ambient intelligence
A framework for cyberphysical laboratories in Information Systems and Computer Engineering courses

João Henrique Fernandes Glória
joaohgloria@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal
October 2018

Abstract
Embedded systems are present every day in our lives. From cars to planes, to our appliances. In recent years, efforts have been made to integrate teaching programs that address this issue in colleges. Despite all the proposals to facilitate the learning of embedded systems, there are still obstacles in the learning and production of these systems in the classroom. In order to address the difficulties inherent in the production of embedded systems, in recent years some tools have been developed that allow programmers to focus on the applications they want to develop rather than on these difficulties. These tools go through the development of operating systems, virtual machines or high-level abstractions. This work presents a framework focused on the learning of embedded and cyberphysical systems in IST. It consists of three components: a workbench, a set of practical exercises and a new code generation tool focused on the development of distributed embedded applications. After the introduction of the workbench and the set of exercises in the classroom, we came to understand that this type of artifacts stimulates teaching and enthusiasm. The code generation tool has shown promising results but still needs some improvements and adjustments. There are still some tests to be done in the classroom with this framework, but we believe that this may be the way to teach applications for embedded and cyberphysical systems, especially in software-oriented courses, where affinity with low-level systems and hardware is lower compared to software.

Keywords: Embedded systems; laboratory framework; code generation tools; Arduino; education.

1. Introduction
Embedded and cyberphysical systems have been around for years now. The industry has been using them as part of bigger structures, such as cars, airplanes or even home appliances. Nonetheless, the introduction of courses related to embedded and cyberphysical systems in higher education had only gained some traction more recently, and most universities are still struggling to give their students the best methods and instruments for a better understanding and preparation of related topics. In the Information Systems and Computer Engineering bachelor’s degree of Instituto Superior Técnico (IST) of the University of Lisbon, there is no course lecturing software for embedded and cyberphysical systems. The Master’s degree specialization in Robotics has an outdated curriculum when compared to the state of art in embedded systems education, not having adopted nor a microcontroller nor a laboratory installation to lecture practical classes, at the time this work started being written. Despite enjoying the subjects lectured, previous experiments proved that students would prefer the use of real hardware instead of developing for simulators or on top of general-purpose programming environments [13]. New curricula have been being proposed by various universities for the lecture of courses related to software development for hardware. In a general way, for all found literature on experiments for new curricula adoption, universities opted for a Problem-Based-Learning (PBL) program defending that it would better prepare students for future roles in the industry [13, 5, 15, 6]. PBL programs intend to teach multiple topics (e.g. resources limitations, multi-threading, communication protocols, real-time, etc.) through the use of laboratory exercises and projects. When compared with traditional education methods, PBL revealed really promising results: in the University of Granada, there was an increase of 32% in the students’ programming levels [15]; in the Royal Institute of Technology, students who experienced a PBL approach had higher grades on exercises, projects, and exams, higher attendance to classes and more disposition to work extra-hours, compared to students on a traditional education approach [5]; in the Hong Kong University of Science and Technology, the students stated
that the use of this approach increased their interest and motivation on the topics lectured [13].

Despite the inclusion of microcontrollers in education, in order to improve the teaching of embedded software courses, the challenges faced in the design and development of such systems are too complex to address in a course duration time (usually a semester or a year). Michael V. Woodward et. al point out 5 challenges to the development of embedded software: complexity (due to feature bloat onto single systems and network complexity), optimization (long and costly development cycle from design to test), interdependency, verification (checking if the system meets specification is costly and time expensive) and tools (short in diversity and with lack of integration to support all the development cycle) [17]. Many authors point out the resource limitations, like low processing power, as one of the challenges on the embedded systems development [8, 12, 4, 11, 10]. Limitations such as small memory, small processor word sizes, slow clock frequency, energy constraints or low-bandwidth are all barriers to software development. Given that embedded systems are often physical systems, many authors point out that concurrency and real-time are intrinsic features of such systems. Reliability of these systems is associated with the two characteristics and failure is often not tolerable (embedded systems are present in all types of devices such as cars and airplanes). The main challenge here is the lack of abstractions when compared with general-purpose programming such as desktop applications development where real-time is rarely a demand and concurrency is often solved with program ordering or abstractions like semaphores or locks [8, 11, 9, 16, 3].

Some authors already tried to enhance the support on development of embedded systems using different approaches. TinyOS [11] is an operating system developed to help in the development of sensor networks. Other approaches involve the creation of virtual machines [10, 7, 4] or high-level abstractions with new definition languages or extensions of already existent programming languages [12, 3, 7, 14, 4]. A common trend among this works is the creation of high-level abstractions. These high-level abstractions are prevalent in the basic structural components of the applications. Some authors propose abstractions for single units of work [11, 7, 12, 14], while others like TinyGALS propose a whole level of abstraction to define complete systems through composition [3]. The most common concern for these tools tends to be the abstraction of communication through mechanisms that allow intra and inter-process communication through synchronous and asynchronous mechanisms. Concurrency is another highlighted concern with solutions that go from atomic sections (non-preemptive) versus preemptive tasks, on TinyOS [11], to isolation of context and one-word heaps, on Maté [10]. Despite all these mechanisms to help programmers abstract from major embedded and cyberphysical concerns, most previous works tend not to concern about tooling such as code optimization or debug. Only Spash presented a whole debug system with multiple debugging modes [7] and only TinyOS and Flask concerned about code optimization at compile and pre-compile time [11, 12].

In this work we propose a complete framework to teach and help students and engineers with a lack of software for embedded hardware skills, in the development of software for embedded hardware. Specifically, we are targeting students of Information Systems and Computer Engineering courses who usually fit this profile.

2. Architecture

The framework proposed is composed of three components. For the first component, we envisioned a workbench where students could keep in touch with the hardware and could practice some exercises. The second component is a set of exercises for students to have challenges where to apply the workbench. The third component is a code generation tool to help students solving the exercises.

2.1. Workbench

The workbench consists of a set of materials, given to a student or groups of students so that they can develop or learn embedded and cyberphysical systems. The set of materials consists of one to three microcontrollers, breadboards, sensors, actuators, etc.

2.2. Set of Exercises

The set of exercises proposed for the framework has as the main goal helping students to progressively learn the concepts behind the development of embedded and cyberphysical systems and to put them in contact with the selected microcontroller. For every exercise, students should develop all the software and assemble all the hardware. The exercises are divided as follows: exercise 1 is supposed to put students in contact for the first time with the selected microcontroller and one or more actuators; exercise 2 should challenge them to use sensors in their assemblies to control the actuators used in the previous exercise; exercise 3 puts students in contact with inter-boards communication, teaching them about an available communication protocol for their microcontroller, and to create a distributed system; exercise 4/project is a real-life problem where students should be challenged with a real-life scenario.
2.3. Code Generation tool
The Code Generation tool allows to quickly develop embedded and cyberphysical distributed solutions. It is supposed to offer a new high-level abstraction language (almost natural) to ease the programming of embedded and cyberphysical systems. It is composed of four components:

2.3.1 Definition Language
The goal of the definition language is to abstract low-level details of programming in embedded software. The definition language presents three main abstractions: agents define a physical bounded system (i.e., a single program that runs on a single microcontroller) and execute an entire program in an infinite round-robin loop; modules define the most basic unit of behavior and implement a set of self-contained instructions; links define the connection between agents and thus, the creation of distributed applications.

2.3.2 Framework
The framework is a set of directories to organize programs and finally produce compilable deliverables.

2.3.3 Communication Library
All communication must be transparently abstracted so that users of the framework waste no time developing or worrying about it. In order to achieve this, a library with a set of functions was developed to create a communication API that removes almost all communication details from the code itself.

2.3.4 Translator
The translator is the software module that generates code to the target microcontroller. It is divided into two processes: parsing the Definition Language is the initial phase where all the program written with the definition language is validated and parsed into internal well-known structures; composing target microcontroller programs is the final phase where the internal well-known structures are procedurally translated into the microcontroller code.

3. Implementation
3.1. Workbench
The Workbench is composed of one-to-three Arduino Starter Kits [1] depending on the exercises being performed by the users of the framework. Each Arduino Starter Kit includes an Arduino board, a breadboard and set of additional hardware.

3.2. Set of exercises
Every exercise is intended to address one or more characteristics of embedded software or cyberphysical systems. All the exercises were designed for the Arduino board.

The first exercise tests the ability for the development of a simple cyberphysical system. Users should be able to assemble a circuit with four LEDs and produce the necessary software to switch the LEDs on and off.

The second exercise designed for the framework is an extension of the first one, in the sense that we will keep using the LED as the actuator. Nonetheless, instead of controlling the LEDs via software, they will be controlled via sensors feedback. The used sensors are a dimmer, and sensors for light, presence and temperature.

The ultimate goal of the third exercise is to control the LEDs with the sensors, but this time using two Arduino boards: one to control the LEDs and other to read from the sensors. This particularity means no shared resources like memory. In order to make this system work, users should use the I2C wired protocol to make boards communicate with each other.

In the last exercise, the project, users should build a traffic-light system. This system should be a simulation of a crossroad controlled by two traffic-lights with three colors for vehicles, two colors for pedestrians, and a button to signal pedestrians presence, for each traffic-light. The traffic-lights should not communicate with each other, but instead with a mediator which coordinates communication and fault-tolerance. Each traffic-light and mediator must be an Arduino board (3 Arduino boards).

Every exercise assignment paper has: a goal with the objective for the exercise; an estimation time of completion; a description introducing the problem/exercise; references with links to media that might be helpful during the execution of the exercises; recommendations with a list of safety behaviors; an introductory theory (optional) to explain some concepts that must be understood prior to execution of an exercise; an assembly scheme; and question and answer areas left in blank to be adapted by the lecturer.

3.3. Code Generation tool
3.3.1 Definition Language
The three abstractions provided by the tool – modules, agents and links – and their relationships are represented figure 1. All abstractions are described using XML because it is easy to learn and well-known markup language in the industry, generally known by engineering graduates.
Module

Modules are the most basic components of the design. They describe parts of behavior and the flows of communication in an application. One or more modules compose an agent. Every module has: one unique name to identify it in the entire application; one function to allow users to include snippets of runnable code in Arduino; one optional port to define input channels of communication (modules expected to receive data from others should have a port defined). Listing 1 depicts one module definition example.

Functions can be imported from two sources (packages). Standard package (std) functions are built-in functions ported with the tool. Custom package (cst) functions are user-defined functions to extend the tool behavior. Functions have two constraints: they must never return values because every function must be completely independent; and, functions scope is closed for the entire application removing visibility from every other function defined in other application modules, helping programmers define applications free of unintended reassignments.

Ports have three internal properties: data-type defines the type of messages allowed through the communication channel (C++ primitive data types only); pool-size defines the maximum number of messages stored in a module (default value is 1); substitution-policy defines whether new messages should be dropped (true) or inserted (false) when the pool is full (default is true).

Agent

Agents represent the basic unit of processing and a shared-memory system (i.e. the program that will run on top of an Arduino board). Each agent will produce a runnable program on Arduino and thus, the goal of each agent is to be translated, compiled and uploaded into one Arduino. Every agent has three internal properties: one unique name that identifies it across the entire application; one or more modules (the order of declaration of modules is very important because since the Arduino computing model is a round-robin loop the order of declaration will correspond to the order of execution); and one address per protocol to define how agents are physically addressed and must be addressed in the application. Listing 2 depicts one agent definition example.

Addresses definitions are set using the notation <protocol>::<address>, where protocol refers to the communication protocol and the address refers to the port or address used for that protocol.

Link

Links describe how agents relate to each other in terms of communication protocols/technologies. While agent addresses only specify addresses for a given communication technology, links describe how those agents are connected in terms of those technologies/protocols. Every link has two internal properties: one protocol to describe which communication protocol/technology is to be used when binding two agents (At this time we only support I2C protocol); two agents that allow users to form couples of agents. Listing 3 depicts one links file definition example.
3.4. Framework
The tool provides an organized framework of directories, supporting users to produce their programs. The framework intends to help them better organize the components that constitute the program itself in a directory work tree. The root of the application is the source directory (or src) which contains the agents, functions and output directories and the links.xml file. Inside the functions directory, users will find two more directories related to the packaging system (standard and custom packages). The standard directory contains the standard functions packed with the tool. The custom directory will start empty. Inside it, users can create their own functions extending the behavior of the tool. The output directory is where the translation results live. Once the translation process finishes, users will find .ino files, with the names of their agents, ready to compile and deploy.

3.5. Communication
Our tool aims to address the communication challenges part of the development of cyberphysical distributed applications. In those challenges we include addressing local and remote modules and agents, marshaling and unmarshaling of messages, and communication protocol selection for message dispatching. The tool offers five user-level methods, to handle message-passing and communication synchronization, that will be described in the following subsections.

3.5.1 Message passing
put method ensures the message passaging from one producer module to a consumer module. It is a non-blocking method, which means that execution in the producer module resumes after its invocation. put receives as parameters the consumer module name, and the payload of the message itself.

Listing 4: Interface of put() method

```c
put(char∗ consumerModule, T payload) : void
```

get method is responsible for retrieving messages from consumer modules’ buffers. It is a blocking method, which means execution will block until there is something to be read from the consumer module’s buffer. get always returns the type of data defined in the consumer module port, removing that same value from the buffer to ensure that it is read only once.

Listing 5: Interface of get() method

```c
get() : T
```

3.5.2 Synchronization
isAvailable method is responsible for checking whether there is something in the consumer module’s buffer to be retrieved with get. Since get is a blocking method, isAvailable is especially important to protect programs from unintended execution blocking when there is nothing to be read from the buffer. It returns a boolean signaling whether the buffer has messages – (returns true) or not (returns false).

Listing 6: Interface of isAvailable() method

```c
isAvailable() : bool
```

get method provides a way of requesting data. The consumer module sends a message to the producer module signaling some request.

Listing 7: Interface of request() method

```c
request(char∗ producerModule) : void
```

hasRequest method pairs with request, on the producer module side, and behaves as the method isAvailable in a way that it must synchronize communication. It is a non-blocking method that receives the consumer module name that requested a message as a parameter and checks if it was previously signaled by it. This method is important to control communication overheads. Enclosing put operations in a hasRequest query reduces the number of messages between modules by ensuring that the receiver module is expecting a message, instead of flooding the network with unexpected messages.

Listing 8: Interface of hasRequest() method

```c
hasRequest(char∗ consumerModule) : bool
```

3.5.3 Communication Manager library
The tool developed for this work intends to provide seamless and transparent communication, between nodes in a network, to users with no programming effort. The Communication Manager is a library packed with the tool to support this feature. It supplies not only the above set of communication methods but also a pair of structures – Mapping and Flag. As Mapping acts as a support structure for addressing tables (explained in subsection 3.5.4), the Flag structure provides support for the request()/hasRequest() synchronization methods.
3.5.4 Addressing table

During translation (section 3.6), the tool generates an addressing table to support communication between participants in the produced system. The addressing table takes advantage of Mapping structure. Mapping has two properties which promote a two-level addressing mechanism: name, which refers to the first addressing level (the module name); and address, which refers to the second addressing level (the agent address).

When modules execute put or request methods, a lookup is performed on the addressing table. It starts by looking in the first addressing level, the module passed as the argument in the above methods and then, looks on the second addressing level – the agent address. If the address corresponds to a local address (-1), the message is handled internally at the agent. If the address is not local, the message is then marshaled, sent to the target agent through the address and protocol provided and then unmarshaled again in the destination agent, like illustrated in figure 2.

Figure 2: Local communication vs. remote communication

3.6. Translator

Translation is the key feature of the Code Generation tool. This process allows XML descriptions of an application to be translated into C++ programs, valid for compilation in the Arduino compiler. The translation process has two steps: parsing of program definitions and .ino (Arduino file extension) programs composition. The translator was developed in nodejs (ECMAScript) allowing it to run on multiple operating systems.

3.6.1 Parsing program definitions

The tool starts by looking into the agents’ directory as the entry point of translation. The translator looks for .xml files and, one at a time, starts by deconstructing the definitions into JSON. Having the transformation applied, it is possible to start creating ECMAScript Agent objects. Agents names and addresses are validated for uniqueness.

Modules are then parsed from the agents’ definitions, where ECMAScript Module objects are created. The name and port of the modules are easily assigned to the Module ECMAScript object, but the function attribute relies on .ino files where the actual code lives. The translator reads the function attribute and splits its value into package and function name. The tool looks into the framework functions directory and looks for custom or standard directories depending on the package. Then, it uses the already read function name to look for its related .ino file. This type of files must have a specific boilerplate, which is the Arduino’s setup and loop functions so that the tool can retrieve every line of those functions into two ECMAScript Module object’s collections: Setup and Loop.

Finally, the translator looks for the links definition in the framework (links.xml). The existence of this file is mandatory even if not used. As it happens with the agents’ definitions, the links XML definition starts by being deconstructed into a JSON definition. The parsing of the links is supposed to create a graph of connections. For every link in the definition, the agent attributes are read and a node is created (if there isn’t one already). The protocol value is then read and the nodes are connected using that protocol.

3.6.2 Composing .ino programs

.ino programs are the ones that actually run on top of Arduino boards. In order to use our tool to create those programs, first we need to compose them. Having Agents and Modules objects and the Links graph created it is possible to start composing the .ino files (one file per Agent object). For every file the translator outputs: dependencies; global variables; the setup and loop methods; the module functions code; the i2C callback routine.

Dependencies: User dependencies included in module functions are included in the final program. This allows users to import extra behavior from external/third party libraries without the need for developing extra code. Apart from users’ code dependencies, there are some other dependencies related to the correct behavior of our tool’s generated programs (i.e., Arduino Wire library, and our Communication Manager library).

Global variables: Three possible global variables can be generated from the translation process: Buffers, Mappings and Flags (example in listing 9). Buffers are data arrays that store modules incoming messages. Therefore, there is a one-to-one
relationship with modules ports. For every port found in an agent module, a buffer of the port data type and size is created (camel-case notation <moduleName>Buffer).

Mapping denotes an entry in an agent addressing table. This structure contains the name of a module, a buffer and its size, a type, and finally one physical address. The number of mappings reserved as global variables for a given agent is equal to the number of local modules owning a port plus the number of non-local modules used as destination modules using put and request communication methods.

Flag is another Communication Manager’s structure that denotes the presence of requests from modules. This structure contains the consumer module name (the module requesting messages), the producer module name (the module whose messages was requested) and a boolean that marks the existence of requests. The number of flags is equal to the number of modules used as destination modules used in hasRequest methods.

Listing 9: Global variables example: buffer, mappings and flags

```c
char moduleBuffer[1];
Mapping mappings[3];
Flag flags[3];
```

Setup method: is a one-time execution method from Arduino programmatic model, usually used to initialized external libraries and external variables and objects. In our tool, we use it to: initialize mappings and flags objects; initialize necessary communication libraries; include all setup code from modules functions.

Mappings are initialized for certain modules, on every agent, depending on the module relative location to that agent: for local modules owning a port the Mapping structure is initialized with the module name, port data type, an address with value -1, the buffer created for that port, and its size (example in listing 11); for non-local modules found in local puts and requests arguments, the Mappings are initialized using only the module name and the destination address of that module (listing 11).

Listing 10: Mapping initialization of local module with char type port of size 4

```c
mappings[0] = Mapping( "module", "char", -1, moduleBuffer, 4 );
```

Listing 11: Mapping initialization of non-local module with I2C address 248

```c
mappings[1] = Mapping( "nonLocalModule", 248 );
```

For each hasRequest method found, a Flag structure is initialized with the hasRequest argument as consumer module, the current module as producer module, and false as the initial value (listing 12).

Listing 12: Flag initialization

```c
flags[0] = Flag( "moduleA", "moduleB", false );
```

The last two steps of an agent’s setup translation are initializing the communication libraries (if needed) and including programmers’ setup code (listing 13).

Listing 13: Wire library initialization on address 11 and callback registration

```c
Wire.begin(11);
Wire.onReceive(onReceiveCallback);
```

Loop method: is the most straightforward phase of translation. The translator only outputs into the .ino file the name of agent’s modules, by the order of declaration, in a C++ method notation (i.e. followed by () as in module()).

Modules functions: are where most of the user’s code lives. For every agent being translated, the translator loops through all modules parsed in the parsing phase (subsection 3.6.1) writing the lines of code read into .ino file. There are two types of translation: lines of code that do not contain any Communication Manager methods (subsection 3.5) are written to the .ino file as they were written by the user; those which contain Communication Manager lines pass through a translation process.

When invoking the get method, users do not pass any arguments because it should always run in the scope of the module where the method is being called. Hence, the translator injects the arguments necessary to use the get method in the produced program. The necessary arguments for get are the global buffer created for the module getting the data and its size (example in listing 14).

Listing 14: get() method translation

```c
// user exposed version of get()
char c = get();
// translated version of get()
char c = get(moduleBuffer, 2);
```

Also for isAvailable() method the translator adds the argument necessary for it to run on the final version, which is the global buffer created for the module getting the data (example in listing 15).

Listing 15: isAvailable() method translation

```c
// user exposed version of isAvailable()
if (isAvailable()) {
// translated version of isAvailable()
if (isAvailable(moduleBuffer)) {
```

For hasRequest, the translator uses that consumer module name, the name of the module being translated and the agent being translated flags
global variable to write the translated version of the hasRequest method (example in listing 16).

Listing 16: hasRequest() method translation

```c
// user exposed version of hasRequest()
if (hasRequest(moduleA))
// translated version of hasRequest()
if (hasRequest("moduleA", "moduleB", flags))
```

For request, there are two translated versions of the method being written to the .ino file: first a version for local producer modules (i.e. modules living in the same agent as the module being translated), which receives as arguments the consumer module name, the producer module, the mappings global variable and its size and the flags global variable; and a second version for non-local producer modules, which receives all of the first with exception of the flags global variable (example in listing 17).

Listing 17: request() method translation

```c
// user exposed version of request()
request(moduleB);
// translated version of local request()
request("moduleA", "moduleB", mappings, 2, flags);
// translated version of non-local request() request("moduleA", "moduleB", mappings, 2);
```

put translated version will contains the destination module name, the message payload, the mappings global variable and its size (example in listing 18).

Listing 18: put() method translation

```c
// user exposed version of put() method
put(module, payload);
// translated version of put() method
put("module", payload, mappings, 1);
```

I2C callback routine: the Arduino’s Wire library requires a callback routine, which is triggered whenever communication is detected. In our application, two types of messages reach agents: remote put and remote request. To include the callback routine, the translator creates a function called onReceiveCallback and includes all necessary code to handle its messages. The included code is, as the dependencies, conditional. If the program contains flags global variable, requests are coming in and this type of message must be handled. Then the translator includes the code snippet that handles the remote requests. If agent’s modules contain at least one port, put messages will be received. These messages must be handled and the translator includes necessary handler routine.

### 3.6.3 Compile and deploy phases

Once the translation is complete, users have their output directory populated with .ino files, with the names of their agents, ready to compile and deploy. These steps strongly depend on Arduino Software (official Arduino IDE) which provides both features.

### 4. Evaluation

#### 4.1. Workbench and Set of Exercises

After the introduction of the Workbench and Set of Exercises in a Information System and Computer Engineering course in 16/17 and 17/18, according to the lecturer, the materials and the exercises were successful in the sense that they achieved their main goals: they were fit to learn and train software for embedded and cyberphysical systems and they were engaging enough to keep students interested. Also, there was an increasing number of students attending this course in 17/18 that there were in 16/17 or 15/16. This number has increased again in 18/19.

#### 4.2. Code Generation tool

Because the Code Generation tool is continuously being developed and improved, we were cautious and did not include it in the classroom yet. The tests created for the tool evalua ease-of-use with two metrics: lines of code and time of development. Also, given that this tool falls into the code generation genre, we tested the size of programs in memory. There were two groups of 5 engineers each, to test the tool. The engineers had one to five years of working experience in software development, but none experience with Arduino nor software for hardware development. Both groups had to solve simpler versions of the proposed set of exercises, one using C++ and other with our proposed tool.

All the exercises were supervised to give support to the subjects and to take notes from their opinions and difficulties. Both groups were given the necessary tutorials and documentation to complete the exercises (tool documentation [2]).

#### 4.2.1 Results

Regarding lines of code, the tests revealed promising and warning results. For small programs, the average lines of code produced for the tool were 1.5 to 4 times bigger than the ones produced with C++. Medium-sized programs, where communication between boards was involved, had an average of approximately the same lines of code between both approaches. For greater distributed programs with the intensive use of communication between boards, programs developed with the tool had 1.25 times fewer lines than the ones produced with C++ and 2 times fewer lines of communication logic.

Considering time spent developing the exercises, this is the metric that raises more alarm. Development with the Code Generation tool was almost never faster than developing with C++. The worst
scenario was for the first contact with the communication abstractions where subjects took 3 times more developing a program than the ones with C++. The only situation where development with the tool was faster than with C++ was the transformation of a non-distributed application into distributed (third exercise).

Regarding the size of programs in memory, for small programs with no use of communication abstractions, both approaches consumed approximately the same RAM and flash memory. For bigger programs with inter-board communication, the programs produced with the tool revealed some optimization over RAM since it consumed approximately the same or fewer memory than the ones developed with C++. The same does not occur with flash memory because the Communication Library alone weights 1468B in memory. The worst scenarios were when programs demanded communication abstractions using the tool but not with C++. In the second exercise (where this scenario occurs) the program developed with the tool consumed 210B of RAM and 2622B of flash memory, while with C++, 9B and 968B, respectively.

4.3. Conclusion

4.3.1 Workbench and Set of Exercises

We honestly believe that the use of these materials and exercises proposed as part of the framework have a great influence on the learning of embedded and cyberphysical systems. The increasing number of attendees proves that the use of such framework not only engages students but, reduces the fear of working with hardware.

4.3.2 Code Generation tool

Considering the notes taken and the three metrics used to evaluate the tool, we concluded that there are pros and cons when using the tool.

Pros: the effort to produce distributed applications using communication between boards is smaller; the process of transforming already developed non-distributed applications into distributed ones is easier than when developing natively; as a code generation tool, it is already optimized to produce RAM and flash performant programs.

Cons: the setup of the application parts (agents, functions, and communication) must be automated; more built-in functions should be provided to ease and accelerate the development of applications; users must be previously educated since the development paradigm of the tool is different from what they are used to.

5. Conclusion

The Workbench and a Set of Exercises already introduced in an Information System and Computer Engineering course are revealing great success since students are losing the fear of working with hardware and getting really interested in it. The number of attendees of this course is increasing since the introduction of the framework and we believe that there is a direct relationship with it.

Regarding the Code Generation tool, the design of the definition language was a challenge due to the fact that we wanted to abstract the users from details, but at the same time maintain a friendly software dialect. The other challenge was to, given that definition language, apply the necessary transformations for applications defined in XML and C++ to run Arduino boards.

Some tests were made with two groups, one using C++ and other the code generation tool to develop Arduino applications. Despite some bad results when taking time of execution into account, in the other two metrics (lines of code and memory usage) the tool revealed promising. Also, the reactions of the students using the tool were satisfactory.

The tool still needs improvements and fixes, but this might be the direction to take when teaching software for embedded and cyberphysical systems in schools and universities, mainly on software-oriented courses, where the time is finite and the affinity with hardware and low-level systems is far lower than with software.

5.1. Future work

We envision some additions not to the workbench itself but to the courses laboratories workspace. As an extension of the workbench, we would like to create service-oriented stations (nodes or groups of nodes) around laboratory rooms, controlling actuators or sensors. The rationale is that students could interact with elements of the classroom (e.g. temperature sensors, DC motors, etc.).

The Set of Exercises is stable and seems appealing to the students. We have no future plans for it.

Regarding the Code Generation tool, taking into consideration the results and notes taken, there are some corrections and implementations that should be done to turn the tool more robust and usable. In the future, we must implement more built-in functionality in the tool, and automate the setup of creating the various parts of the application that, at this moment, users do manually. Also, there are some more features that we would like to have: multiple communication protocols/technologies to support more than I2C protocol; Deploy on-the-fly: to deploy code to nodes already in production; network topologies to allow the users to choose specific network topologies to which the tool should adapt and configure by itself; services architecture so that agents could consume information from others by requesting that information
instead of sending it; a **graphical interface** to ease even more the use of the tool.

**Acknowledgements**

Thanks to my parents that allowed me and supported me in my studies in the prestigious college Instituto Superior Técnico, and that made me believe that I could finish this work despite all the adversities. Thanks to my girlfriend Ana Rita that supported me during almost every line of this work and gave me the strength needed to keep writing it until the end. Thank you to my three friends Michael, Volo, and Luís for reading and reviewing this work countless times and for all the support given. Finally, a big thank you to my professor and advisor Alberto Cunha. Thank you for always believing in me, and for all the trust and respect shown during the elaboration of this work. This work is dedicated to you all. Honorable mention to all the participants in the tests.

**References**


