures the external hardware interrupts generated by the fan itself twice per turn. The corresponding block diagram is shown in Fig. 5 and the application code (which has about 20 lines) in Listing 5.

```
val pid = PID(1.0, 0.5, 0, 50, 4000) // Inputs
val pulse = PulseInputCounter(Pin('B', 9)).out
val measure = IO.adc1.out
val speedGain = SpeedGain(4000.0 * 45.0) // Logic
val mux = Mux2[uint16]()
val not = Not()
val pwm = IO.pwm3 // Output

pulse --> speedGain.in
speedGain.out --> pid.measure // PID input measure
measure --> pid.setpoint // PID setpoint from the potentiometer

Constant(uint16(50)).out --> mux.in1
pid.out --> mux.in2
IO.btn1.out --> not.in // Stop the fan using the button
not.out --> mux.sel

mux.out --> pwm.in // Fan PWM command
```

Listing 5. Regulation application code

5. Conclusion

ESPeciaL is still in its infancy and is limited to certain applications because of the DAG-constraints on the code as well as a basic scheduling based on the IPO model. Despite these limitations, we demonstrated in this paper that this programming framework could already be used to write regulation applications for embedded systems running on real hardware. The programming model – which remains simple in this first iteration – includes some of the specificities of embedded systems (such as interrupts) in a transparent manner. Based on our programming experience with standard embedded code, this early implementation of ESPeciaL seem to demonstrate that the dataflow-based DSL description is shorter than its C++ counterpart. In addition, having the possibility to describe the application in terms of interconnected blocks provides a level of abstraction that was very convenient for developing the presented examples, which is encouraging. In addition, the presence of a simulator-based output enabled us to automate the testing of complete regression suites for the code generator.

In further work we will introduce more complex peripherals in the simulator. In addition, we will also add to the framework a multi-tasking OS which will remove the limitations of the IPO model to capture even more complex application scenarios. For this future implementation, we will integrate the lightweight modular staging approach [11] to generate the code. We will also consider the implementation of a graphical editor for the dataflow graph which could directly generate the DSL code.

ESPeciaL is an open-source project available at `https://github.com/hevs-isi/especial-frontend`.

References

[1] Martin Grimheden and Martin Törngren. What is embedded systems and how should it be taught? *ACM Transactions on Embedded Computing Systems*, 4(3):633–651, 2005.

[2] Anaïs Guignard and Jean-Marc Faure. Formal models for conformance test of programmable logic controllers. *Journal Européen des Systèmes Automatisés*, 47(4-8):423–446, 2013.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[4] David Jeff Jackson and Paul Caspi. Embedded systems education: future directions, initiatives, and cooperation. *ACM SIGBED Review*, 2(4):1–4, 2005.

[5] Peter Jamieson. Arduino for teaching embedded systems. are computer scientists and engineering educators missing the boat? *Proc. FECS*, pages 289–294, 2010.

[6] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[7] Ingo Maier and Martin Odersky. Deprecating the observer pattern with Scala.React. Technical report, 2012.

[8] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[9] Amon Millner and Edward Baafi. Modkit: blending and extending approachable platforms for creating computer programs and interactive objects. In *Proc. of the 10th Intl. Conf. on Interaction Design and Children*, pages 250–253. ACM, 2011.

[10] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, 2014.

[11] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proc. of the 9th Intl. Conf. on Generative Programming and Component Engineering*, pages 127–136, New York, 2010. ACM.

[12] Claudia Urrea and Walter Bender. Making learning visible. *Mind, Brain, and Education*, 6(4):227–241, 2012.

[13] Marilyn Wolf. *Computers as components: principles of embedded computing system design*. Elsevier, 2012.