Ambient Intelligence

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

João Henrique Fernandes Glória

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Alberto Manuel Ramos da Cunha

Examination Committee

Chairperson: Miguel Nuno Dias Alves Pupo Correia
Advisor: Alberto Manuel Ramos da Cunha
Member of the Committee: Renato Jorge Caleira Nunes

November 2018
Acknowledgments

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 Luis 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.
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 produced 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. This framework 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 understand that this type of artifacts stimulates teaching and enthusiasm. The code generation tool has shown promising results but still needs some improvements and corrections.

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 inferior than with software.

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

Os sistemas embebidos estão presentes todos os dias na nossa vida. Desde carros, a avões, aos nossos electrodomésticos. Nos últimos anos têm sido feitos esforços para integrar programas de ensino que abordem este tema nas faculdades. Apesar de todas as propostas para facilitar a aprendizagem de sistemas embebidos continuam a existir obstáculos na aprendizagem e produção destes sistemas na sala de aula.

Para endereçar as dificuldades inerentes à produção de sistemas embedados, nos últimos anos têm sido produzidas algumas ferramentas que permitem aos programadores focarem-se nas aplicações que pretendem desenvolver e não nestas dificuldades. Estas ferramentas passam pelo desenvolvimento de sistemas operativos, máquina virtuais ou abstrações de alto-nível.

Este trabalho apresenta uma framework focada na aprendizagem de sistemas embebidos e ciberfísicos no IST. Esta framework é constituída por três componentes: uma bancada de trabalho, um conjunto de exercícios práticos e uma nova ferramenta de geração de código focada no desenvolvimento de aplicações embebidas distribuídas.

Após a introdução da bancada de trabalho e do conjunto de exercícios na sala de aula, compreendemos que este tipo de artefato estimula o ensino e entusiasmo. A ferramenta de geração de código demonstrou resultados promissores, necessitando ainda de algumas melhorias e correções.

Ainda existem alguns testes a realizar na sala de aula com esta framework, mas acreditamos que este poderá ser efectivamente o caminho para o ensino de aplicações para sistemas embebidos e ciberfísicos, principalmente em cursos orientados a software, onde a afinidade com sistemas de baixo nível e hardware é inferior à com software.

Keywords: Sistemas embebidos; framework para laboratório; ferramentas de geração de código; Arduino; ensino.
## Contents

**List of Tables**  xi

**List of Figures**  xiii

**List of Listings**  xv

**Acronyms**  xvii

1 Introduction  1

1.1 Context  1

1.2 Proposal  2

1.2.1 Workbench  3

1.2.2 Set of exercises  3

1.2.3 Code generation tool  3

1.2.4 Goals  3

1.3 Outline  3

2 Background and State of Art  5

2.1 Background  5

2.1.1 Embedded systems  5

2.1.2 Arduino  7

2.2 Support on the development of embedded and cyberphysical systems  8

2.2.1 TinyOS  9

2.2.2 TinyGALS  10

2.2.3 Flask  11

2.2.4 Mate  13

2.2.5 Agilla  14

2.2.6 Splish  16

2.2.7 WuKong  17

2.3 Conclusion  18

2.3.1 Features table  20

3 Architecture  23

3.1 Overview  23

3.2 Workbench  23

3.3 Set of Exercises  24

3.4 Code Generation tool  24

3.4.1 Definition Language  24

3.4.2 Framework  25
List of Tables

2.1 State of art comparison table .................................................. 21
5.1 Lines of code from control group (average) ................................. 50
5.2 Time spent coding (average) ...................................................... 50
5.3 Memory usage by control group (best solution) ............................. 51
List of Figures

2.1 Arduino IDE .................................................. 8
4.1 Definition Language model UML diagram .......................... 30
4.2 Local communication vs. remote communication ....................... 37
4.3 Parsing program definition - Part 1 of 4 ............................. 38
4.4 Parsing program definition - Part 2 of 4 ............................. 39
4.5 Parsing program definition - Part 3 of 4 ............................. 40
4.6 Parsing program definition - Part 4 of 4 ............................. 41
A.1 Arduino UNO board ............................................ 88
Listings

4.1 Module definition with name, function and port ................................................. 31
4.2 Custom function example with injected variable .................................................. 31
4.3 Agent definition with two modules ........................................................................ 33
4.4 Links example definition ...................................................................................... 33
4.5 Framework folder work tree structure example ..................................................... 34
4.6 Interface of put() method ..................................................................................... 34
4.7 Interface of get() method ..................................................................................... 35
4.8 Interface of isAvailable() method ......................................................................... 35
4.9 Interface of request() method ............................................................................... 35
4.10 Interface of hasRequest() method ........................................................................ 36
4.11 Dependency inclusion ........................................................................................... 41
4.12 Global variables example: buffer, mappings and flags ........................................ 42
4.13 Mapping initialization of local module with char type port of size 4 ..................... 42
4.14 Mapping initialization of non-local module with I2C address 248 ......................... 42
4.15 Flag initialization .................................................................................................. 42
4.16 Wire library initialization on address 11 and callback registration ......................... 43
4.17 get() method translation ...................................................................................... 43
4.18 isAvailable() method translation ........................................................................ 43
4.19 hasRequest() method translation ........................................................................ 44
4.20 request() method translation .............................................................................. 44
4.21 put() method translation ...................................................................................... 44
A.1 Example of program definitions ............................................................................. 65
A.2 Example of Arduino programs written by the produced tool ................................. 66
Acronyms

**API**  Application Programming Interface. 12, 25, 28

**CPS**  Cyberphysical Systems. 6

**EEPROM**  Electrically Erasable Programmable Read-Only Memory. 7

**FIFO**  First In First Out. 9

**GUI**  Graphic User Interface. 53

**I2C**  Inter-Integrated Circuit. 28, 32, 37, 41–43, 45, 48, 54

**IDE**  Integrated Development Environment. 6, 7, 45

**IST**  Instituto Superior Técnico. 1, 7, 23, 29, 54

**JSON**  JavaScript Object Notation. 37, 39

**LED**  Light-Emitting Diode. 27, 28, 48

**M2M**  Machine-to-Machine. 17

**OS**  Operating System. 10

**PBL**  Problem-Based-Learning. 1

**PWM**  Pulse Width Modulation. 24, 28, 48

**RAM**  Random Access Memory. 7, 13, 50, 51, 53

**RISC**  Reduced Instruction Set Computers. 5

**ROM**  Read Only Memory. 13

**RTOS**  Real-Time Operating System. 6

**USB**  Universal Serial Bus. 7, 45

**VM**  Virtual Machine. 13

**WSN**  Wireless Sensor Networks. 6

**XML**  eXtend Markup Language. 17, 29, 30, 33, 37, 39, 53, 55
Chapter 1

Introduction

1.1 Context

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. More recently, these have gained some traction in the hobbyist sector, turning them very popular for startups and small personal projects. Nonetheless, the introduction of courses related to embedded and cyberphysical systems in higher education had only gained some traction in more recent years, 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.

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. 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; in the Royal Institute of Technology, students who experienced a PBL approach had higher grades on exercises, project, and exams, higher attendance to classes and more disposition to work extra-hours, than students on a traditional education approach; 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.

Arduino board (and its integrated microcontroller) was one of the embedded development platforms introduced into universities embedded software courses, due to its open-source and low-cost characteristics. In Peter Jamieson experiments, the use of the Arduino board had positive outcome in term of the students’ proposed and delivered projects compared with the previous years used DE2 and...
PIC microcontrollers. He argues that Arduino does not expose programmers to low-level details as it is common on multiple microcontrollers [4]. Authors defend the use of Arduino stating that it has suitable and simple examples to demonstrate general-purpose programming practices [3].

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) [5]. Many authors point out the resource limitations, like low processing power, as one of the challenges on the embedded systems development [6, 7, 8, 9, 10]. Limitations such as small memory, small processor word sizes, slow clock frequency, energy constraints or low-bandwidth are all barriers to software development. Due to the fact 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 like 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 [6, 9, 11, 12, 13]. Some authors agree that programming communication between nodes of an embedded system’s network, and all related reliability and fault-resilience is extremely complex and time expensive. Programmers end up spending more time concerned with technical details than writing real application code. Also, the complexity of networks grows with the complexity of coding these systems [7, 10, 13].

Some authors propose the use of higher level of abstractions to reduce the complexity of developing embedded software [5, 6]. Michael V. Woodward et. al focus on the idea of Model-Based Designs as executable system models, as the basis of the specification, design, implementation, and verification, with the support code generation tools [5]. Edward A. Lee proposes abstractions that yield more understandable programs that are both concurrent and timed [6]. Many authors present, each, a tool or abstraction, in order to improve the experience of development of software for embedded and cyberphysical systems, networks of embedded systems, or to facilitate the interfacing of software and hardware [7, 13, 14, 8, 15, 9, 10].

1.2 Proposal

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 from Information Systems and Computer Engineering courses who usually fit this profile.

The framework we are proposing is composed of three components: a workbench, a set of exercises and a code generation tool.

It was designed around Arduino boards, which were the hardware already at disposal at Instituto
Superior Técnico. These boards were previously evaluated and were found to be the most usable to start working and studying embedded and cyberphysical systems.

The reason why we selected Information Systems and Computer students and engineers as our primary subjects for the framework, was because of the lack of fundamentals and awareness that these individuals commonly have when addressing themes like hardware. Information Systems and Computer engineers may effectively integrate and/or cooperate in projects involving hardware, making this knowledge a must-have skill in their future.

1.2.1 Workbench

Every user of the framework is to be given a set of materials that compose the framework workbench. This set of materials consists of one to three Arduino boards (depending on the exercise to be performed), breadboards and various other pieces of hardware (sensors, actuators, resistors, etc.).

1.2.2 Set of exercises

The set of exercises was designed to engage engineers and students to develop software for hardware, or more specifically embedded and cyberphysical systems. In this set of exercises, we pretend to face individuals to common challenges of cyberphysics such as control of sensors and actuators, communication between nodes in a network, etc.

1.2.3 Code generation tool

With the code generation tool, we are going to ease the development of software by offering a structured definition language where programmers can easily define their programs. Then this tool will be responsible for generating (most) all the code, focusing mainly on producing the communication layer of code necessary for the distributed application to function.

1.2.4 Goals

We pretend the above-mentioned framework to:

- Introduce students and engineers to embedded software;
- Be proper to use in the studies of embedded and cyberphysical systems;
- Accelerate the development of embedded software;
- Allow the development of wide-scale distributed systems with transparent communication to the programmer;
- Most importantly, ease the development of embedded and cyberphysical systems to programmers with greater software skills than hardware skills, as it is the case of Information Systems and Computer Engineering students and engineers.

1.3 Outline

This work is organized as follows: subsection 2.1 presents brief descriptions to help the reader understand some of the concepts presented in the rest of the work; chapter 2.1.2 presents the state of work.
in the field of assisted development of embedded software; chapter 3 describes the architecture of our proposal, while chapter 4 its implementation; finally chapter 5 presents the results achieved by it and chapter 6 encloses with final conclusions and what can be done as future work.
Chapter 2

Background and State of Art

2.1 Background

This section defines some concepts necessary to understand the rest of the work. More precisely it defines what is an embedded system (and wireless sensor networks and cyberphysical systems), and what is Arduino.

2.1.1 Embedded systems

An embedded system is an electrically powered system composed of hardware and software (and sometimes, some additional parts) in order to perform one single and dedicated function, repeatedly [16][17][18]. Some authors, describe these systems as any computing system other than desktop computers and mainframes. The rationale behind is that by the fact that desktop computers allow users to reprogram their use, while an embedded system is programmed once, performing only the desired function [17][19].

Embedded systems can be found in a myriad of devices, such as consumer electronics (cell phones, pagers, digital cameras, camcorders, etc.), home appliances (microwave ovens, answering machines, thermostat, home security, etc) and automobiles (transmission control, cruise control, fuel injection, anti-lock brakes, etc.)

Frank Vahid and Tony Givargis define two main characteristics for every embedded system [18]:

- Single-functioned: embedded systems execute only one program repeatedly.
- Tightly constrained: design and implementation of embedded systems have really tight constraints usually measured in cost, size, performance, and power. Embedded systems generally should cost just a few dollars, must be sized to fit on a single chip, must perform fast enough to process data in real-time, and must consume minimum power to extend battery life.

Peter Marwedel, divide embedded systems into three distinct classes, depending on their size and use-case [17]:

- Small scale embedded systems: are designed around an 8 to 16 bits microcontroller with little hardware and software complexity, usually, battery operated. When developing embedded software for these, an editor, assembler and cross assembler (specific to the microcontroller or processor used) are the main programming tools.
• Medium scale embedded systems: are designed around one to few microcontrollers or DSPs or Reduced Instruction Set Computers (RISC) with 16 to 32 bits, with both hardware and software complexity. Programming tools for these class of systems are: RTOS, Source code engineering tool, Simulator, Debugger and Integrated Development Environment (IDE) (figure 2.1 is an example of an IDE specific for Arduino).

• Sophisticated embedded systems: are the most refined class of embedded systems presenting enormous hardware and software complexity. These are used for cutting-edge applications that need hardware and software co-design and integration in the final system. Examples of this type of systems are automobiles, airplanes, etc.

**Wireless sensor networks**

Wireless Sensor Networks (WSN) consist of multiple sensors connected to each other wirelessly, cooperating to monitor large physical environments [20]. A WSN usually consists of a great number (few tens to thousands) of sensors spread across large areas, sometimes inaccessible, working and communicating among each other, and finally, communicate with a centralized base station, to send their sensor data to remote processing, visualization, analysis, and storage systems.

Due to the characteristics of such networks, sensors do not solely read some information from the environment but have more responsibilities, such as onboard processing, communication, and storage capabilities. These enhancements allow sensors to contribute to the network with in-network analysis, correlation, and fusion of its own sensor data and data from other sensor nodes [20].

Jennifer Yick et al, define two different types of WSN depending on their structures [21]:

• Unstructured WSN contain a large number of nodes deployed in an ad hoc fashion. Once deployed the nodes are left unattended to perform monitoring and reporting functions. Network maintenance, such as managing connectivity and detecting failures, is then difficult to achieve due to a large number of nodes.

• Structured WSN have all or some of the sensor nodes deployed in a pre-planned fashion. This type of structure leverages from having fewer nodes deployed, with lower network maintenance and management costs.

Wireless sensor networks can be found in a multitude of areas, such as military target tracking and surveillance, biomedical health monitoring, hazardous environment exploration, etc.

**Cyberphysical systems**

Cyberphysical Systems (CPS) are systems that intrinsically integrate both physical and cyber (or logical) parts. They do not result from joining a physical system with a logical one, but from integrating both in the creation of a single cyberphysical system [12].

The physical part involves components related to an environment interaction, like sensors or actuators, while the logical part relates the system’s computing part where reside terms like network, processing or programming. The logical part is the one responsible for monitoring, coordination, and control of this type of systems.

Overall, a CPS is described as an interaction between both physical and logical parts through a loop of feedback from physical processes, resulting from the interaction between the former with the real world, and the consequent reaction from the latter over that feedback. [22] [11] [12]
2.1.2 Arduino

In this background, we could have addressed various microcontrollers available in the market, some even cited by other works of reference. Nonetheless, since the beginning of the development of this work Arduino was the only microcontroller to be considered. The reason behind this was the fact that Instituto Superior Técnico had already acquired this microcontroller to teach students developing software for embedded hardware.

Arduino interface board provides a low-cost and low-maintenance, open-source, durable and easy to use and setup (plug-and-play) technology to create microcontroller-based projects [4, 23, 24]. It is a small microcontroller board with a USB plug ready to connect to a computer, and a number of connection sockets that can be wired to external electronics such as motors, relays, light sensors, etc., being Arduino often associated to physical computing.

It was created with the goal of helping teach students and popularized in 2005 by Massimo Banzi and David Cuartielles as a commercial solution [23]. The most popular version of Arduino boards is UNO (fully detailed in appendix A.5).

Versions

UNO is the latest and most popular version of the Arduino family. Nonetheless, there are other with different capabilities and uses. The following are some examples:

- **Mega**: 58 I/O pins, 128KB of flash memory, 8KB of RAM and 4KB of EEPROM [25].
- **Bluetooth**: Instead of the USB connection with the computer, there is a Bluetooth hardware.
- **LilyPad**: small version of Arduino popular in wearables [26].

Programming environment

Arduino development is supported with the official IDE the Arduino Software [27] (available for Windows, macOS, and Linux). In order for users to program with it, the Arduino must be connected to a computer through USB. Beyond programming, the IDE allows users to select the type of board they are using (versions in subsection 2.1.2), check the program for errors, compile and even upload programs into Arduino boards. Once uploaded, the programs immediately start.

Also, Arduino Software has some useful examples built in, ready for use (e.g. how to blink a LED or how to use debounce technique on buttons).

Arduino uses C or C++ as programming languages, which means that in addition to the imperative programming paradigm implicit to C it also allows the use of object-oriented programming of C++. When starting to program with Arduino, users are faced with the mandatory boilerplate code:

- **setup** method: executes once at start time. Useful for objects and libraries initialization, for example.
- **loop** method: executes in a round-robin fashion, topmost to bottommost.

Standard libraries

Arduino Software allows the use of multiple standard libraries. There are libraries for mathematical operations, making random numbers, manipulating bits, detecting pulses on input pins using interrupts, etc.
Custom libraries

Arduino’s users can create their own libraries. These must be placed inside the Libraries directory where the Arduino Software is installed [28]. Every custom library should follow the structure:

- `<name>.h`: the header file;
- `<name>.cpp`: implementation file;
- `<name>.txt`: keyword specification;
- Examples/: folder with examples.

Advanced I/O

Arduino is single threaded, which means there is only one execution flow per program. Nonetheless, it allows variations to the flow of execution using interruptions to run code outside that flow, by attaching interruptions to pins D2 and D3. The use of interruptions is controlled at a software level by the built-in functions `interrupts()` and `noInterrupts()`, to enable and disable interruptions, respectively [29].

2.2 Support on the development of embedded and cyberphysical systems

The following subsections present the state of art of operating systems, languages and tools to help in the development of embedded systems, such as sensor networks and cyberphysical systems. The following subsections are organized as follows: first a brief introduction of the work is given, with the authors mention and their proposals; then, how each work is modeled in terms of tools/abstractions provided to the end-users; finally, it is explained how concerns like communication and concurrency have been being worked, in order to provide end-users the ease-of-use pretended by our work.
2.2.1 TinyOS

TinyOS \cite{9} is an operating system for sensor networks, implemented in NesC language, based on an event-driven architecture, supporting Atmel AT90L-series, Atmel ATmega-series, and Texas Instruments MSP-series processors.

TinyOS exposes programmers to two main abstractions:

- **Components**: the most basic structure in the operating system and abstraction to hardware resources. TinyOS already provides built-in components for sensors, ad-hoc routing, power management, timers, etc. Components are divided into two types:
  - Modules: simple components. In modules, programmers declare the component's behavior with commands and events, and private state variables and buffers.
  - Configurations: seen as super-components due to their ability to aggregate multiple components, and specify how their instances bind to each other forming one single component. TinyOS provides an abstraction called Wiring Specification to connect components defining which components are used among the application. Interfaces can be wired multiple times allowing fan-out distribution throughout components. The top-level configuration describes the full application.

- **Interfaces**: the abstraction for services between components. Programmers should declare in their components the interfaces that they provide and consume. This way, interfaces describe the interaction between components.

Intra-process communication between components is offered with three abstractions:

- **Commands**: non-blocking mechanisms for components’ inter-communication. Components should expose commands as services in their interfaces, for other components to consume them. Consequently, the implementation of the command should reside in the defining component. This type of communication is synchronous and its results are immediately returned.

- **Events**: are non-blocking mechanisms for components’ inter-communication. Components request events from others, that may be signaled asynchronously in the future. Calling components should have event handlers implemented to receive the result of the communication.

- **Tasks**: are asynchronous intra-communication mechanisms. They represent the internal concurrency within components, being the only communication mechanism with access to the components’ internal state. Tasks do not execute at call-time; instead, they are scheduled to run at a later time. Both commands and event handlers can schedule tasks as a result of their execution.

These three mechanisms allow TinyOS to execute real-time operations through commands and events, deferring long-running operations to tasks. TinyOS non-preemptive \texttt{FIFO} scheduler is responsible for collecting scheduled tasks and compute them to completion.

Concurrency is handled in TinyOS by enforcing Race-Free Invariant, which means that: ”any update to shared state is either SC-only or occurs in an atomic section”. Synchronous code (SC) is code reachable from tasks, while asynchronous code (AC) is code reachable from at least one interrupt handler. Since the scheduler runs tasks to completion in a non-preemptive fashion, this mechanism already ensures safe data-races. Then, programs implemented using tasks as the only communication mechanism, comply with the Race-Free Invariant.
However, there may be races between AC and AC or AC and SC. In those cases, updates on shared state must be performed using atomic sections, which ensure that NesC code runs atomically, hence complying with the Race-Free Invariant. To ease the programming of such scenarios, NesC compiler already validates the invariant at compile time ensuring that almost all concurrency is covered by it.

Communication between nodes in a network using TinyOS is achieved using Active Messages (AM). AM offer a unified interface for wired and wireless communication and consist of unreliable, single-hop datagram protocol messages with 36 bytes. Nodes are notified of AM arrival by registering handlers specific types of messages. When AM arrive, the node notifies every handler registered through an event.

At compile time, TinyOS is optimized to include only the required OS services, remove dead code and redundant operations, and minimize component crossings. These optimizations make it reduce simple applications size for around 60%.

2.2.2 TinyGALS

TinyGALS [13] is a “globally asynchronous and locally synchronous approach” for programming event-driven embedded systems.

Elaine Cheong et al. used three abstractions to model their tool:

- **Components**: the basic elements of TinyGALS and are composed of:
  - sets of internal and external variables;
  - sets of methods: methods have access to components’ variables that operate on these variables. Components definitions must have an interface for the methods. The implementation should only reside in the components that provide the method. Components can then consume methods from others. Local communication between components is performed synchronously.

- **Modules**: structures bigger than components composed of:
  - sets of components;
  - sets of initialization methods (methods belonging to the components);
  - sets of input and output ports;
  - sets of variables external to the components.

- **Systems**: the abstraction that defines the full system. These are composed of:
  - sets of modules;
  - sets of global variables and their mappings with modules’ external variables;
  - sets of connections between module output ports and input ports;
  - the name of an input port of one module for the starting point of execution.

Intra-process or intra-components communication is abstracted using Links. They bind together components’ methods’ interfaces to each other or to modules’ inputs or outputs, creating a flow of execution inside modules. This execution flow created by Links is performed synchronously.
Inter-process, or between modules communication is performed asynchronously using Connections. They use FIFO queues to store the messages received, allowing both sender and receiver to continue execution. Messages passed from one module’s output port to another are placed as tokens in the latter’s queue. Sometime later, TinyGALS’ runtime scheduler removes those tokens passing them to the component bound to the module’s input port (through a Link).

To guarantee safe write from modules on system global variables Elaine Cheong et al. created TinyGUYS (Guarded Yet Synchronous) as a mechanism for sharing global state. With TinyGUYS, modules may read global variables synchronously but write only in an asynchronous fashion using a buffer of size one where the last module to change it prevails. TinyGUYS are then mapped to modules’ parameters that are further mapped to components external variables.

Elaine Cheong et al. created a tool for code synthesis, for Berkley motes. Given a valid definition of a system, modules, and components, the code synthesis is performed in multiple steps:

- **Application’s start point:** the code generator creates the `app_start()` function based on the specified name of the input port defined as the starting point.

- **System initialization:** the code generator creates a system-level function `app_init()`, containing all modules’ initialization methods, which will be called by the TinyGALS scheduler before running the application. For this process, the order of declaration of modules in the system definition is taken into account, since it determines the order in which initialization methods are called.

- **Links and Connections:** created through a set of aliases that map both links between components and connections between modules. For each Connection between modules, it is created a set of scheduler data structures and functions:
  
  - For each input port, a queue is created, with its length specified in the system definition (input ports with parameterless functions have no queue associated for optimization), a pointer to keep track of the queue head and a counter for the number of tokens.
  
  - For each output port, the generator creates a function with the output port name, that will be called every time values are written in the port. Communication between input and output ports is performed through the `put()` function generated at the input port. `put()` is responsible for placing the arguments in the input port queue and notifying the event scheduler with the correct module identifier in order to trigger it in the future.

- **Global variables:** a pair of data structures (data storage and a flag) and a pair of functions (for writing and read access) are generated. The writing function writes values to the location storage, while the flag value indicates whether the scheduler needs to update the global variable with the value in the location storage.

### 2.2.3 Flask

Flask [7] is a programming language for sensor networks focused on providing an easy-to-use dataflow programming model. Flask is interpreted at compile time to generate NesC programs which are then compiled into binary. It allows programmers to focus on the behavioral part of the application rather than on details of routing code.

Flask goals are to provide abstractions to deal with low-level details like communication or concurrency, provide a programmatic wiring language focused on flexibility for construction of new abstractions,
and permit compilation to optimized node binary and integration with existing network operating systems.

Geoffrey Mainland et al. exposes two abstractions for the solution model:

- **Dataflow operators**: structures that may receive or produce values, which are single typed primitive variables, or tuples of primitive values. These structures can only have one output, but multiple inputs (i.e., receive information from multiple dataflow operators).

- **Dataflow graphs**: sets of dataflows operators combined together using wires, creating the program execution flow. Graphs are executed as depth-first and any given traversal ends when it reaches an asynchronous request (posting wire) or an operator that does not produce any output. The set of dataflow operators in a traversal is called an atomic subgraph.

Intra-process communication, and hence between dataflow operators, are described using two intra-communication abstractions:

- **Wire**: connect dataflow operators synchronously passing information between them.

- **Posting wire**: asynchronously connect dataflow operators. It depends on the TinyOS event mechanism to post messages for future handling.

Inter-process communication is provided through one single abstraction, Flows. They allow dataflow graphs to be distributed across a sensor network using different communication topologies (point-to-point, multipoint-to-point, point-to-multipoint, and multipoint-to-multipoint). The structure representing flows is identified by a 16-bit identifier, and support a publish/subscribe API to receive/send data.

Programmers are not exposed to routing, maintenance and buffering details being all abstracted by the flow itself. Publishing and subscribing to a flow is accomplished by wiring the appropriate operator into a program’s dataflow graph.

The novelty in Flask is its unique programmatic Wiring Language, allowing dataflow graphs to be described by a program instead of a static representation. Wiring programs are based on:

- **Streams**: an abstract source of typed values representing a dataflow subgraph with a single output wire;

- **Stream Combinators**: expressions that generate and manipulate streams. Runtime code for each operator in the dataflow is implemented in NesC.

This gives programmers the ability to use and interface with existing NesC and TinyOS components. Geoffrey Mainland et al. provided both a mechanism to program inline NesC functions, making easier to write dataflow operators directly in the wiring program, and a suite of built-in operators that provide commonly-used functionality such as timers, sensor data acquisition, and radio communication.

Flask results were gathered regarding:

- **Ease-of-use**: measured in terms of lines of code, and compared with two other tools for sensor network development. Flask solution had around 10000 fewer lines of code.

- **Communication overhead**: tests against one single tool in a well-defined environment. Despite both Flask and the comparing solution had delivered the same number of packets per time with success, Flask floods the network with more packets causing a greater overhead.
• Memory size: comparing with two other tools. In this metric, Flask presents one of its best advantages. Despite generating more code related to communication than any other tool, the lack of code generated for application purpose makes it use half of ROM than the comparing tools. In terms of RAM, Flask ended up using as much memory as the other tools.

2.2.4 Maté

Maté [10] is a communication-centric virtual machine for sensor networks, based on the idea of transferrable capsules of instructions, in order to reprogram networks already deployed, in an ad-hoc fashion. It was designed to run on top of mica and rene2 motes. It consists of a byte-code interpreter running on TinyOS, as an application component.

Philip Levis et al. created the Capsules abstraction: these contain instructions that will run on Maté VM. Each capsule contains a version (of the software, or instructions) and identifying information, and, at most, 24 instructions with one byte each (to fit into TinyOS packets, making receptions atomic and hence, conserving RAM). Capsules can be seen as full programs. Therefore, one program may consist of multiple capsules. Capsules are divided into four types:

• message send and message receive: run on a virtual machine’s context where the operands stack is non-persistent across invocations because, for these capsules, the context expects to find a message on top of the operands stack. Regarding operands, they are divided into values, sensor readings and messages, being some instructions only able to operate on certain types.

• clock timer: runs on a virtual machine’s context where the operands stack persists across executions to keep information available for future executions. Hence, when clock capsules are installed, value 0 is placed on top of the operand stack, as a start value.

• subroutine: are programs usually consisting on more than one capsule. These are triggered and returned by call and return instructions, respectively.

Maté VM’s contexts of execution (one per capsule type) are composed of the operands stack (16 bytes long), and a return address stack (8-bytes long), for subroutine calls only. The instructions set is divided into three classes:

• basic: instructions’ byte is represented as 00iiiiii, where iiiiii is the instruction’s identifier, and comprehend operations such as arithmetic, halting and simple hardware control. Arithmetic instructions are parameterless because Maté makes use of an operands stack, where operands reside and where instructions operate on.

• s-class: instructions’ byte is represented as 01iiixxx, where iii is the instruction’s identifier and xxx is an argument. This class of instructions are only executed on send message arrival and departure contexts, to read from in-memory structures such as message headers.

• x-class: instructions’ byte is represented as 1ixxxxxx, where i is the instruction identifier and xs are the argument. x-class consist of only two instructions:
  – pushc: which pushes constants;
  – and blez: to branch on a less than or equal to zero condition.

The novelty presented on Maté is its “infection network”. The use of capsules allows programs with new behavior to be deployed and installed on-the-fly without the need of recompile and reassemble the
network. Every time a node in the network receives a capsule, this one's version is checked, and if it is a more recent version than the one already installed, the new capsule is installed.

Network infection is performed by all nodes in the network, in a "virus propagation" fashion. Once a capsule runs to completion, it is forwarded to neighbor nodes, being only necessary to install one capsule once in a single node, to propagate it across all network. Because of this approach, if the programmer does not desire to upgrade or install capsules in certain nodes, these must be programmed as part of a different network.

Since versions are identified by 32 bits, it is possible for a network to be reprogrammed for very long periods of time.

Data-race prevention is guaranteed, in an inter-context level, because each one owns their own operand and return address stacks. Therefore, no memory is shared among contexts. Intra-context data-race does not need to be prevented because every Capsule runs to completion in a synchronous fashion.

To enable shared memory between contexts, Maté implements a one-word heap available to all contexts, allowing communication using the instructions \texttt{sets} and \texttt{gets}.

In order to evaluate Maté, Philip Levis et al. reimplemented BLESS, an ad-hoc routing protocol, already implemented in TinyOS, with 600 lines of code. They labeled their instructions based on their complexity and level of encapsulation of TinyOS constructs:

- simple: for instructions that merely operate on the operands stack;
- downfall: for instructions that call commands on other TinyOS components;
- quick split: for instructions that use TinyOS events asynchronism;
- long split: the same as quick split, but with more computation cost.

When compared with native code, simple and downfall instructions had a higher cost on clock cycles, revealing overheads of 33.5 to 1, and 9.5 to 1, respectively. Instructions that encapsulate behavior, as split instructions, revealed low or no overhead.

When compared with the native version, the overhead caused by the instructions translates into a more energy consumption. However, when comparing energy spent on install time, Maté overcomes its binary version, because packet (or Capsules) transfers are less energy intensive than binary upload. Considering execution and installation, Maté is only preferable for small programs, because it is when energy spared on installation balances energy spent on execution.

### 2.2.5 Agilla

Agilla \cite{fok2006agilla} is a middleware for increased network flexibility and ease of application development built on top of TinyOS for MICA2 motes. It is a mobile agent-based middleware where each agent consists of a virtual machine with dedicated instruction set and data memory.

Chien-Liang Fok et al. used one abstraction to represent their applications’ model. Agents are autonomous structures scattered throughout the network, with different types, and the source of all the application behavior. Agents have the ability to migrate across nodes and coexist with multiple other agents in a single node. Agents’ life cycle starts when they are uploaded into the network or cloned by other agents. These structures execute as autonomous virtual machines containing their own
instructions memory, data memory, program counter, operand stack, and heap. Agents’ life cycle ends when their task is complete, using halt operation, in order to free resources.

Inter and intra-process communication are supported by Tuple Space abstraction. Tuple Spaces are the means of coordination and communication between agents. They are a shared memory model, where datum is a tuple representing an ordered set of fields. Each field has a type (integer, string, location or sensor reading) and a value. Tuples are accessed through other unique template ordered set of fields.

Since the Tuple Spaces live inside the nodes and not inside agents, it creates a high level of decoupling, promotes agents autonomy and provides agents with contextual information.

- **Intra-process communication**: local communication between agents and the tuple space is locally offered as a publish-subscribe service. Agents interested in a given tuple template, register it in the tuple, and every time a message matches the template, the interested agent gets notified. This process is supported by the following tuple operations:
  - `out` to insert a tuple;
  - `in` and `rd` to remove and copy from the tuple space in a blocking fashion;
  - `inp` and `rdp`, which are the `in` and `rd` operations in a non-blocking fashion.

- **Inter-process communication**: to support remote communication and coordination between agents, three more tuple space’s operations were included: `rout`, `rinp` and `rrdp` are synonyms of `out`, `inp` and `rdp`. These take an additional parameter with the location of the interested node. Communication using remote operations relies on unicast communication between the sender and receiver’s tuple spaces.

- **Neighbor lists**: provide nodes with the list of all one-hop neighbors found using beacons. Agents can consult the neighbor list using `numnbr` and `getnbr`. Multi-hop operations are completely transparent to the user. Nodes are identified using their physical location instead of a numerical identification. Chien-Liang Fok et al. use this notation, not only because location enriches wireless sensor networks context, but because it is also used to create primitives to enable in-region operations.

The novelty in Agilla is the components’ migration and cloning features.

- **Migration**: the code moves from one node to another;
- **Cloning**: the code is cloned to another node in the network.

These features represent the way that Agilla transfers agents from node to node in the network.

Migration and cloning operations process in strong and weak modes:

- **Strong mode**: both state and code are transferred or cloned, resuming execution;
- **Weak mode**: only code is transferred or cloned, starting execution from the beginning.

Agilla architecture is divided into multiple components:

- **Agent Manager**: responsible for managing agents’ (typically, around 4) life cycle memory allocation, and notify Agilla Engine about their readiness;
- **Context Manager**: keeps track of the node and one-hop neighbors’ locations;
• Instruction Manager: provides the missing dynamic memory allocation of TinyOS. It is responsible for agents code memory allocation on time of arrival, fetching instructions in runtime and packaging the code on agent departure;

• Tuple Space Manager: implements the non-blocking operations, and it is responsible for the asynchronous communication and memory allocation on tuples.

• Agilla Engine: serves as the virtual machine kernel, handling concurrent execution in a round-robin scheduling fashion. Every agent executes a fixed number of instructions before switching context unless a long-running instruction (like \texttt{sleep}) is executed. Agilla Engine is also responsible for reception/emission and unmarshaling/marshaling of agents on arrival and departure. Agents are migrated one hop at a time, with four retransmission attempts, acknowledge messages, and a fallback system. Remote tuple space operations are also Agilla Engine’s responsibility. The result from such operations is sent back, with two retransmission attempts and no acknowledge messages.

Chien-Liang Fok et al. tested their solution using a 5x5 grid of MICA2 motes in an environment simulation. Their main focus was on performance and reliability throughout the network. With a simple agent with two tasks: migration, and remote insertion in tuple spaces, it came forth that from 1 to 5 hops, the success of migration dropped from 100% to around 90%, while remote insertion dropped from 95% to around 70%.

Also, regarding the same operations and hops variation, latency on migration went from around 200 milliseconds to more than 1 second, while remote insertion went from around 50 milliseconds to around 300 milliseconds.

Regarding latency on remote instructions, migration and cloning had higher overheads and higher variance, explained by the uncertainty on package losses. On local operations, general purpose instructions seemed to have a latency of at most 150 milliseconds, while tuple spaces instructions never had less than 200 milliseconds.

2.2.6 Splish

Splish \[15\] is a visual programming environment designed for ease-of-development of cyberphysical systems, for Arduino based microcontrollers. It aims to help non-specialists to quickly develop cyberphysical applications. Splish emphasizes delivering an icon-based visual programming environment, Arduino I/O support, enabling monitoring and debugging using a PC, and offering portability and extensibility.

Splish programs are programmed using two visual abstractions:

• Icons: represent the variables or flow controls of one program, and are the basic structures to build applications. Some built-in icons are already provided with Splish (e.g. an icon to control the Arduino board built-in LED). Splish programs must always have two special icons: entry icon (the start point of a program) and exit icon (the end point of a program).

• Connectors: structures that connect icons, defining the program flow.

Splish programs are compiled into a stack-based virtual machine bytecode (designed specifically for Splish purposes) bytecode instead of Arduino machine language. The virtual machine instructions can weight a single byte to multiple bytes. The instruction set only supports arithmetic, logical and I/O operations. The virtual machine is divided into two structures:

• Memory: contains all the running instructions and the program counter pointer.
- Stack: includes stacked variables and operands, the stack pointer and frame pointer.

Splish graphical user interface provides multiple features beyond model abstractions:

- Compile: allows compilation of programs and visual feedback of the process through a status panel.
- Upload: allows transfer of compiled programs to Arduino boards between computer and board (through serial communication).
- Debug and Monitoring: are supported by an Arduino written program uploaded with the real application program. For monitoring purposes, the application runs step-by-step or automatically, being the microcontroller status sampled during execution or upon user request. In debug mode, code execution is controlled by the user step-by-step.

### 2.2.7 WuKong

WuKong [14] is a flexible middleware support, for developers and users of Machine-to-Machine (M2M) systems to abstract the development and deployment of sensors, using high-level primitives instead of node-centric programming, implemented for Arduino. It tries to achieve transparent sensor detection, device selection, system configuration, and software deployment.

Niels Reijers et al. created three frameworks to support their middleware:

- Sensor profile framework: defines a logical abstraction of sensor capabilities for heterogeneous nodes. It allows WuKong to track the physical resources all over the network and to manage the communication between them;
- User policy framework: defines objectives and context management;
- System progression framework: supports software upgrade in situ.

WuKong uses a master-slave architecture where the master is the gateway-node responsible for:

- interfacing with the outside world;
- managing decision making and deployment;
- configuring the sensor nodes.

Each node, besides the master node, is uploaded with a sensor profile, which makes it possible to run WuKong M2M applications, by deploying all the application to the master node. The latter is responsible for discovering every node, in an initial phase, access and configure them using the uploaded sensor profile.

The development of WuKong applications is made using a flow-based programming (FBP). This allows programmers to focus on an abstract flow of information, while the FBP program generates the low-level details for physically implementing the flow. The FBP is developed in an IDE and is exported as XML. WuKong’s FBP uses two abstractions for development of M2M systems:

- Components: the behavioral part of the application, and will be realized by a physical sensor or a piece of running software. They expose external interfaces, using a set of properties, and can only be connected with components with matching properties. WuKong offers a library of predefined components, for commonly-used sensor hardware and functional elements; for more experienced developers, it is possible to add custom components, containing application-specific functions.
• **Links:** abstractions used to connect components, in order to create an execution flow. They can consist of:
  - a set of conversions from one unit to another;
  - a set of filters to propagate only useful values;
  - a set of reliability constraints;
  - and custom code to extend predefined behavior.

While developing components, the programmers are presented with WuClasses, which are structures like object-oriented classes, with their own properties, implementations of processing, which only expose a single method: `update()`. It implements the class behavior, and it is called whenever a property is changed or a sampling rate interval is set.

WuObjects are the main units of processing, hosted in each node, and represent instances of WuClasses. These are divided into three different groups depending on their purpose:

• Hardware access: related to hardware components and are implemented in C. At startup, for every piece of hardware, it is instantiated a WuObject per WuClass.

• Common processing components: logic behavior implemented in C or Java. At startup, no WuObjects are instantiated. Instead, the master node uses the sensor profile to create instances when the application is deployed.

• Application specific processing: WuClasses not included in the standard library, implemented in Java.

The sensor profile framework is responsible for:

• allowing the master node to find which WuClasses and WuObjects are available in every node;

• load WuClasses and instantiate WuObjects implemented in Java;

• trigger the execution of WuObjects;

• and, propagate changes between linked properties of WuObject both locally or remotely.

Inter and intra-process communication is controlled and managed by the sensor profile. When one change is detected on one component, `update()` is triggered to update its properties. If those properties are connected to another component property, the sensor profile framework propagates the new value to the linked component. Communication between local and remote nodes is thus transparent.

### 2.3 Conclusion

Regarding the previous subsections, we conclude that different approaches were used until now, in order to tackle problems faced in the development of embedded and cyberphysical systems and sensor networks. Some of the studies rely on the use of operating systems to address low level details [9], others on high-level abstractions [7] [13] [15] [14] [8], some on instruction virtualization with use of virtual machines [8] [15] [10].

Each study tries to tackle different difficulties faced when developing these type of systems. Most all tools, focus on creating transparent communication and remove the effort of detailing all aspects of it from developers [9] [13] [7] [14] [10] [8]. Communication is a common characteristic of embedded systems and a crucial aspect of sensor networks.
Others focus on delivering safe concurrent code \cite{9,13,7,10,8}. Most of them have support for interruptions or implement schedulers for concurrency. Also, in order to support concurrency, some tools implement shared state between operations.

Solutions using high-level abstractions offer code synthesis, usually to transform programs into already used programming languages so they can be compiled into binary or be executed in a virtual machines \cite{13,7,14,15}.

Some solutions offer standard libraries with common modules and/or controllers for ease-of-use, allowing extendibility by experienced developers \cite{9,7,14}.

Despite Splish \cite{15} being the simpler tool, it is the only one focused on debugging and monitoring cyberphysical applications, revealing that efforts are mostly turned to ease-of-development on code production time, and not on testing time.

Finally, regarding the platform supported by each solution, most of them were designed for motes of the Berkeley family, revealing its relevant use on education and investigation \cite{9,13,7,8,10}. However, some systems already support Arduino, proving the increasing relevance of this board \cite{15,14}.

TinyGALS has the most well-defined abstractions for what we envisioned to be an embedded distributed application. The use of the three user-level abstractions component, module, and system, allow users to clearly define their programs in different levels of granularity.

Flask, TinyGALS, and Wukong present inspiring ideas on how to handle intra and interprocess communication with their user-level abstractions and internal mechanisms. The three clearly distinguish and allow how to define local communication in a single microcontroller as they do between microcontrollers. This is an intended feature in our code generation tool and the designs used in these works are a major influence.

The use of built-in functionality is another influence to what we envisioned to be our code generation tool. In this area, TinyOS, Splish, Flask, and Wokung are the references among the studied works.

Despite the features present in some of the works matching what is intended for what we expect to be our code generation tool, only Wukong and Splish support the use of Arduino, the microcontroller for which we designed our entire framework. Splish is a too basic tool, only supporting the creation of programs with one board and no support for communication between boards.

Wukong \cite{14} is the closest tool to what was desired, already complying with transparent communication between motes, code synthesis from high-level abstractions, use of Arduino and provision of built-in libraries. However, the desired solution for this work must have some requisites not complied by Wukong:

- **It should allow users to design all types of network topologies**, not depending on a Master-Slave architecture imposed by Wukong;

- **It should not depend on a virtual machine for execution** (Arduino has a microcontroller with low memory resources and thus, having a virtual machine would occupy part of the memory that should have the program itself);

- Extra functionality should be **developed in native code**, to keep users closer to the microcontroller environment. Wukong extra behavior is implemented in Java and not in C++ which is Arduino native language;

- **Communication related code should be validated in translation time**, preventing errors such as missing nodes in the design. Wukong lacks such mechanisms.
2.3.1 Features table

The following table (2.1) summarizes the features presented in every work analyzed in the state of art. The relevant features found were:

- Model abstractions;
- Local communication abstractions (inside the same microcontroller);
- Remote communication abstractions (between microcontrollers);
- Concurrency mechanisms;
- Code optimization;
- Code validation;
- Support for debug;
- Built-in functionality;

Also included are the main focus of the work, and the microcontrollers supported.
<table>
<thead>
<tr>
<th></th>
<th>Model abstractions</th>
<th>Local communication abstractions</th>
<th>Remote communication abstractions</th>
<th>Concurrency mechanisms</th>
<th>Code optimization</th>
<th>Code validation</th>
<th>Support for debug</th>
<th>Built-in functionality</th>
<th>Focus</th>
<th>Microcontroller</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyOS</td>
<td>Components / Interfaces</td>
<td>Commands, Events / Tasks</td>
<td>Active messages</td>
<td>Preemptive tasks / atomic sections / Events</td>
<td>Code removal and optimization</td>
<td>Validates concurrency at compile time</td>
<td>No</td>
<td>Sensors, routing, timers, etc.</td>
<td>Sensor networks</td>
<td>AT90L, ATMega series</td>
</tr>
<tr>
<td>Splish</td>
<td>Icons / Connectors</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>I/O icons</td>
<td>Cyberphysical systems</td>
<td>Arduino</td>
<td></td>
</tr>
<tr>
<td>Flask</td>
<td>Dataflow operators / graphs</td>
<td>Wires</td>
<td>Flows</td>
<td>TinyOS events / graphs</td>
<td>N/A</td>
<td>N/A</td>
<td>No</td>
<td>Communicate sensors, timers, etc.</td>
<td>Sensor networks</td>
<td>Chipcon CC2420 Radio, TMote Sky / MICAz</td>
</tr>
<tr>
<td>TinyGALS</td>
<td>Components, Modules / Systems</td>
<td>Links</td>
<td>Connections</td>
<td>TinyGUYS</td>
<td>Code reduction during synthesis</td>
<td>N/A</td>
<td>No</td>
<td>N/A</td>
<td>Event-driven embedded systems</td>
<td>MICA</td>
</tr>
<tr>
<td>Maté</td>
<td>N/A</td>
<td>set/get methods</td>
<td>Capsules</td>
<td>Execution context isolation / One-word heap</td>
<td>N/A</td>
<td>N/A</td>
<td>No</td>
<td>N/A</td>
<td>Sensor networks</td>
<td>MICA / rene2</td>
</tr>
<tr>
<td>Wukong</td>
<td>Components</td>
<td>Links</td>
<td>Links</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>No</td>
<td>Built-in components</td>
<td>Machine-to-machine systems</td>
<td>Arduino</td>
</tr>
<tr>
<td>Agilla</td>
<td>Agents</td>
<td>Tuple spaces</td>
<td>Tuple Spaces, Neighbor lists / Migration and Cloning</td>
<td>TinyOS events</td>
<td>N/A</td>
<td>N/A</td>
<td>No</td>
<td>N/A</td>
<td>Sensor networks</td>
<td>MICA2</td>
</tr>
</tbody>
</table>
Chapter 3

Architecture

In this work, we propose an embedded and cyberphysical systems learning framework for students with a lack of software for embedded hardware skills, as it is the case of Information Systems and Computer engineers and students. In this chapter, we are going to present the main guidelines for our framework and explain some of the decisions taken.

3.1 Overview

Our framework proposal started with the development of a workbench for the Embedded Systems Applications course of the Information Systems and Computer Engineering master's degree. We envisioned a laboratory where students or groups of students would have a set of materials to work with, that would allow them to build or learn how to build embedded and cyberphysical systems. The distribution of such materials raised the question of what would students do with it. We answered that question with the first solution of our framework, a set of exercises to teach students some basics about cyberphysics and embedded systems. We prepared four exercises: three introductory exercises for students to keep in touch with these systems, and a real-life problem for students to put in practice what they learned in the first three exercises. At last, due to the investigation made for this work, we decided that it would be proper, especially for students with a lack of software for embedded hardware skills, to create a tool to ease and accelerate the development of embedded and cyberphysical systems. Then, we have created the second solution for our framework: a code generation tool to help students in solving the exercises.

3.2 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. Given the constraints of this work, the workbenches will be composed of Arduino Starter Kits, which are currently available at IST. These kits possess each, an Arduino board with a microcontroller and I/O interfaces and a collection of hardware (LEDs, buttons, sensors, etc.). The reason we pretend to give students three Arduino boards instead of only one is to allow them to create distributed embedded systems.
3.3 Set of Exercises

The set of exercises proposed for the framework has as the main goal to help students to progressively learn the concepts behind the development of embedded systems and to put them in contact with the selected microcontroller. The way we envisioned these exercises was to progressively introduce students to the basic concepts of cyberphysical systems in a hands-on series, where students could build systems and learn at the same time, being finally tested in a real-life problem. 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. In this exercise, students should start using actuators as part of the systems and learn concepts related to the assembly of such systems (I/O, resistors, etc.). The actuators should be controlled via software. Also, this is the first time that students must turn an electrical scheme into a real hardware system.

- Exercise 2: should challenge students to use sensors in their assemblies. Also, students should keep using actuators, but this time, these must be controlled via sensor inputs. In this exercise, students should understand the differences between analog and digital I/O, and concepts such as analog measures calibration, [PWM] etc.

- Exercise 3: puts students in contact with inter-boards communication. In this exercise students should use an available communication protocol for their microcontroller, to create a distributed system. As in the previous exercise, students should control actuators using sensors inputs, but this time physically separate the sensors from the actuators using two boards. The goal of this exercise is for students to understand how to use, at least, one communication protocol for embedded systems.

- Exercise 4 / Project: is a real-life problem. In this last exercise, or project, students should be challenged with a real-life scenario (e.g., building an automated crossroad traffic-light system, building an automated vehicle toll with coins and car-presence detection, etc.). The goal of this project is to apply all the previous concepts learned, using at least three microcontrollers, in order to give emphasis to communication.

3.4 Code Generation tool

The Code Generation tool is a tool for students with a lack of affinity with software for embedded hardware 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. The Code Generation tool is composed of four components: Definition Language, Framework, Communication Library, and Translator.

3.4.1 Definition Language

The Definition Language of code generation tool is the most important part of it in terms of the usability since it is the part of the tool exposed to users. The goal of the definition language proposed for the code generation tool is to abstract low-level details of programming in embedded software. The definition language presents three main abstractions, agents, modules, and links:

- Agents: define a physical bounded system (i.e., a single program that runs on a single microcontroller). Agents execute an entire program in an infinite round-robin loop;
• Modules: define the most basic unit of behavior. Implement a set of self-contained instructions. One or multiple modules may constitute an agent;

• Links: define the connection between agents. This is the abstraction that allows the communication between agents and thus, the creation of distributed applications.

3.4.2 Framework

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

3.4.3 Communication Library

Communication is one of the most important parts of the tool since it should be possible to easily create distributed applications. 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. Part of this API are functions related to message-passing between agents and/or modules, and communication-synchronization between agents.

3.4.4 Translator

The translator is an important component of the code generation tool. Is the piece of software that actually generates code to the target microcontroller. It consists of a composition of two processes that transform our proposed Definition Language into the target microcontroller’s native programming language:

• Parse 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 transformed into actual code (i.e., the target microcontroller’s native programming language).
Chapter 4

Implementation

In this chapter, we intend to detail the implementation of all the proposed components of the framework (chapter 3: the Workbench, the Set of Exercises and the Definition Language).

4.1 Workbench

The Workbench is composed of one-to-three Arduino Starter Kits 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. Appendix A.6 presents all the included in one Arduino Starter Kit.

4.2 Set of exercises

The exercises proposed for the framework are divided into four works to be developed by single developers or teams of developers. Every exercise is intended to address one or more characteristics of embedded software or cyberphysical systems. We designed three simple exercises related to each other that will progressively teach users the basics of embedded software and prepare them to a final project, the fourth exercise. All the exercises were designed for the Arduino board.

4.2.1 Exercise 1

The first exercise tests the ability for the development of a simple cyberphysical system. In this exercise users of the framework should be able to assemble a circuit with four LEDs and produce the necessary software to cycle between on and off LEDs. This exercise allows users to have their first contact with cyberphysical systems by using a type of actuator, the LED. The difficulty of assembling and programming effort for this exercise is supposed to be low because of the fact that it was designed for engineers and students with low or no level of expertise in the development of embedded and cyberphysical systems. Programmers should use only one Arduino board.

4.2.2 Exercise 2

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: a dimmer, and light, presence and temperature sensors. This allows users to...
address more cyberphysical concerns such as sensors and their physical characteristics and techniques such as Pulse Width Modulation (PWM) native to Arduino boards. Users can start the assembly of the system from the previous exercise. Programmers should develop this system using only one Arduino board.

### 4.2.3 Exercise 3

The third exercise challenges programmers with communication between boards, to achieve distributed embedded applications. Once again, this exercise can be extended from the previous one. The ultimate goal of this 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 to work, users should use the Inter-Integrated Circuit (I2C) wired protocol to make boards communicate with each other. This exercise intends not only to challenge users with communication but to teach them a specific communication protocol.

### 4.2.4 Project

We designed a project that tests all the learned subjects in a real-life problem. In this project users of the framework should build a traffic-light system. The traffic-light system should be a simulation of a crossroad controlled by two traffic-lights with three colors for vehicles (green, yellow and red), two colors for pedestrians (green and red), and a button to signal pedestrians presence, for each traffic-light. The crossroad traffic-lights should work like real-life ones. This project has a particularity: the traffic-lights should not communicate with each other, but instead with a mediator that should coordinate the crossroad. The reason we designed the exercise like this was not only to make the project more challenging but also to turn the system more scalable. The traffic-lights should have an extra feature: when communication loss is detected between a traffic-light and the mediator the former should start blinking yellow; then the mediator should instruct the other traffic-light to do the same. Each traffic-light should be designed in a single Arduino board, as it should the mediator, being necessary three Arduino boards to complete this project.

In the preparation of every exercise, we developed an assignment paper. Every paper has:

- **Goal**: presents the ultimate objective for the execution of the exercise;

- **ETA**: is an average time for completion of the exercise;

- **Narrative/Description**: is the introduction of the problem/exercise. It states with detail how the assignment is to be developed;

- **References**: are links to media that might be helpful during the execution of the exercises (e.g. tutorials, APIs, etc.);

- **Recommendations**: depict a list of behaviors that users must have while executing the exercises so that every exercise is performed safely;

- **Theory (optional)**: may be introduced in the papers in order to explain some concepts that must be understood prior to execution of an exercise;

- **Assembly scheme**: is the electrical diagram for the exercise. It is one way of assembly. Users are allowed not to follow the exact scheme as long as the system works as expected;
• Question and Answer area: is the place where lecturers have the opportunity to make some questions as part of the exercises. This area is left in blank so that lecturers can decide which type of questions are more relevant for the class.

Appendix [A.4] presents the assignment papers used in IST for the Applications for Embedded Systems course lectured in 16/17 and 17/18, designed as part of this framework.

4.3 Code Generation tool

The Code Generation tool is composed of four components:

• Definition Language: used by the framework users to write their projects. This definition language allows embedded software or even general purpose software entry-level users to start developing programs with almost no understanding of programming languages. It was designed using XML and human-readable tags with explicit meaning.

• Framework: define a set of directories to organize programs and finally produce compilable deliverables for Arduino boards.

• Communication Manager library: is the set of functions and structures, bundled in a C++ library, to help users clearly define communication intra and inter-process. It hides all communication logic.

• Translator: is a piece of software developed to translate programs developed using this tool Definition Language into C++ compilable programs for Arduino boards.

4.3.1 Definition Language

The three abstractions provided by the tool – modules, agents and links (subsection 4.3.1) – are used by users to compose not programs, but definitions of Arduino distributed or simple applications.

• Modules: represent the basic units of behavior.

• Agents: composed of modules, representing processing units that will be translated, compiled and uploaded into Arduino boards.

• Links: describe connections between agents to create systems as networks of agents.

All abstractions are described using XML because it is easy to learn and well-known markup language in the industry, most often known by engineering graduates. Also, there are numerous XML manipulation libraries in the computer science community, which eases the development of related tools.
Module

Modules are the most basic components of the design. They describe parts of behavior and the flows of communication in an application (listing 4.1). Modules can be seen as methods in a C program because they run few lines of code in their own scope. Altogether they form the program, most specifically an agent (described in more detail in subsection 4.3.1).

Name: is one of the markup attributes that define modules. Its purpose is to identify a module in the entire application. Modules’ names are unique not only in their agent but in all the application. The goal is to reduce redundancy when defining programs, but also to ensure no repetition when producing the first addressing level of the application’s addressing table (subsection 4.3.3).

Function: the implementation of modules in XML excludes code from its definition since Arduino code must be produced in C or C++ programming languages. Therefore, another markup attribute of modules is function. Functions allow users to include snippets of runnable code in Arduino in the module definition. Those snippets are imported using only their name and package in the form of <package>::<name>.

There are only two sources of packaging in the tool:

- Standard package or std: ported with the tool offers the users a library of basic functions such as mathematics instructions and basic hardware control.
- Custom package or cst: way of introducing new features in user applications; users can create their own functions in C or C++ extending the tool behavior.

In order to produce functions, users must write them using the Arduino program architecture, i.e. writing the setup and loop functions (listing 4.2). The same function can be imported by multiple modules, allowing reusability.

Functions bear two constraints: they must never return values; and, functions’ scope is closed for the entire application removing visibility from every other function defined in other application’s modules. The rationale behind this was, first, the purpose of functions – or modules – is to implement closed behavior (i.e. modules must complete their execution and not depend on any other module to complete it), and second, closing the scope of the functions helps programmers define applications free of unintended
reassignments. In order for functions, and hence modules, to share data they must use the exposed methods for message passing `get()` and `put()` (detailed in subsection 4.3.3).

Since functions do not implicitly share data through the program flow, users can (in their definition) pass one argument to the function which will replace a variable called \_variable\_ in their custom implementations (example in listing 4.2) by injecting it during translation (subsection 4.3.4). This feature might be specially useful to reuse functions that may depend on a single variable.

**Port**: is the last markup and the only optional attribute used in the module definition. It defines input channels of communication with the module being defined. Modules which should receive data from others should have a port defined. Ports can have three sub-attributes to declare a data type, a maximum size of messages, and a substitution policy:

- Data type is defined with `data-type` to enforce communication through messages of that data type only. It can be set by the user, being restricted to primitive data types supported by C++ language.

- The maximum number of messages is defined with `pool-size`. If it is not defined by the user, then modules can only store one message.

- Substitution policy, which defines whether new messages should be dropped or inserted when the pool is full is defined with `substitution-policy`. The default value is true, which specifies that more recent messages should replace older ones. Otherwise, new messages are dropped.
Agent

Agents (listing 4.3) are compositions of [modules] (subsection 4.3.1). Since modules are basic units of behavior, agents represent the application itself, or part of it if there is more than one agent. Agents also 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. Agents can also be seen as physical boundaries since they are physically separated from each other, i.e. do not share memory and may eventually not be wired together (e.g. wireless connection).

Name: is an agent attribute that identifies it across the entire application. Agent names are unique through all the application. As it happens with modules, the decision aims to reduce redundancy when defining programs, but also to ensure uniqueness when producing the second addressing level of the application's addressing table (detailed in subsection 4.3.3).

Modules: are defined as part of agents using the module attribute (subsection 4.3.1). The order of declaration of modules is very important because since the Arduino's computing model is a round-robin loop the order of declaration will correspond to the order of execution.

Addresses: agents describe physical boundaries (one agent per Arduino). Resources like memory are then separate and not accessible between agents. To provide ways of communicating between agents, in their definition must be declared the technology(ies)/protocol(s) of communication used to interface with it, and the corresponding address(es) for that/these technology(ies)/protocol(s) hardware. Addresses are the part of the definition that describes how agents are physically exposed and must be addressed in the application. The address markup attribute is defined using the <protocol>::<address> notation – e.g. I2C::24. The purpose of this address attribute is for the eventuality of modules (subsection 4.3.1) sending messages to modules that do not belong to their agent; in those situations, the modules know how to forward the messages.

Links

Links describe how agents relate to each other in terms of communication protocols/technologies. While addresses (subsection 4.3.1) only specify agents’ addresses for a given communication technology, links describe how those agents are connected in terms of those technologies/protocols.

For a given agent there may be declared two technologies/protocols (e.g. I2C and Bluetooth, with the respective addresses 7 and 000A3A58F310). Nonetheless, this information is private to agents until translation time (section 4.3.4) and there is no way of modules to know how to send messages to other modules belonging in different agents. By describing which protocols/technologies bind agents together, links make it possible (using the agents’ addresses tag) to create valid communication after translation (section 4.3.4). Despite the existence of an outer tag attribute named links, agents must be paired in different link tags in order to create relations (example in listing 4.4).

Protocol: is the attribute in the Links definition which describes which communication protocol/technology is to be used when binding two agents in an Arduino application. (At this time we only support I2C protocol).

Agent: is the attribute that allows users to form couples of agents. A valid link definition must have two agents names, binding those agents using the specified protocol.
4.3.2 Framework

The tool provides an organized framework of directories, supporting users to produce their programs. The framework intends to help them better organizing the components that constitute the program itself – agents (subsection 4.3.1), modules, functions (subsection 4.3.1) and links (subsection 4.3.1) – 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. The reason there is a directory for agents, but not one for modules is that agents are the entry point for translation (section 4.3.4) and it would not make sense to have modules defined outside of their containing agents since they are not reusable. For the opposite reason, since functions are reusable, creating a folder where it is possible to organize them instead of declaring them inside the agents, increases the reusability of the tool. Hereupon, users can create their agents inside the agents’ folder under the XML extension (e.g. agent.xml).

Inside functions directory, users will find two more directories related with the packaging system referred in subsection 4.3.1 (standard and custom packages). The standard directory contains the standard functions packed with the tool. Users should have no need of modifying this directory but are allowed to use all provided functions using the standard prefix (std::). The custom directory will start
empty. Inside it, users can create their own functions – extending the behavior of their applications – using the .ino extension (the Arduino file extension). Those functions can then be used in agents’ modules, using the custom package prefix (cst::).

The output directory is where the translation results live (subsection 4.3.4). Once the translation process finishes, users will find files with .ino extension, with the names of their agents, ready to compile and deploy.

Listing 4.5: Framework folder work tree structure example

```
src/
  agents/
    agentOne.xml
    agentTwo.xml
  functions/
    standard/
      standardFunction.ino
    custom/
      customFunction.ino
  output/
  links.xml
```

4.3.3 Communication

Our tool aims to address the communication challenges inherent in the development of cyberphysical distributed applications. By communication challenges we include addressing local and remote modules and agents, marshaling and unmarshaling of messages, and communication protocol selection for message dispatching.

Despite all these concerns being abstracted from the user, five methods are provided to program the flow of that communication. All these methods must be called inside module functions (subsection 4.3.1). Two of these functions, get() and put(), provide message passing between modules, while the remaining three, isAvailable(), request() and hasRequest(), ensure message synchronization.

To facilitate the understanding of the following subsections, we have created two semantics: one for the modules requesting or reading messages called the “consumer modules”, and a second for the modules that have the purpose of sending or satisfying message requests, called the “producer modules”.

Message passing

**put** method: ensures the message passing 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 4.6: Interface of **put()** method

```
put(char* consumerModule, T payload) : void
```

Depending on the port policies of the consumer module and its presence in the producer module agent’s definition, **put** acts in different ways. If the consumer module port’s substitution policy is set to
true, then when it is full the oldest message is discarded and replaced with the new one; otherwise, the new message is discarded.

If the consumer module belongs to the same agent as the producer module, \texttt{put()} promptly places the message payload in the consumer module buffer because they share memory and buffers are global inside an agent (details in subsection \ref{subsec:sharing-memory}). Otherwise, the message is marshaled and sent through the communication channel binding the agents that own both producer and consumer modules. At the time of arrival, at the consumer module’s agent, the message is then unmarshaled and placed in the proper consumer module’s buffer.

\texttt{get} method: is responsible for retrieving messages from consumer modules’ buffers. It is a blocking method which means execution will block until a certain condition is achieved. \texttt{get} will block until there is something to be read from the consumer module’s buffer. \texttt{get} does not receive any parameters: its execution is always in the context of the consumer module, and all messages are retrieved from the consumer module’s buffer. \texttt{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 only read once.

\begin{verbatim}
Listing 4.7: Interface of get() method

get() : T

Synchronization

\texttt{isAvailable} method: is responsible for checking whether there is something in the consumer module’s buffer to be retrieved when using \texttt{get}, or not.

Since \texttt{get} is a blocking method, \texttt{isAvailable} is specially important to protect programs from unintended execution blocking when there is nothing to be read from the buffer. As with \texttt{get}, \texttt{isAvailable} does not receive any parameters.

\begin{verbatim}
Listing 4.8: Interface of isAvailable() method

isAvailable() : bool

Its execution is always in the context of the consumer module and thus the buffer being queried is the consumer module’s buffer. \texttt{isAvailable} returns a boolean signaling whether the buffer has messages – (returns true) or not (returns false).

\texttt{request} method: provides a way of requesting data. The consumer module sends a message to the producer module signaling some request.

If the producer module belongs to the same agent as the consumer module, \texttt{request} promptly marks the respective flag as requested (explained in detail in subsection \ref{subsec:sharing-memory}), because they share memory and flags are global inside an agent. If not, the request is marshaled and sent through the communication channel binding the agents that own both consumer and producer modules. At the time of arrival, at the producer module’s agent, the request is then unmarshaled marking the proper flag. The method is non-blocking and thus the program flow continues after its execution. There is no messages acknowledgment nor timeouts associated with the request.

\begin{verbatim}
Listing 4.9: Interface of request() method

request(char* producerModule) : void

35
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 name of the consumer module that requested a message as a parameter and checks if it was previously signaled by it.

hasRequest 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 4.10: Interface of hasRequest() method

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

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 a set of communication methods but also a pair of structures – Mapping and Flag.

The Communication Manager methods are the ones exposed and explained in subsection 4.3.3, but with different interfaces. The reason they have different interfaces is that users do not need to know low-level details of communication, but instead use it at an application level. Therefore, we conclude that for each method there is a user version and – as explained in detail in subsection 4.3.4 – a final or translated version.

As Mapping acts as a support structure for addressing tables (explained in subsection 4.3.3), the Flag structure provides support for the request()/hasRequest() synchronization methods. Flag contains a consumer module, a producer module and boolean to mark if the former requested messages from the latter.

The documented code for the Communication Manager can be found in Appendix A.1.

Addressing table

During translation (section 4.3.4), the tool generates an addressing table to support communication between participants in the produced system. The addressing table takes advantage of Mapping structure (part of the Communication Manager library). Apart from some properties useful to communication (explained in more detail in subsection 4.3.4), Mapping has two properties which promote a two-level addressing mechanism:

- name, which refers to the first addressing level (the module name).
- address, which refers to the second addressing level (the agent address).

During agent translation, mappings are created for modules. Depending on the existence, or not, of the module in the agent’s definition, the address on the Mapping structure is equal to -1 if the module does exist, or equal to the address of the agent defining that module (found on the that agent’s address tag) for the communication technology/protocol binding both agents (subsection 4.3.1).

When modules execute put or request methods, a lookup is performed on the addressing table (or mappings – subsection 4.3.4). 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 then marshaled, sent to the target agent through the address
and protocol provided (at the present moment, the tool only provides communication via I2C but the technology/protocol selection is already possible) and then unmarshaled again in the destination agent, like illustrated in figure [4.2]

Figure 4.2: Local communication vs. remote communication

Case 1:
1.1. moduleB looks up for moduleA on addressing table.
1.2. agentOne contains moduleA: the lookup stops.
1.3. message queued on moduleA buffer.

Case 2:
2.1. moduleA looks up for moduleC on addressing table.
2.2. agentOne does not contain moduleC (is on i2C:8 agent).
2.3. marshal message for I2C protocol.
2.4. message passed to agentTwo.
2.5. unmarshal message on i2C handler.
2.6. message queued on moduleC buffer.

4.3.4 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 major steps:

1. Parse the XML definitions of the application transforming the collected information into proper object-oriented structures;
2. Take those structures and translates them into C++ programs outputting .cpp files ready for compilation.

Parse program definitions

Agents: Given a valid application (i.e. the correct use of the framework (section 4.3.2)) 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. The reason we apply this transformation is explained by the use of Node.js runtime on the creation of the tool (allowing platform independence), which uses ECMAScript as the programming language. JSON representations are easier to maintain and manipulate, when using ECMAScript, than XML representations.

Having the transformation applied, it is possible to start creating ECMAScript Agent objects. There are two agent attributes that do not need any transformations, which are the name and the addresses.
Therefore, they are assigned to the Agent object, not without passing by a unique name validation (agents must have unique names across applications) and unique addresses validation (agents must not have more than one address for a given technology/protocol, nor have repeated addresses across the application for a given technology/protocol) as illustrated in Figures 4.3 and 4.4.

**Modules:** Despite modules are easily described in the agent definition, their inclusion in the program is not. At this point, Module objects are created, and their name and port are promptly assigned. This happens because all data related to those attributes are described in the XML definition, 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 (subsection 4.3.1) and function name. The tool looks into functions directory and looks for custom or standard directories depending on the package (custom or cst, or, standard or std). Then, it uses the already read function name to look for its related .ino file.

.ino 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 Module's collections: Setup and Loop.

The tool starts its first preprocessing task: since functions can take one argument at its definition, while reading lines of code from .ino files, the transpiler replaces every occurrence of the keyword _variable_ for the argument passed in the module definition. When this process finishes for all of the modules in an agent, Module objects are assigned to the Agent object.

**Links:** The goal of the links translation is to create a graph of connections between agents. Every
node in the graph denotes an agent and the connection between two nodes the technology binding those two agents together in the real system. 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. This graph will be of extreme importance when translating the communication supported by the tool. Then for every link in the definition, the agent attributes are read and a node is created (if there isn’t already one). The protocol value is then read and the connection for that protocol connects both nodes.

**Composing .ino programs**

.ino programs are the ones that actually run on top of Arduino boards. In order to use our tool to create Arduino programs, first we need to compose them.

The process of composition of .ino programs starts after the parsing of XML definitions ends and both Agent objects and Link graph are created. This process consists on successive writes into .ino files (one file per Agent object). For every file the translator outputs:

- dependencies;
- global variables;
- the `setup` and `loop` methods (specific of the Arduino software architecture);
- the module functions code;
- the I2C callback routine necessary to handle I2C communication between agents.
**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
Listing 4.11: Dependency inclusion

```cpp
#include <CommunicationManager.h>  // Communication Manager library
#include <Wire.h>                   // Arduino's Wire library
#include <Stepper.h>                // third party library
```

with the correct behavior of our tool's generated programs (as it is the case of Arduino's Wire library, and our Communication Manager library (example in listing 4.11)).

In order to optimize our solution, the translator has a conditional inclusion of dependencies. The Communication Manager library is only included if there is any trace of possible communication (e.g. modules using Communication Manager methods or having ports declared). The Arduino's Wire library is only included if there is at least one address in the translated agent using the I2C protocol. This conditional inclusion prevents unnecessary code from being included, decreasing the size of the program being compiled and uploaded to the target Arduino.

**Global variables:** Three possible global variables can be generated from the translation process: Buffers, Mappings and Flags (example in listing 4.12).

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. At this step, mappings are declared but not initialized. 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 in both put and request communication methods (subsection 4.3.3).

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. It is directly related to the hasRequest method (subsection 4.3.3). Request messages from other modules must be stored until they are fulfilled; that is the role of flags. As for mappings, at this point, the translator only declares flags but does not initialize them. The number of flags is equal to the number of modules used as destination modules used in hasRequest methods.

Listing 4.12: 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 specific objects: mappings and flags.
- initialize necessary communication libraries (like I2C).
- 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:

- Local modules owning 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 (listing 4.13);

Listing 4.13: Mapping initialization of local module with char type port of size 4

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

- Non-local modules found in local puts and requests: the Mappings are initialized using only the module name and the destination address of that module (listing 4.14). The address is found using the links graph to discover the technology/protocol binding the translated agent to the destination agent and then, looking for the address (in the address tag (subsection 4.3.1)) used in the latter for that technology/protocol.

Listing 4.14: Mapping initialization of non-local module with I2C address 248

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

Flags are also initialized in this step. For each hasRequest call 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 for the boolean that marks the existence of requests.

Listing 4.15: Flag initialization

```c
flags[0] = Flag("moduleA", "moduleB", false);
```
The last two steps of agent’s setup translation are initializing the communication libraries and including programmers’ setup code.

Since we only support I2C at this point, we check for I2C communication use in the program definition for every agent. If the protocol is in use, the Wire library (Arduino library supporting I2C protocol), is initialized with the address found in the address tag of the agent for that protocol. Also, because Wire requires a callback to handle incoming messages we register the onReceiveCallback (explained at I2C callback routine).

The user’s setup code inclusion is the last step where the translator just writes the exact lines of code, written by the user in the setup method of the agent modules’ functions, into the .ino file setup method.

Listing 4.16: Wire library initialization on address 11 and callback registration

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

Loop method: phase of translation is the most straightforward. 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(). The Arduino loop method, part of its programmatic model, is a round-robin method (i.e. it starts executing at the topmost line of code and ends at the bottommost, restarting all over again). Hence, most of the program logic should reside here (asynchronous callbacks like I2C callback routine do not). For readability reasons, we do not include the actual code, but only the modules functions invocation.

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 4.3.4) 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 4.3.3) are written to the .ino file as they were written by the user; others pass through a translation process. When programming with this tool, users are presented with a simplified version of the Communication Manager methods. However, there is a final version of those methods with more information needed for the correct behavior in both intra and extra agent communication.

When using the get method, users do not pass any arguments to this function because it should always run in the scope of the module where the method is being called. Hence, the translator adds 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 4.17).

Listing 4.17: get() method translation

```cpp
// user exposed version of get() method
char c = get();

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

The same concept of get() method applies to isAvailable() method, as it always runs in the scope of calling module. The translator adds the argument necessary for it to run on the final version, which is the the global buffer created for the module getting the data (example in listing 4.18).

Listing 4.18: isAvailable() method translation
hasRequest is another Communication Manager’s method being translated due to its need for extra arguments. The user’s version of it takes as argument the consumer module, i.e. the module requesting data. 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 4.19).

Listing 4.19: hasRequest() method translation

```c
// user exposed version of hasRequest() method
if (hasRequest(moduleA)) {

// translated version of hasRequest() method
if (hasRequest("moduleA", "moduleB", flags)) {
```

Lines containing request methods are also translated. request user’s version has one argument which is the producer module (i.e. the module to whom messages are requested). There are two final 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 4.20).

Listing 4.20: request() method translation

```c
// user exposed version of request() method
request(moduleB);

// translated version version of local request() method
request("moduleA", "moduleB", mappings, 2, flags);

// translated version of non-local request() method
request("moduleA", "moduleB", mappings, 2);
```

Finally, lines containing the put() method are also translated into one final version. put() receives two arguments: the message destination module and the message payload. Both arguments are used in its final version which contains the destination module name, the message payload, the mappings global variable and its size (example in listing 4.21).

Listing 4.21: 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 demands for a callback routine, which is triggered whenever communication is detected. In our application, there are two types of messages reaching agents: remote put and remote request messages. Both must be handled differently, not only because of their semantic differences but because the number of arguments coming in each one is different (subsection 4.3.3). This difference requires the code to handle the messages to be completely independent, which brings the advantage of also being both, or not, included in the final produced program.

To include the callback routine, the translator creates a function called `onReceiveCallback` and includes all necessary code to handle its messages. The code included is, as the dependencies, conditional. If the program contains a `flags` global variable, requests are coming in and this type of message must be handled. Then the translator includes the piece of code that handles the remote requests. If agent modules contain at least one port, put messages are coming in. This type of message must be handled and the translator includes the piece of code to handle it. Once again, this conditional inclusion of code prevents unnecessary code from being compiled and uploaded to Arduino.

The callback routine code can be found in Appendix A.2.

### Compile and deploy phases

Once the translation is complete, users have their output directory (section 4.3.2) populated with `.ino` files, with the names of their agents, ready for compile and deploy. These steps strongly depend on Arduino Software (official Arduino IDE) which provides both features. Users can open their `.ino` programs and use the Upload button to compile and deploy the program to their selected Arduino board (or use Verify button to just compile and check for errors).

Users select their Arduino board to deploy using two other Arduino Software’s features: board and port selection. Board selection allows users to choose an Arduino version (architecture) among the ones supported by the IDE. Port selection allows users to select one specific USB port where the board is connected. This method allows programmers to have multiple Arduinos connected to their computer during development/deployment.

### Use Case

Appendix A.3 shows an example of a successful translation, from declaration to `.ino` programs. The program is supposed to connect two Arduinos changing messages between each other.
Chapter 5

Evaluation

This chapter exposes the tests performed over the framework and its results. We divide the chapter in Workbench and Set of Exercises, and Code Generation tool because the components were evaluated separately.

5.1 Workbench and Set of Exercises

The Workbench and Set of Exercises proposed as part of the framework were tested during the first semesters of 16/17 and 17/18 on the Applications for Embedded Systems course of the Information Systems and Computer Engineering MSc degree.

According to the lecturer of this course, 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.

Despite the fact that there were no means of collecting data from the students (the evaluation of the framework was performed during Summer vacations), we believe that the use of real hardware and a set of exercises that exposes students to hands-on and real-life challenges had a major influence in this increasing number of attendees.

5.2 Code Generation tool

The Code Generation tool was tested in a different way. Because it is continuously being developed and improved we were cautious and did not include it in the classroom yet. Instead, it was assembled a group of individuals to perform a set of tests using the tool and a set of metrics the evaluate it.

5.2.1 Metrics

Based on similar tools we decided that good metrics to test ease-of-use in software development would be:

- lines of code;
- time of development.
Also, because our tool falls into the code generation genre, we decided to add another metric:

- size of programs in memory.

This last metric is important because, in hardware such as Arduino, the memory resources are limited.

5.2.2 Tests

The tests for ease-of-use were divided into four programming exercises, with different levels of difficulty. These were simpler versions of the Set of Exercises proposed. The exercises consisted in:

1. Blink a **LED** using Arduino for the first time (this test exercises the use of actuators and gives a soft introduction to Arduino and the tool).

2. Control a **LED** using a dimmer and exploring the concepts of **Pulse Width Modulation** (this test gives subjects an introduction to sensors, basics of electronics and some interaction between modules of software).

3. Using the software produced in exercise 2, physically separate the **LED** and dimmer in two Arduino boards, making them communicate via **I2C** (this test exercises the use of communication between Arduino boards and the use of the tool for distributed applications).

4. Create a traffic-light system with two Arduino boards, simulating an intersection, both controlled and synchronized by a third board (this test served as a final project to consolidate the subjects learned in the previous exercises).

The tests for ease-of-use were applied to two separated groups of five subjects. All the subjects were software engineers with one to five years of working experience in software development. None of them had experience with the development of software for hardware nor any contact with Arduino.

Each group had to produce the programs to fulfill the exercises. One of the groups (the control group) wrote their programs using the Arduino language (C++). The other, wrote the programs using our tool. No hardware assembly was necessary, since it was already provided during the tests, and the tool (the system under test) only supports the development of software.

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 [31]).

5.2.3 Notes

During the execution of the exercises, we took notes of the opinion and difficulties of the subjects.

When performing the exercises with C++, the control group made multiple comments about how similar it was to program Arduino compared to their usual day to day programming challenges. Nonetheless, two out of five subjects referred to the development of Arduino as tedious because it was no different from what they were used to do. In terms of difficulties, it seems that Arduino is already easy to learn and start with. The C++ was a non-issue to the subjects since they were given the necessary tutorials and documentation to execute the exercises. The main issue for the subjects was the use of communication between boards, mainly in the last exercise, which demanded a single board to communicate with multiple boards at once.
When performing the exercises with the tool, the five subjects were completely surprised by the paradigm used by it. Some referred that it was confusing at the start, but after the second exercise, they mentioned it as challenging, engaging and easy to use. The difficulties found by observations of the subjects’ execution of the exercises was mainly the setup of the programs. At this moment, the tool demands a lot of setups to declare agents, functions, and communication. Not only the subjects seemed confused navigating inside their programs directories, but this was also significant when comparing the time of execution using the tool versus C++. Additional comments referred to the expectation of finding more built-in functions, instead of writing their own.

5.2.4 Discussion

When evaluating the tool, given the lines of code, we took three relevant observations (table 5.1):

1. Small applications such as the ones in exercise 1 and 2 demand more lines of code when using the tool than when using C++. This happens because of the setup necessary to produce the agents and functions for the tool to generate Arduino programs.

2. Comparing the two approaches, as applications grow the lines of code tend to even between them. This showed that the first observation was no reason for concern.

3. When comparing not only the lines of code but going further and comparing lines related to application logic and communication logic, the bigger the application, fewer the lines for communication logic. In the last exercise (the one demanding more communication logic) the average number of lines of code for communication produced with the tool is half of the ones produced with C++. This compensates for the excess of lines produced for application logic caused by the setup.

Considering time spent developing the exercises (this time does not include time spent on preparation for the exercise nor time spent reading any documentation), we took four observations (table 5.2):

1. The users started the first exercise with no familiarity with the tool and its paradigm. A lot of questions was made during the exercise about how it worked and how things should be done. This caused the subjects to take some more time completing the exercise when using the tool, than when using C++. This showed us, that adopting this type of tool takes some time and education.

2. The second exercise was the one that revealed a wider variance between the time of execution. Subjects using the tool took almost three times more the time spent coding than users using C++. Due to observation, we realize that the main reason behind this was the fact that in this exercise subjects using the tool had to begin using communication between modules. This is a major difference when comparing to usual programming paradigms, mainly because the tool is used as a definition language.

3. In the third exercise, when transforming already developed non-distributed applications in distributed ones, the subjects took less time developing with the tool than with C++. This time the subjects were already familiar with communication using the tool and it was straightforward. Using C++, subjects had to learn how to communicate between boards.

4. The last exercise was the larger one. At this point, because of the previous exercises, both groups were familiar with the code necessary to complete their solutions. Observation allowed us to understand that when using the tool, despite the ease to define communication between boards, the navigation between the number of files and the setup necessary to create this application, consumed a lot of time during execution.
The third and last metric does not interfere with the ease of use of the tool, but is of extreme importance due to the constraints of Arduino (small flash and RAM memories) (table 5.3):

- The memory used by the best program created with the tool was higher (than the best created with C++) on the second exercise due to the use of communication, which generates some data structures to handle message traffic. This did not concern us and it is understandable. When comparing with C++ solutions also using board communication the use of RAM is approximately the same, since we tried to translate the definition language into a memory performant program.

- Considering the results, the main concern detected is the size in flash memory of the programs produced using the tool when communication is to be used. The root cause is the size of the Communication Manager library, which is included in every file using communication. This library alone weights 1468B.

<table>
<thead>
<tr>
<th>Exercise 1</th>
<th>Exercise 2</th>
<th>Exercise 3</th>
<th>Exercise 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior related lines</td>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Communication related lines</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
<td>4</td>
<td>15</td>
</tr>
</tbody>
</table>

**Table 5.1: Lines of code from control group (average)**

<table>
<thead>
<tr>
<th>Exercise 1</th>
<th>Exercise 2</th>
<th>Exercise 3</th>
<th>Exercise 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior related lines</td>
<td>9</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Communication related lines</td>
<td>0</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>9</td>
<td>13</td>
<td>18</td>
</tr>
</tbody>
</table>

**Table 5.2: Time spent coding (average)**

<table>
<thead>
<tr>
<th>Exercise 1</th>
<th>Exercise 2</th>
<th>Exercise 3</th>
<th>Exercise 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time spend coding (average)</td>
<td>1min 12sec</td>
<td>2min 10sec</td>
<td>2min 51sec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exercise 1</th>
<th>Exercise 2</th>
<th>Exercise 3</th>
<th>Exercise 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time spend coding (average)</td>
<td>2min 48sec</td>
<td>6min 13sec</td>
<td>2min 43sec</td>
</tr>
</tbody>
</table>

To depict the size of the programs in memory we collected the best solution from each exercise from each group.
### 5.3 Conclusion

#### 5.3.1 Workbench and Set of Exercises

Considering the opinions collected from the lecturer of the Applications for Embedded Systems, we honestly believe that this type of training and, more specifically, 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, year after year, of more attendees, proves that the use of such framework not only engages students but, reduces the fear of working with hardware as it normally happens with Information Systems and Computer engineers, which have almost no hardware knowledge.

#### 5.3.2 Code Generation tool

Considering the results of the tests we took good inferences of the actual state of the tool and what can be improved in the future. Taking into account the notes taken and the three metrics used to evaluate the tool we concluded that there are pros and cons to the use of the tool.

**As pros:**

- The effort to produce distributed applications using communication between boards is smaller and easier.
- The process of transforming already developed non-distributed applications into distributed ones is easier than when developing natively (C++).
- As a code generation tool it is already optimized to produce [RAM] and flash performant programs. There is still a problem when including the Communication Manager library, which can be solved by sectioning the library into smaller ones, and including only the parts needed.

**As cons:**

- The setup of the parts of the application (agents, functions, and communication) must be automated so that users don’t waste time writing it.
- 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 use to.

---

**Table 5.3: Memory usage by control group (best solution)**

<table>
<thead>
<tr>
<th></th>
<th>Exercise 1</th>
<th>Exercise 2</th>
<th>Exercise 3</th>
<th>Exercise 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control group</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash memory</td>
<td>940B</td>
<td>968B</td>
<td>6548B</td>
<td>7494B</td>
</tr>
<tr>
<td>RAM</td>
<td>9B</td>
<td>9B</td>
<td>732B</td>
<td>748B</td>
</tr>
<tr>
<td><strong>Group using the tool</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash memory</td>
<td>928B</td>
<td>2622B</td>
<td>6018B</td>
<td>9258B</td>
</tr>
<tr>
<td>RAM</td>
<td>9B</td>
<td>210B</td>
<td>492B</td>
<td>757B</td>
</tr>
</tbody>
</table>
The actual state of the tool does not allow us to already introduce it in the classroom, but it revealed signs of helping the students to develop systems for Arduino. The goal of this code generation tool was to understand if there were strategies to help to develop embedded systems in an easier way, than simply writing the systems native code.

Our tool proved helpful in wider systems with communication involved, with good feedback from subjects developing in it. This combined with what was verified in related work gives us the confidence to verify that the use of tools to help development may be the way no only to teach but also develop high-level software for embedded systems.
Chapter 6

Conclusion

In this work, we proposed a framework for training and development of embedded and cyberphysical systems for people with lack or no affinity for development of software for embedded hardware.

Parts of this framework were a Workbench and a Set of Exercises. These two components of the framework 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 relation with it.

Another part of the framework was the creation of a code generation tool to help students develop distributed applications for embedded systems. Using this tool, users should develop focusing mainly on the behavioral part of the application and rely on all the communication aspects such as configuration, message passage and code validation to it.

Analyzing previous work on support for the development of software for embedded hardware we understood that efforts were being done in the following directions: the creation of operating systems, virtual machines, and code generation tools (some with the support of a GUI). All of these attempts tried to help their users with constraints of embedded systems such as concurrency or communication. Only two gave support to Arduino, one used a virtual machine and another lacked the features that our work intended to provide. At this moment, we have a code generation tool with a definition language that serves as the interface for the user to define his program.

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. These transformations are generic but also performant. Arduino has some memory constraints that had to be taken into account so that our tool did not consume all the memory (flash and RAM) turning them unusable.

The tool was tested with two groups of engineers (5 each) with working experience in software development. Both groups had to work on 4 different programming challenges. The control group had to use the Arduino native language, C++, while the tested group had to use the tool. The tests were performed to test the ease of use of the tool and compare how the groups reacted to the two different approaches. 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.

6.1 Future work

At this moment, the framework is in a state that allows its inclusion in the classroom. The workbench is stable and [Instituto Superior Técnico (IST)] possesses all the necessary materials to keep lecturing courses using it. Nonetheless, we envision some additions not to the workbench itself but to the courses laboratories workspace. As an extension of the 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 this way interact with elements of the classroom (e.g. temperature sensors, DC motors, etc.).

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

Regarding the Code Generation tool, many tests were still to be done to have more certainty about the ease of use of tools that help developing software for embedded systems. Because this work was developed between semesters, we could not test the developed tool with a real class of students during the semester nor build systems with much more than three Arduino boards. These two aspects should be tested in the future to verify how real students embrace the Code Generation tool in real class challenges and how it allows scaling applications with a large number of boards in communication.

Taking into consideration the results and notes taken (chapter 5), 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 should do manually.

6.1.1 Code Generation tool Roadmap

There is a long road-map to enrich the Code Generation tool not only to help students but to turn it into an industry acceptable tool:

- Multiple communication protocols/technologies: at this moment the tool only supports I2C as a communication protocol between boards in an application. It is desired to support multiple communication protocols to create more interoperable application with the tool. The tool should support wireless communication protocols to allow the development of applications for unplugged boards, allowing more disperse architectures.

- Deploy on-the-fly: to reconfigure an application using the tool users should reconnect their boards to a computer in order to redeploy the application or parts of it. In the future, the tool should have the ability to deploy produced code on-the-fly to boards already in production. This feature could easily be implemented by taking advantage of the Bluetooth protocol. In the Bluetooth version of Arduino, or even using a Bluetooth shield, it is already possible to deploy code from a computer to a distant board.

- Network topologies: allow the users to choose specific network topologies to which the tool should adapt and configure by itself. This is an important feature since it would allow an entire reconfiguration of the network with no development effort.

- Services architecture: the tool communication works in a producer-consumer paradigm. In the future, it is desirable to have a services communication paradigm so that agents could consume
information from others by requesting that information. This would be a major feature in the tool considering its use in the classroom. With a services architecture, students or lecturers could create specific data provider agents and strategically position them around classrooms or laboratories so that other students could access the information by request (e.g. in a client-server paradigm).

- Graphical interface: as documented in related work, some tools to help develop software for embedded systems use graphical interfaces. In our work, we created a definition language using XML but for more inexperient users, with no knowledge of XML or even programming, the use of a graphical interface would ease the use of the tool and quickly introduce them to the Arduino and embedded systems environment.
Bibliography


Appendix A

Appendix chapter

A.1 Communication Manager

The following code snippet depicts the Communication Manager library responsible for the communication between Arduino boards in a distributed application developed using the tool developed in the scope of this work. It is composed of two files: the header file where the interface is declared and the implementation file.

```cpp
// CommunicationManager.h
#ifndef _COMMUNICATION_MANAGER_H_
#define _COMMUNICATION_MANAGER_H_

#if defined(ARDUINO) && ARDUINO >= 100
#include "Arduino.h"
#else
#include "WProgram.h"
#endif
#include "HardwareSerial.h"

#include <limits.h>

class Mapping {
public:
    char* _name;
    char* _charArray;
    int* _intArray;
    char* _type;
    int _address;
    int _size;

public:
    Mapping();
    Mapping(char*, char*, int, int*, int);
    Mapping(char*, char*, int, char*, int);
    Mapping(char*, int);
};
```
class Flag {
    public:
        char* _name;
        bool _flag;

    public:
        Flag();
        Flag(char*, bool);
};

int get(int*, int);
char get(char*, int);
bool isAvailable(int*);
bool isAvailable(char*);
void put(char*, char, Mapping*, int);
bool put(char*, int, Mapping*, int);
bool hasRequest(char*, Flag*);
void request(char*, char*, Mapping*, int);
void request(char*, Mapping*, int, Flag*);

#endif

// CommunicationManager.cpp
#include <CommunicationManager.h>
#include <string.h>
#include <Wire.h>

Mapping::Mapping() { }
Mapping::Mapping(
    char* name, char* type, int address, int* buffer, int size
) {
    this->_name = name;
    this->_type = type;
    this->_address = address;
    this->_intArray = buffer;
    this->_size = size;
}

Mapping::Mapping(
    char* name, char* type, int address, char* buffer, int size
) {
    this->_name = name;
    this->_type = type;
    this->_address = address;
    this->_charArray = buffer;
    this->_size = size;
Mapping::Mapping(char* name, int address) {
    this->name = name;
    this->address = address;
}

Flag::Flag() {}

Flag::Flag(char* name, bool flag) {
    this->name = name;
    this->flag = flag;
}

int get(int* buffer, int maxSize) {
    while (buffer[0] == INT_MIN) {}
    int temp = buffer[0];
    for (int i = 0; i < maxSize - 1; i++) {
        buffer[i] = buffer[i + 1];
    }
    buffer[maxSize - 1] = INT_MIN;
    return temp;
}

char get(char* buffer, int maxSize) {
    while (buffer[0] == CHAR_MIN) {}
    char temp = buffer[0];
    for (int i = 0; i < maxSize - 1; i++) {
        buffer[i] = buffer[i + 1];
    }
    buffer[maxSize - 1] = CHAR_MIN;
    return temp;
}

bool isAvailable(int* buffer) {
    return (buffer[0] != INT_MIN);
}

bool isAvailable(char* buffer) {
    return (buffer[0] != CHAR_MIN);
}

void put(char* module, char val, Mapping* mappings, int size) {
    Mapping mapping;
    for (int i = 0; i < size; i++) {
        if (strcmp(mappings[i].name, module) == 0) {
            mapping = mappings[i];
            break;
        }
bool put(char* module, int val, Mapping* mappings, int size) {
    Mapping mapping;
    for (int i = 0; i < size; i++) {
        if (strcmp(mappings[i].name, module) == 0) {
            mapping = mappings[i];
            break;
        }
    }
    if (mapping._address == -1) {
        for (int i = 0; i < mapping._size; i++) {
            if (mapping._charArray[i] == CHAR_MIN) {
                mapping._charArray[i] = val;
                break;
            }
        }
    }
    else {
        Wire.beginTransmission(mapping._address);
        Wire.write("P");
        Wire.write(module);
        Wire.write("C");
        Wire.write(val);
        Wire.endTransmission();
    }
}

bool hasRequest(char* module, Flag flags[]) {
}
for (int i = 0; ; i++) {
    if (strcmp(flags[i].name, module) == 0) {
        bool temp = flags[i].flag;
        flags[i].flag = false;
        return temp;
    }
}

void request(char* module, Mapping* mappings, int size, Flag* flags) {
    Mapping mapping;
    for (int i = 0; i < size; i++) {
        if (strcmp(mappings[i].name, module) == 0) {
            mapping = mappings[i];
            break;
        }
    }
    if (mapping._address == -1) {
        for (int j = 0; ; j++) {
            if (strcmp(flags[j].name, module) == 0) {
                flags[j].flag = true;
            }
        }
    } else {
        Wire.beginTransmission(mapping._address);
        Wire.write("R");
        Wire.write(module);
        Wire.endTransmission();
    }
}

void request(
    char* producerModule, char* consumerModule, Mapping* mappings, int size
) {
    Mapping mapping;
    for (int i = 0; i < size; i++) {
        if (strcmp(mappings[i].name, producerModule) == 0) {
            mapping = mappings[i];
            break;
        }
    }
    Wire.beginTransmission(mapping._address);
    Wire.write("R");
    Wire.write(consumerModule);
    Wire.endTransmission();
}
A.2 I2C callback routine

The following code depicts the I2C callback routine, loaded into every Arduino program when using the tool developed in the scope of this work. This code is responsible for the unmarshal of every message received via the I2C protocol.

```c
void onReceiveCallback(int bytes) {
    char instruction;
    char module[2];
    if (Wire.available()) instruction = Wire.read();
    if (Wire.available()) module[0] = Wire.read();
    module[1] = '\0';
    if (instruction == 'P') {
        char type;
        if (Wire.available()) type = Wire.read();
        if (type == 'I') {
            int val = 0;
            while(Wire.available()) {
                int i = Wire.read();
                val = (val * 10) + i;
            }
            put(module, val, mappings, '+mappingsSize+');
            return;
        } else if (type == 'C') {
            char val;
            while(Wire.available()) {
                val = Wire.read();
            }
            put(module, val, mappings, '+mappingsSize+');
            return;
        }
    }
    if (instruction == 'R') {
        for (int i = 0; ; i++) {
            if (flags[i].name == module) {
                flags[i].flag = true;
                return;
            }
        }
    }
}
```
A.3 Tool use case

The following snippet of code depicts an example of a use case for the tool produced in the scope of this work. The example shows the program definitions written in order to produce a distributed application where one Arduino is sending to another the characters in the ‘Hello’ word, one character every second, in an infinite loop.

Listing A.1: Example of program definitions

```xml
<agent definition>
  <name>agentOne</name>
  <address>I2C::7</address>
  <module>
    <name>b</name>
    <function>cst::functionOne(2)</function>
  </module>
  <module>
    <name>c</name>
    <function>std::sleep(500)</function>
  </module>
</agent>

<agent definition>
  <name>agentTwo</name>
  <address>I2C::11</address>
  <module>
    <name>a</name>
    <function>cst::helloWorld()</function>
    <port>
      <data-type>char</data-type>
      <pool-size>1</pool-size>
    </port>
  </module>
  <module>
    <name>c</name>
    <function>std::sleep(2000)</function>
  </module>
</agent>

<links definition>
  <link>
    <protocol>I2C</protocol>
    <agent>agentOne</agent>
    <agent>agentTwo</agent>
  </link>
</links>
```
Listing A.2: Example of Arduino programs written by the produced tool

// agentOne program
#include <CommunicationManager.h>
#include <Wire.h>
Mapping mappings[1];
Flag flags[1];

void setup(void) {
  Wire.begin(7);
  Wire.onReceive(onReceiveCallback);
  flags[0] = Flag("a", "b", false);
  mappings[0] = Mapping("a", 11);
}

void loop(void) {
  b();        
  c();
}

void b(void) {
  static int initialValue = 2;
  char* hello = "Hello";
  char c = hello[initialValue % 5];
  if (hasRequest("a", "b", flags)) {
    put("a", c, mappings, 1);
    initialValue++;
  }
}

void c(void) {
  delay(500);
}

void onReceiveCallback(int bytes) {
  char instruction;
  char module[2];
  if (Wire.available()) instruction = Wire.read();
  if (Wire.available()) module[0] = Wire.read();
  module[1] = '\0';

  if (instruction == 'P') {
    char type;
    if (Wire.available()) type = Wire.read();
    if (type == 'I') {
      int val = 0;
    }
  }
}
while(Wire.available()) {
    int i = Wire.read();
    val = (val * 10) + i;
}
put(module, val, mappings, 1);
return;
} else if (type == 'C') {
    char val;
    while(Wire.available()) {
        val = Wire.read();
    }
    put(module, val, mappings, 1);
    return;
}
}
if (instruction == 'R') {
    char producer[2];
    if (Wire.available()) producer[0] = Wire.read();
    producer[1] = '\0';
    for (int i = 0; i < 1; i++) {
        if ((strcmp(flags[0].producerModule, producer) +
            strcmp(flags[0].consumerModule, module)) == 0) {
            flags[i].flag = true;
            return;
        }
    }
}
}

// agentTwo program
#include <CommunicationManager.h>
#include <Wire.h>
char aBuffer[1];
Mapping mappings[2];

void setup(void) {
    Wire.begin(11);
    Wire.onReceive(onReceiveCallback);
    Serial.begin(9600);
    for (int i = 0; i < 1; i++) aBuffer[i] = CHAR_MIN;
    mappings[0] = Mapping("a", "char", -1, aBuffer, 1);
    mappings[1] = Mapping("b", 7);
}

void loop(void) {
    a();
    c();
}
```c
void a(void) {
    if (isAvailable(aBuffer)) {
        char c = get(aBuffer, 1);
        Serial.print(c);
    } else {
        request("a", "b", mappings, 2);
    }
}

void c(void) {
    delay(2000);
}

void onReceiveCallback(int bytes) {
    char instruction;
    char module[2];
    if (Wire.available()) instruction = Wire.read();
    if (Wire.available()) module[0] = Wire.read();
    module[1] = '0';

    if (instruction == 'P') {
        char type;
        if (Wire.available()) type = Wire.read();
        if (type == 'I') {
            int val = 0;
            while (Wire.available()) {
                int i = Wire.read();
                val = (val * 10) + i;
            }
            put(module, val, mappings, 2);
            return;
        } else if (type == 'C') {
            char val;
            while (Wire.available()) {
                val = Wire.read();
            }
            put(module, val, mappings, 2);
            return;
        }
    }
}
```
A.4 Exercises

The following documents are the assignment papers for the exercises developed for the framework proposed.

A.4.1 Exercise 1

“Building an embedded system”

Exercise 1

<table>
<thead>
<tr>
<th>Group:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Goal:
This work goal is to put students for the first time in touch with Actuators (LEDs in this work).

ETA: 30 minutes

Description:
Build an embedded system using Arduino UNO board to control 4 LEDs. Every second one LED should be on, while the other three off. During the four seconds, all LEDs should have been on at least one time. After those four seconds the system must start over with the same behavior. For this class it will be given a diagram of the assembly for students to accelerate the process of assembly.

References:

Recommendations:
In order to fulfill your work with security and not damaging the hardware involved, remember to carry out the recommendations below. As you are working fill the boxes to be certain that you fulfill all security measures.

- Always work with the circuit disconnect from the source.
- Call the professor or responsible for the laboratory, before you connect the circuit to the source.
Make sure the circuit is well connected (resistors, capacitors, etc.) to prevent a short circuit, or damage the hardware.

Choose the right resistor:
Results:

- Code:

```cpp
void setup() {
}

void loop() {
}
```

- Questions:

Q:

A:

Q:
A.4.2 Exercise 2

“Sensing the Real World”

Exercise 2

<table>
<thead>
<tr>
<th>Group:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student 1:</td>
</tr>
<tr>
<td>Student 2:</td>
</tr>
<tr>
<td>Student 3:</td>
</tr>
</tbody>
</table>

Goal:
The goal of this laboratory class is for students to retrieve real world’s physical measures and at the same time to create the behavioral relation between Sensors and Actuators present in the description provided.

ETA: 70 minutes

Description:
Build an embedded system using Arduino UNO board to control 3 LEDs depending on the state of 3 different sensors (temperature, potentiometer and light intensity). The LED controlled by the temperature LED must be turned on when it senses temperature values over [ ]. The potentiometer LED must blink with periods of time between [ ] and [ ] seconds depending of the rotation applied to it. The light intensity LED must change its own light intensity based on the light intensity collected from the environment (read Reference 4 about Pulse Width Modulation). A diagram of the assembly is given for students to accelerate the process of assembly.

References:
Recommendations:

In order to fulfill your work with security and not damaging the hardware involved, remember to carry out the recommendations below. As you are working fill the boxes to be certain that you fulfill all security measures.

- Always work with the circuit disconnect from the source.
- Call the professor or responsible for the laboratory, before you connect the circuit to the source.
- Make sure the circuit is well connected (resistors, capacitors, etc.) to prevent a short circuit, or damage the hardware.

Analogic measures calibration:

The readings retrieved from the sensors do not correspond to accurate measurements, but values between 0 and 1024 that depend of the circuit inside them. Due that, some may need calibration. When the the value is something relative a simple mapping is sufficient, but to get the real measure some mathematical calibration may be needed.

For example, when rotating a Servo motor with input from a potentiometer we know that when the value of the potentiometer is 0 then the servo angle must be 0° as well. Logically, if the value of the potentiometer is 1023 (its max) than, the servo angle must be 180° (its max).

\[
\text{map(value, 0, 1023, 0, 180)}
\]

By other side, real temperature is given by a specific formula dependent of the voltage in the circuit:

\[
(T = \frac{(\text{sensor value} / 1024.0) \times 5.0 - 0.5 \times 100}{100})
\]

Work with Analog Sensors (code):

To control an analog sensor you must attach it to an Arduino analog pin. The software allocation of a sensor to an analog pin is done using the following:

```c
int const tempSensor = A0; where A0 is the physical pin where the sensor is attached.
```

To read the value assigned to a specific pin use Arduino function `analogRead(PIN)`, as follows:

```c
int temperatureValue = analogRead(tempSensor);
```

Debug:

In order to control some variables and debug your program it is possible to print them to the Serial Monitor present in Arduino IDE. This kind of mechanisms is useful, for example, when
you want to keep track of the temperature variation but you don’t know the exact temperature in the room.

First start by starting a serial communication with the PC:

```c
void setup() { ... Serial.begin(9600); ... }
```

Then, just `Serial.print` your variables and/or strings:

```c
Serial.println(temperatureV alue);
```

---

**Diagram:**

- T: Temperature
- L: Light
- P: Presence
- Pot: Potentiometer
Results:

- Code:

```c
void setup() {
}

void loop() {
}
```
Questions:

Q:

A:

Q:

A:
“Wired Communication”

Exercise 3

Group:

<table>
<thead>
<tr>
<th>Student 1</th>
<th>Student 2</th>
<th>Student 3</th>
</tr>
</thead>
</table>

Goal:
The goal of this work is for students to understand how wired communication between two embedded systems work. In the end of this laboratory students should master I2C communication in Arduino boards and understand its implications.

ETA: 90 minutes

Description:
This work is based on the assembly of the previous one, but with one slighty difference, For the first time there are two Arduinos UNO to use, and Sensors must be on one Arduino board while the Actuators (LEDs) on the other. The two Arduinos must communicate via I2C. The goal of the assembly is the same as the third work:

"(...) control 3 LEDs depending on the state of 3 different sensors (temperature, potentiometer and light intensity). The LED controlled by the temperature LED must be turned on when it senses temperature values over [...]. The potentiometer LED must blink with periods of time between [ ] and [ ] seconds depending of the rotation applied to it. The light intensity LED must change its own light intensity based on the light intensity collected from the environment (read Reference 4 about Post Width Modulation). (...)"

A diagram of the assembly is given for students to accelerate the process of assembly.

References:
Recommendations:

In order to fulfill your work with security and not damaging the hardware involved, remember to carry out the recommendations below. As you are working fill the boxes to be certain that you fulfill all security measures,

- Always work with the circuit disconnect from the source.
- Call the professor or responsible for the laboratory, before you connect the circuit to the source.
- Make sure the circuit is well connected (resistors, capacitors, etc.) to prevent a short circuit, or damage the hardware.

Inter-Integrated Circuit (I2C):

In order for this assembly to work, you will need to create communication between both Arduino boards. For this work you will use I2C communication. I2C works in a Master-Slave protocol where one or more Slaves can be connected to one Master on the same bus. However, there is no fault tolerance policy between the Arduinos’ communication.

To control the I2C bus, fortunately, there is already an Arduino Library to handle the application level of the communication (read Reference 4 about the Wire library).

To start communication between boards both Arduinos must start the Wire communication:

```cpp
void setup() { Wire.begin(8); } // begin only receives an argument if the board is a Slave. Masters do not need an address.
```

Future communication may be performed as Master-Writer:

```cpp
void loop() {
  // Master writes for a Slave to read
  Wire.beginTransmission(8); // Transmission for port 8 of the I2C bus
  Wire.write("Value:"); // Wire.write(string) reads every char as a byte
  Wire.write(800); // Wire.write(int) reads the int as byte
  Wire.endTransmission();
  ...
}
```

Slaves need some more effort to read. When some message is coming, the I2C port causes an interrupt. Wire Library uses the onReceive function to associate a callback function to the interrupt:

```cpp
void setup() {
  ...
  Wire.onReceive(callbackFunction);
}
```
void loop() {
  ... 
}

For the Master example above this function will consume and print the string and let the
  integer be printed separately

void callbackFunction(int i) {
    while (i < Wire.available()) {  // make sure there is something to read
        char c = Wire.read();       // read the next byte as a char
        Serial.print(c);            // print the char
    }
    int x = Wire.read();          // read the next byte as an int
    Serial.println(x);            // print the int in a new line
}
Results:

- Code - Arduino 1:

```c
void setup() {
}

void loop() {
}
```
• Code - Arduino 2:

```c
void setup() {
}

void loop() {
}
```
A.4.4 Project

“Marquês do Pombal Traffic Light System”

Goal:
The goal of this project is for students to build an automatic traffic light system, where all traffic lights are connected via I2C with a controller. With this project students should learn how to deal with problems such as Concurrency, Fault Tolerance, and understand the concept of Modular Programming.

ETA: 270 minutes

Description:
*Câmara Municipal de Lisboa* is willing to change all the traffic light system around the monument of Marquês do Pombal. Due to that, they are hiring multiple teams to develop, each one, two traffic lights and one controller. The main requirement is that every team’s traffic light *MUST* function properly with any other team’s controller. In the end they are buying the best traffic light and the best controller.

Traffic Light:
The traffic light consist of an Arduino UNO with multiple I/O devices. Every traffic light must have 5 LEDs: 3 of them represent the cars color traffic rules, green, yellow and red; and 2 represent the pedestrians color traffic rules, green and red. Also, every traffic light must have a pedestrian button and a potentiometer to control the time between color transitions.

Controller:
The controller is also an Arduino UNO with I/O devices attached. 3 LEDs, each one with one function: a green and a red to show the controller status (on and off) and a blue one to signal the presence of communication. Such as the traffic lights, the controller must have a button to switch between on and off.

Requirements:

1. The initial state of the system must be with the controller turned off (red light on and green light off), and both traffic lights blinking yellow, meaning they are not in control of the controller.
2. It must be possible to turn the controller on and off, pressing the button:
   a. When turned on, the controller must signal one of its traffic lights to go green and
      the other to go red.
   b. When turned off, the controller must signal both traffic lights to start blinking
      yellow, going back to initial state.
3. While receiving or sending data the controller’s blue LED must blink.
4. The traffic light color transitions must be:
   b. Yellow -> Green.
   c. Green -> Yellow.
5. It must be possible to control the red-yellow-green-yellow-red cycle time of a single traffic
    light using a potentiometer (min: 2 seconds between light transition; max: 10 seconds
    between light transition).
6. It must be possible to shorten the cycle time by half by pressing the traffic light
    pedestrian button.
7. When one traffic light is performing a red-yellow-green-yellow-red cycle, the other must
    have its red light always on (pedestrian green).
8. All communication between the traffic lights must be performed via controller (read
    Section Modular Programming).
9. It must be possible for both controller and traffic light, to detect faults of communication:
   a. After 30 seconds of faults:
      i. The traffic light that detected missing communication must start blinking
         yellow.
      ii. The controller must communicate to the other traffic light to start blinking
         yellow, too. Then it must turn itself off, going back to the initial state of the
         system.
10. All system communication must be performed using I2C.

Modular Programming:

Communication between traffic light - controller - traffic light must be specified by an API in
order to create a modular programming approach. This enables two possibilities: first, it allows
two teams of programmers to build both controller and traffic light separately and
simultaneously; second, it allows controllers and traffic lights from different teams to work
properly.

Controller - Traffic light communication API:

Controller - controller should implement functions to be called when the following messages are
received via I2C:
- RED (X): where X is the identifier of the traffic light that sends the message. It alerts the controller that the specific traffic light has turned red.
- PING (X): where X is the identifier of the traffic light that sends the message. It requests the controller for a signal that it is still alive.
- ACK (X): where X is the identifier of the traffic light that sends the message. It is the "I'm alive" signal from the traffic light that sends the message.

Traffic lights - traffic lights should implement functions to be called when the following messages are received via I2C:
- ON (CLR): where CLR is the traffic light color identifier which the traffic light should be started with (RED: 1; YELLOW: 2; RED: 3).
- OFF: upon receiving this message the traffic light should start blinking yellow.
- GRN: this message signals the traffic light to start its color cycle.
- TIME (X): where X is the time in milliseconds between 2000 and 10000. This message signals the traffic light to reset the time between light transitions.
- PING: It requests the traffic light for a signal that it is still alive.
- ACK: It is the "I'm alive" signal from the controller.

I2C with Multi Master-Slave diagram:

A diagram is given so students can understand how to build a multi Master-Slave circuit for Arduino UNO.

Recommendations:

In order to fulfill your work with security and not damaging the hardware involved, remember to carry out the recommendations below. As you are working fill the boxes to be certain that you fulfill all security measures.
<table>
<thead>
<tr>
<th>Always work with the circuit disconnect from the source.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call the professor or responsible for the laboratory, before you connect the circuit to the source.</td>
</tr>
<tr>
<td>Make sure the circuit is well connected (resistors, capacitors, etc.) to prevent a short circuit, or damage the hardware.</td>
</tr>
</tbody>
</table>
A.5 Arduino UNO board architecture

Figure A.1: Arduino UNO board

Board architecture (UNO)

- Microcontroller: it has 28 pins (Arduino UNO version \(32\)) and it is fitted into a socket at the center of the board. Internally, the microcontroller is composed by:

  - UART (universal asynchronous receiver-transmitter);
  - RAM: where data related to runtime is stored (2KB);
  - Flash memory: where the program itself is stored (32KB);
  - EEPROM (electrically erasable read-only memory): a non-volatile memory that persists after turning off;
  - I/O ports;
  - CPU: capable of performing 16 million operations per second.

- Power connections: Arduino has 5 different power related pins. 3.5V, 5V, and 9V provide the respective voltages. Plus, there is the \(\text{RESET}\) pin, which serves the same purpose as the reset button (restarts programs), and the \(\text{GND}\) pin (ground) for 0 volts.

- Analog inputs: pins labeled from A0 to A5 (6 analog pins) can be used to measure the voltage connected to them.

- Digital connections: pins labeled from D0 to D13 (14 digital pins). Contrary to analog inputs they can only have two states: on \((\geq 2.5V)\) or off \((<2.5V)\). Pins 0 and 1, labeled as RX and TX respectively, are both reserved for USB communication between the board and the computer. These can be used as digital inputs or outputs, to both measure and transmit voltage, or as analog outputs, using Pulse Width Modulation technique \([33]\) on pins D3, D5, D6, D9, D10, and D11.
A.6 Arduino Starter Kit

The Arduino Starter Kit includes:

- 1 Projects Book (170 pages);
- 1 Arduino UNO;
- 1 USB cable;
- 1 Breadboard 400 points;
- 70 Solid core jumper wires;
- 1 Easy-to-assemble wooden base;
- 1 9V battery snap;
- 1 Stranded jumper wires (black);
- 1 Stranded jumper wires (red);
- 6 Phototransistor;
- 3 Potentiometer 10kOhms;
- 10 Pushbuttons;
- 1 Temperature sensor [TMP36];
- 1 Tilt sensor;
- 1 alphanumeric LCD (16x2 characters);
- 1 LED (bright white);
- 1 LED (RGB);
- 8 LEDs (red);
- 8 LEDs (green);
- 8 LEDs (yellow);
- 3 LEDs (blue);
- 1 Small DC motor 6/9V;
- 1 Small servo motor;
- 1 Piezo capsule [PKM17EPP-4001-B0];
- 1 H-bridge motor driver [L293D];
- 1 Optocouplers [4N35];
- 2 Mosfet transistors [IRF520];
- 5 Capacitors 100uF;
- 5 Diodes [1N4007];
• 3 Transparent gels (red, green, blue);
• 1 Male pins strip (40x1);
• 20 Resistors 220 Ohms;
• 5 Resistors 560 Ohms;
• 5 Resistors 1 kOhms;
• 5 Resistors 4.7 kOhms;
• 20 Resistors 10 kOhms;
• 5 Resistors 1 MOhms;
• 5 Resistors 10 MOhms.