University of the study of Camerino

School of Science and Technology

Degree in Computer Science (Class LM-18)

Model-Driven Development of IoT based Software

Student:

Marasca Luca
ID n. 107104

Supervisor

Polini Andrea

Co-Supervisor

Fornari Fabrizio

A.Y. 2019/2020

I'm personally convinced that computer science has a lot in common with physics. Both are about how the world works at a rather fundamental level. The difference, of course, is that while in physics you're supposed to figure out how the world is made up, in computer science you create the world. Within the confines of the computer, you're the creator. You get to ultimately control everything that happens. If you're good enough, you can be God. On a small scale.

-Linus Torvalds, creator of Linux

Abstract

There is an increasing demand for software systems that utilize the new Internet of Things (IoT) paradigm to provide users with the best functionalities, through transforming objects from traditional to smart ones. In general, the Internet of Things (IoT) represents a comprehensive environment that consists of a large number of smart devices interconnecting heterogeneous physical objects to the Internet.
In recent years, a number of approaches have been proposed to enable the development of such IoT systems. However, developing IoT systems that adapt at runtime is still a major challenge. But, since many domains such as logistics, manufacturing, agriculture, urban computing, home automation, ambient assisted living and various ubiquitous computing applications have utilised IoT technologies; has been decided to find an efficient and adaptable approach for implementing a system IoT based.
The proposed approach, suggest a methodology based on Model-Driven Engineering that focuses on creating and exploiting domain models, which are, conceptual models of all the topics related to a specific problem. The conceptual model taken in exam it's the BPMN (Business Process Model and Notation), since, it has become a successful and efficient solution for coordinated management and optimised utilisation of resources/entities.
The final result, it's the definition of a framework based on a model-driven approach and a software tool (both explained in the following chapters), that whether used can act as meeting point between the two reality of modeling and programming, simplifying the interactions between them, and reducing the time spent for the product implementation.

Index

Abstract **5**

1 Introduction **9**
- 1.1 Business Process Model . . . 9
- 1.2 Internet of Things . . . 11
- 1.3 Objectives . . . 12

2 Related Work **13**
- 2.1 Modeling IoT scenarios . . . 13
- 2.2 How To Represent IoT in Model . . . 14
- 2.3 Model translation . . . 19

3 Methodology **23**
- 3.1 BPMN and modeling environments . . . 23
- 3.2 Represent IOT using BPMN . . . 25
- 3.3 Specify Additional Information . . . 26
- 3.4 Convert BPMN to Code . . . 27
- 3.5 Used Technologies . . . 29
  - 3.5.1 Xtext . . . 29
  - 3.5.2 Starting with Xtext . . . 30
  - 3.5.3 Xtend . . . 31

4 Implementation **33**
- 4.1 The grammar . . . 33
- 4.2 Code generation . . . 46

5 Supported devices and software usage **59**
- 5.1 Sensors, protocols and language . . . 60
- 5.2 How to use the software . . . 74

6 Framework Testing **75**
- 6.1 Test 1 . . . 75
- 6.2 Test 2 . . . 78
- 6.3 Test 3 . . . 82
- 6.4 Test 4 . . . 84

7 Conclusion **87**

8 Acknowledgements **93**

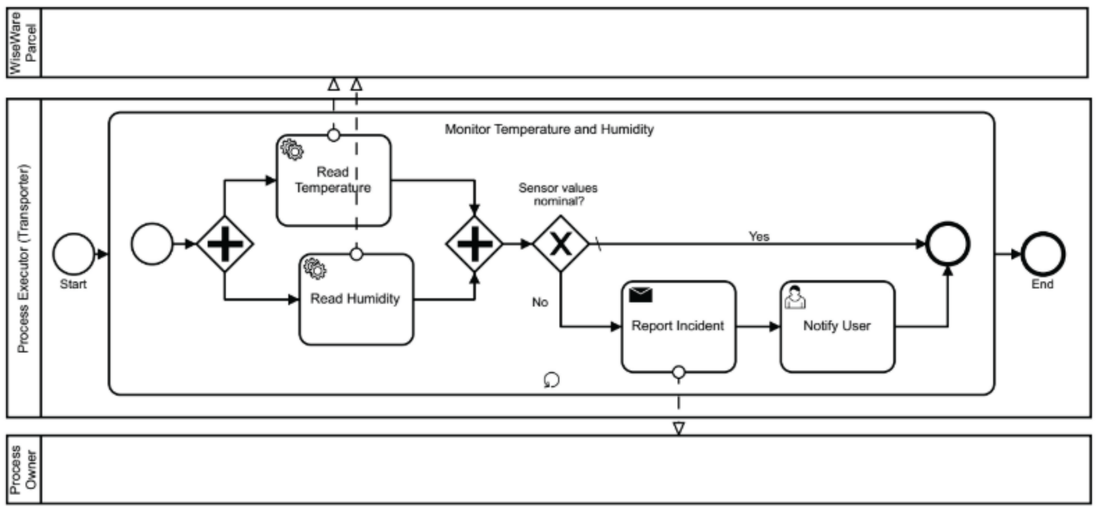
1. Introduction

In this chapter of the paper, it's proposed an introduction about the work that has been performed. In particular, the first two subsection regards a theoretical introduction about the Business model and the IoT environment. Then an explanation of the objectives it's given, taking in exams the main issues that the work solves.

1.1 Business Process Model

Over the past two decades, the term "business model" has frequently been misused by both academics and practitioners. It is common to hear the term being used by managers, consultants or scholars from diverse fields and even in the popular media. In general, Model-Driven Engineering (MDE) is based on models that in an early stage in the development minimize the technological aspects, so that communication between users, analysts, and system developers, can be more efficient, allowing the selection of the technological platform until the end of the process [36]. Since that, it's possible to define a business process as a collection of linked tasks which find their end in the delivery of a service or product to a client, so, a Business process execution is the actual operation to produce business products or achieve business objectives based on predefined business process models [39]. A business process can also been defined as a set of activities and tasks that, once completed, will accomplish an organizational goal. Processes are executed within application systems that are part of the real world involving humans, cooperative computer systems as well as physical objects [36]. Processes must involve clearly defined inputs and a single output. These inputs are made up of all of the factors which contribute (either directly or not) to the added value of a service or product. These factors can be categorized into management processes, operational processes and supporting business processes [10].

Since the business process could be more or less articulated, there's the necessity of represent it for better follow the various tasks that has to be performed. Business Process Model, is a common concept used to build and compose software [34], that emerged in order to provide a better understanding of business processes in organizations [14]. In general, the result of business process modelling is a process model, which consists of a set of activity models and execution constraints between them [15]. It's possible to define a Business Process Model, as a network of graphical objects, which are activities (i.e., work) and the flow controls that define their order of performance [35]; with business process modelling, then, companies and organizations can gain explicit control over their processes. Currently, there are many notations in the area of business process modelling, and Business Process Model and Notation (BPMN) is denoted as the "de facto standard" [13]. BPMN It is usually illustrated with activities and events that are associated with control flows [16].
In figure 1.1 Is shown an example of BPMN that has been used in an academic article for a demonstration.



Picture 1.1: An example of BPMN concerning the temperature and humidity monitor [17]

1.2 Internet of Things

The Internet of Things (IoT) is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. In other words, IoT represents a comprehensive environment that consists of a large number of smart devices interconnecting heterogeneous physical objects to the Internet [32], Internet of Things technologies and applications are evolving and continuously gaining traction in all fields and environments, including homes, cities, services, industry and commercial enterprises [38]. This, makes IoT one of the most important trends in the field of computer science. The Internet of Things (IoT) has grown in recent years to a huge branch of research: RFID, sensors and actuators as typical IoT devices are increasingly used as resources integrated into new value added applications of the Future Internet and are intelligently combined using standardised software services [25]. These devices are typically very limited in terms of CPU, memory, and power resources. Joining the fact that IoT are mostly designed for very specific applications, and must be able to self-configure in large-scale sensor applications, makes the task of programming this type of networks very challenging [9]. Currently exists a considerable number of programming languages for WSN providing distinct levels of hardware and network awareness. Some of them force the user to program basically at the hardware level, but others use sophisticated compilers to abstract away from the sensor devices and from the network infra-structure. Nevertheless, all of them fail in one important aspect: none is based on a model for describing computations in sensor networks allowing properties of the language, its runtime and applications to be demonstrated.

1.3 Objectives

The main goal is the implementation of a framework based on a Model-driven architecture (a software design approach for the developing of software systems) and software tool that will act as code generator.
Thus, using a traditional approach, the actual implementation, of a business process is typically performed by software developers, this poses two major problems. Firstly, the implementation usually deviates from the intentions and expectations of the business analysts because cross-domain communication is prone to misunderstandings. This deviation typically renders the respective business process less effective since the technology implementations should follow the business process and not vice versa. Secondly, changing the model of a business process necessitates the adaption or even recreation of parts of the implementation. The main idea, in relation of modeling, is then, to define an approach based on models, for close the gap between the business process and the IoT behaviour. The software tool in this sense, it's an important part of the framework, since, it can be viewed as a meeting point between the two examined reality, since it creates a starting point for the programmer in relation of how the BPMN has been made; in this way the possible misunderstanding that could happen with a tradition approach is significantly reduced. Since the software use to create code, it represent also a reduction of the time spent by the programmer for the implementation, since a part or the whole code it's automatically provided from the tool. The work, provide then a framework based on, a model-driven approach for represent IoT device in a BPMN and on a compiler that will translate the defined model in code that can be used by the programmer: it's obviously recommended that the generated code will be revisioned, since there's the possibility that the code that want to be developed has to be different in some part from the auto-generated code.
In the next chapters, will take place an overview of the related work, that includes the study of different academic articles used for create an overview of the problems. After that, there is a chapter that includes the adopted methodologies and implementation followed. This is followed by a chapter regarding the usage of the software and the supported possibility. At least it's placed a description of the tests made and the conclusions.

2. Related Work

In this chapter is explained all the related work that regards the first part of the paper. As explained above, the first part of the work concern a study of different academic articles, that allow to lead to a concrete solution for the problem. Those academic articles, regard the BPMN in general, and the IoT, and from those, a lot of useful information has been obtained.
The problem proposed in the introduction or a part of that, has been confronted by numerous contributions, in particular, in the following lines is written a summary of the work studied; in particular, the problem has been evaluated and divided in issue the will be analyzed.

2.1 Modeling IoT scenarios

The first issue that should be solved, regards the technologies to use for the model-driven approach; Business processes can be modeled by many methods and techniques. According to [3] the most commonly used notation standards are BPMN, eEPC and UML.
In general, has been decided to work using BPMN, because as [2] confirm, the BPMN standard, it's very commonly used in the process managing scenario, and also, because it's appropriate for the goal that has to been reached, since we have the concept of tasks, events, ecc...
Other models, like UML has been discarded,since it requires multiple models which cover different aspects and different involved entities of the business process, meanwhile, using BPMN approach, it's possible to represent all the aspects of the business process using a single model.

2.2 How To Represent IoT in Model

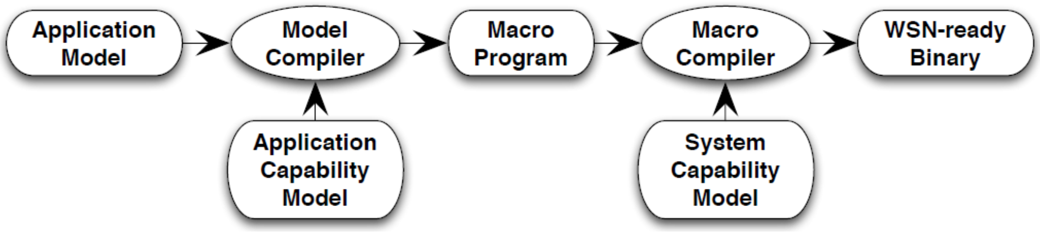
The second part of the study was aimed to reduce the gap between IoT devices and BPMN, since current approaches typi- cally consider the WSN as a stand-alone system. As such, the integration between the WSN and the back-end infrastructure of business processes is left to application developers. Unfortunately, such an integration requires considerable effort and significant expertise spanning from traditional information systems down to low-level system details of WSN devices.
As proposed by [4] a possible solution it's to define the process layer of IoT applications and enact them through a process engine; for realizing that, the authors, proposed to use already existing technologies like: Bosch IoT Things service (https://www.bosch-iot-suite.com/things/) , Unicorn, Griphon and Chimera.
In our case by the way, we aim to reduce the gap already in the model part. For doing it, different type of solutions, has been evaluated;
in general it's possible to divide those approach in two families:

- BPMN model only
- BPMN extensions

With BPMN model only is intended the fact that the BPMN itself should not be extended. Current approaches allow modellers to define both business processes and IoT devices behaviour at the same level of abstraction, using, for instance, BPMN-based approaches. BPMN already provides the concepts to define the behaviour of various participants, by using different pools. In this kind of approach, anyway, the compiler can't have all the information for deriving a correct and complete code; so, there's the necessity to modify the BPMN traditional structure or to use another type of model for adding the missing information.
For example in [7] has been produced a work, in which the business process is transformed in "Callas", and the interaction amongst participants is specified through collaboration diagrams. Supporting the execution of these hybrid processes requires bridging the gap between high-level BPMN and the programming code that IoT devices can execute. These approaches use a three-step procedure:

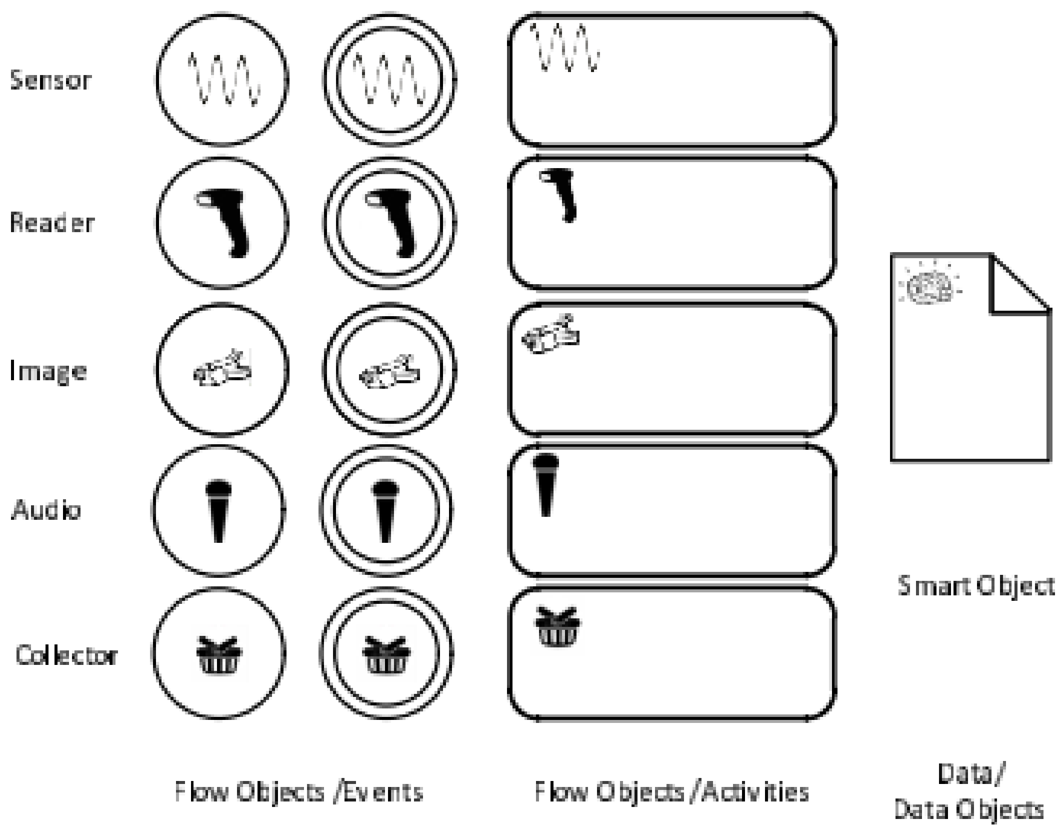
(1) translation of the process model to a Wireless Sensor Network (WSN) neutral intermediate language; (2) translation of the intermediate code to a platform specific executable code; and (3) deployment of the executable code into IoT devices. Also in [8] the model has been translated in “Callas”. The difference between this paper and the previous, is the fact that here, the gap between IoT and business process has been deleted using the BPMN resource class to integrate the information about IoT devices into the model, and has been used the BPMN performer class to define the IoT devices that will be participants of the process.
In [12] 's BPMN, each pool contains a task which uses sub-tasks. Such a sub-task can be refined by an additional model or can be a basic task which maps to an API function of the target platform. Sub-process are used when there's the necessity to model the internal details of work in a lower-level process diagram. In the end, every sequence flow results in a sequence of basic tasks being called. The set of all basic tasks is part of the platform API which the software developers implement once and reuse for all business processes which involve the given target platform. In this case, The core of the work is a compiler which translates models into executable. The translation of association flows is straightforward. Data items become variables and access to data stores is mapped to calls of persistence functions provided by the platform API. This translation is correct as long as no task is ever invoked more than once at the same time. This is called “static mode” because it provides a one-to-one mapping between tasks and its enclosed data objects. In contrast, in the “dynamic mode” there are parallel tasks which are invoked more than once at the same time. Then each invocation of such a task needs its own set of data objects. Thus, in the generated code, the corresponding variables are dynamically instantiated with each invocation and access to them is performed indirectly. It is undecidable in general if a task is parallel or not. However, it is possible to identify most non-parallel tasks by a simple sequence flow analysis, or otherwise the user might be able to assist by annotating tasks as non-parallel which is a property that can be checked at runtime by the generated code. Since the execution order and the successful execution of the activities composing the process have to be checked to automate and keep track of business processes, organizations deploy [33], another advantage of this approach is the fact that it gives a meeting-point (subtasks) from project manager to software engineer.

As [6] explains, BPMN provides an extension by addition mechanism that ensures the validity of the BPMN core elements while specifying additional constructs. The modeling language provides a set of generic business process elements, independent from a specific domain and it is often necessary to extend BPMN with custom concepts in order to represent characteristics of a particular vertical domain. BPMN is one of very few modeling languages that provides a set of generic extension elements within its meta model. In particular, the extensions of a BPMN can performs using Extension points or external schemes: in the first case, a standard extension points provided by BPMN engine vendors is used (Aktiviti, jBPM, etc...); meanwhile in the second case, there's the necessity of creating file with extension ".xsd", in which will be defined the desired proprieties. This kind of mechanism after all, is often performed introducing other type of models in the business process, like for example a class diagram or something like that; this means that in the proposed scenario, the project manager that will manage the BP, should also learn other type of models. An example of this type of approach has been proposed by [11]: they provided a unified programming framework and a compilation chain that, from high-level business process specifications, generates code ready for deployment on WSN nodes called "makeSense". As it shown in Figure 2.1, the model compiler takes as input the application model and an application capability model. The latter is a coarse-grained description of the WSN, providing information such as the type of sensors/actuators available and their operations. The model compiler translates these descriptions into a program written in a macro-programming language. By leveraging the system capability model, the macro compiler can generate different code for different nodes, based on their application role.



Picture 2.1: Compiling business process models into WSN-executable code.

Another approach that aim to the representation of IoT in BPMN using extension, has been proposed by [18] and [19]; in particular, their work, regards, the issue of modeling ubiquitous business processes. Ubiquitous computing denotes the third generation of modern computing where one person owns and operates multiple computers. In details, a ubiquitous business process is a location-independent business process that turns its business environment into a source of data and/or a target of outcome with the least of human interventions [20]. For solving the well-known issue, they propose an extension called Ubiquitous Business Process Model and Notation (or uBPMN). The extension is conservative because everything true about BPMN v2.0 persists. From a modeling standpoint, uBPMN was introduced to represent the recent ubicomp input technologies: These are (as shown in 2.2)sensors that quantify the physical data (e.g., temperature, position), smart readers that read data represented in standardized fashion (e.g., bar-code, NFC), cameras, microphones and collectors that gather information from remote or local files (e.g., Cloud, Internet Packets) or proxy devices.



Picture 2.2: uBPMN additional modeling elements.

UBPMN, also provides other feature/characteristics that can be used to help embrace ubicomp within the process flow: Automatic Identification and Data

Capture (or AIDC), Context-awareness (is the ability of a system to rapidly adapt itself to the context of an entity [21]), Augmented reality's(It allows the real-time fusion of physical objects with virtual ones [22]), sustainability (the ability to maintain a certain level in the long run) and ambient intelligence that brings intelligence to the everyday environments and makes IoT responsive to user interaction [23].

2.3 Model translation

After the evaluation of the modelling part, there's also the issue regarding the transformation of the BPMN that it's assumed to have all the information needed into an executable code. The first possible approach, has been proposed by [8] & [7]: In particular, the idea is to generate an abstract syntax tree from the BPMN's XML, after that, this AST, will be traverse multiple times for validation and in the end, the tree is translated into callas, in particular:

- Event elements: message received start events are translated into function calls and the process triggered by these events are translated into function definitions.
- Activity elements: send and receive tasks have a direct correspondence with the send and receive constructs from the language.
- Gateway elements: Exclusive gateways are represented by if statements.
- Connecting object elements: sequence flows are modelled by Callas control flow mechanisms, which is sequential composition, branching, looping, and function calls. The
- Data elements: data objects and their associations are modelled by Callas program variables that store objects and,

The second possibility that has been evaluated, take as example the paper [11]: in particular, the model compiler takes as input the application model and an application capability model(another model). The latter is a coarse-grained description of the WSN, providing information such as the type of sensors/actuators available and their operations. The model compiler translates these descriptions into a program written in a macro-programming language. The macro compiler takes as input the macro-program generated by the model compiler and a system capability model. The latter provides finer-grained information on the deployment environment (e.g., how many sensors of a given type are deployed at a location). The macro-compiler generates executable code that relies only on the basic functionality provided by the run-time support available on the target nodes. The third possibility is proposed by [12], in details, the authors defined a compiler, and two different ways for compiling: The static mode and the dynamic

mode. In the static mode, the translation of association flows is straightforward. Data items become variables and access to data stores is mapped to calls of persistence functions provided by the platform API. This translation is correct as long as no task is ever invoked more than once at the same time. In contrast, in the dynamic mode there are parallel tasks which are invoked more than once at the same time. Then each invocation of such a task needs its own set of data objects. Thus, in the generated code, the corresponding variables are dynamically instantiated with each invocation and access to them is performed indirectly. It is undecidable in general if a task is parallel or not. However, it is possible to identify most non-parallel tasks by a simple sequence flow analysis. For the undecidable cases the dynamic mode is always safe but it might be needlessly inefficient. Another approach that has not been studied in papers regards the use of ANTLR parse generator, in general, the idea, is to define a grammar and some rules that will be referred to BPMN's syntax, and using that for generate an AST. ANTLR is a public-domain parser generator that combines the flexibility of hand-coded parsing with the convenience of a parser generator, ANTLR has many features that make it easier to use than other language tools. Most important, ANTLR provides predicates which let the programmer systematically direct the parse via arbitrary expressions using semantic and syntactic context; in practice, the use of predicates eliminates the need to hand-tweak the ANTLR output, even for difficult parsing problem [24]. There's also the necessity to evaluate how to translate the model, or better, which is the target language in which the model should be translated, in fact, we can abstract the IoT devices hardware, using Callas, or it's possible to make the user decide every time, in which device should the code be executed.
Another important decision for our compilers regard the language in which the BPMN should be translated, in general, from the papers proposed in the references, there are two possible solution:

- Callas
- Specific target language

Callas sensor programming language can be an alternative to the target platform-specific languages, since it can be executed in every IoT device for which there is a Callas virtual machine available. This way, it's possible to abstract hardware specifications and make executable code portable among IoT devices from different manufacturers. This programming language is based on a formal model that allows properties like type-safety and soundness of the runtime system to be proved. This translates into a safety assurance for applications in the sense that common runtime errors can be prevented by design or statically at compile time. The language uses a custom byte-code format and runs it on a virtual machine created for this purpose. The second alternative is to translate the model into a target specific code for a selected device, for example using a GUI or at console, the user can be able to select where the selected model should be executed.

3. Methodology

It's possible to see the framework introduced in section 1.3 as a stack composed from three different level of abstraction, where, in order, the modeling it's higher level, the additional information given for the code generation it's the intermediate layer and the result code it's the lowest level layer. In particular, in this chapter, taking in exam the stack, different aspects of the work are evaluated, where, starting from the highest level to the lower one, are evaluated the possible issues and solutions that has been encountered.
In the next sections, are evaluated all the issues that has been encountered during the work, and the chooses that has been made for the possible problems/situations

3.1 BPMN and modeling environments

The first goal that has to be reach, it's the definition of a model driven approach, for representing the IoT using the business process model notation.
For this reason, an important necessity was to choose an appropriate BPMN software, in this sense, two software has been evaluated:

- Visual Paradigm
- Camunda modeler

Visual Paradigm is a world-wide leading award-winning enterprise management and software development suite, which provides all the features needed for enterprise architecture, project management, software development and team collaboration in a one-stop-shop solution [42]. Both of those tools are valid for work with the Business Process Modeling Notation and both of them have pros and cons.
Visual Paradigm offers a more complex and complete Gui, Camunda is simpler

and more intuitive.
Using one of the two software, users can interact with a shared repository: in case of visual paradigm, it's possible to upload datas on a repository that works in a similar way as GitHub; on the other hands Camunda's file can be uploaded on website called "Cawemo" that can be shared with other registered user making part of the team, and all the members that can view the project, can download, modify and interact with the project and the BPMN that are part of it.
In visual paradigm, the XML can be exported from the model, in any moment into an XML file, in a similar way, Camunda, automatically generate XML that can be checked in every moment just switching tab in the bottom part of the software, so meanwhile the BPMN is generated, it's possible to check along the way, and imagine the structure of the grammar that has to be implemented. Anyway, the XML exported from VP, results more chaotic, long and confusing that the one produced by camunda. Since as explained in the next chapters, the XML needs to be parsed and red somehow for working on it, the choose between those two proposed software has been Camunda.

3.2 Represent IOT using BPMN

Once the software for representing the model has been decided, the next step consist of find an approach for solving the issue regarding the IoT representation into the BPMN diagram. This issue born from the fact that the Business Process Modeling and Notation it's an high level overview of the business process, meanwhile, the IoT for being corrected represented needs to be express in a low-level environment. Since there's not an unique way for solving the problem, seeing as, all the solutions has pros and cons, the idea it's to consider all of them valid, and then, give to the Modeler the choose about which approach it's more adapt in relation of the situation which the BPMN takes place. The first proposed approach, that it's also the more efficient, since it gives a better logical distinction between the diagram's abstraction level, consist of represent the BPMN in a traditional way, and then, whether tasks that regards IOT are present, represent them using a call activity linking to another business model; this business model will be lower level representation of the IoT environment where the modeler can specify information regarding the devices. Using this approach, the main BPMN can still be for his original purpose that anyway will be enriched by an optional BPMN that can be used by the software tool and the programmer.
The second approach, that it's similar to the first one consist of represent the BPMN using the traditional method as well as explained in the first solution, but instead of representing the IoT information in a separated BPMN, the idea it's to define in the same file/diagram a sub-process in which will be contained the information regarding the low-level environment. In this case by the way, if there's the necessity to show the BPMN to a client or a final user that doesn't know the IoT details, it can results a little bit confusing.
The third and last proposed approach, consist of represent the IoT at the same level of abstraction of the main BPMN, in this case by the way, the result diagram can be very confusing. In conclusion, all the three method are valid and works with the software tools, but in general it's recommended to use first approach, since in sense of logical distinction and layout it's the less confusing. By the way, in relation of the situation that the BPMN represent the modeler can choose free one of the proposed approach.

3.3 Specify Additional Information

Once one of the proposed solution for solving the issue regarding the IoT representation has been chosen, the next problem regards the fact that, since the BPMN it's an high level overview, there's no a traditional approach for representing the additional low-level information needed in an IoT environment for the code generation. For solving the issue , several solutions could have been adopted, like for example the definition of an extension with the goal of storing those data; but since the main idea it's to create something easy to use, the best way for include the information it's use one of the fields already present in the BPMN elements. The final choose solution it's then, to write the data into field documentation, in case of a task and use the field "condition-expression" in case of gateway. At least, the information that has to be written needs to have a specified format, for permits the reading by the software tool. Because of it, has been specified an internal grammar into the main one, it's used for define the information format in every case. This language it's explained in the chapter 5.

3.4 Convert BPMN to Code

In this section are evaluated the different solution for reading and work with the XML provided by camunda. The easiest way for reading a file and working with it is to read it using a common programming language like java and then working with it. In this sense, by the way, the rest of the implementation would become pretty hard to manage, since, with this approach, the content of the file cannot be managed in a easy way.
The second approach regard the implementation of a compiler. In this sense, we need to define a language that can express the various construct provided by the BPMN and camunda. We will also need some other information, regarding data that cannot be defined with just a standard BPMN. In the next few rows, some theoretical concepts are clarified.
In computing, a compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). A compiler is likely to perform many or all of the following operations: preprocessing, lexical analysis, parsing, semantic analysis (syntax-directed translation), conversion of input programs to an intermediate representation, code optimization and code generation. A preprocessor is a program that processes its input data to produce output that is used as input to another program, in our case the preprocessor is Camunda that will produce the XML.
Lexical analysis, lexing or tokenization is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens (strings with an assigned and thus identified meaning); lexer use to work with a parser that, analyzing a string of symbols, either in natural language, will compute languages or data structures, conforming to the rules of a formal grammar.
Semantic analysis or context sensitive analysis is a process in compiler construction, usually after parsing, to gather necessary semantic information from the source code. It usually includes type checking, or makes sure a variable is declared before use which is impossible to describe in the extended Backus–Naur form and thus not easily detected during parsing.
A compiler is obviously based on a language, that, in turn is described by a grammar. The latter, describes how to form strings from a language's alphabet that

are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context—only their form. A formal grammar is defined as a set of production rules for strings in a formal language.

3.5 Used Technologies

3.5.1 Xtext

Xtext is an open-source software framework for developing programming languages and domain-specific languages (DSLs). Unlike standard parser generators, Xtext generates not only a parser, but also a class model for the abstract syntax tree, as well as providing a fully featured, customizable Eclipse-based IDE. To specify a language, the developer has to write a grammar in Xtext's grammar language. This grammar describes how an Ecore model is derived from a textual notation. From that definition, a code generator derives an ANTLR parser and the classes for the object model. Both can be used independently of Eclipse, in fact, for testing the parse-tree correctness another tools called "antlr-works" has been used. The Xtext parser in this sense it's a powerful tool because allow the programmer to write and traverse tree using an higher level of abstraction then the other parsing software. In Xtext, can be specified a grammar written in an "higher-level" than ANTLR; the Xtext grammar, will automatically transformed to an ANTLR one, this allow the programmer to manage and write a grammar in an easy way. In the following Figure it's showed a part of the Xtext grammar written in ANTLR.
In computer-based language recognition, ANTLR (pronounced antler), or Another Tool for Language Recognition, is a parser generator that uses LL(*) for parsing. ANTLR takes as input a grammar that specifies a language and generates as output source code for a recognizer of that language. Indeed other than lexers and parses, ANTLR can be used to generate parse trees. These parse trees are unique to ANTLR and help processifng abstract syntax trees.
A parse tree or parsing tree or derivation tree or concrete syntax tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar. Parse trees are usually constructed based on either the constituency relation of constituency grammars (phrase structure grammars) or the dependency relation of dependency grammars. Parse trees may be generated for sentences in natural languages (see natural language processing), as well as during processing of computer languages, such as programming languages.

3.5.2 Starting with Xtext

Since there's not a complete and easy guide that can help to start using Xtext, in this section, is given a fast overview of it. For installing Xtext, the following steps needs to be followed [43]:

- Open the Eclise IDE
- Help → Install new Software → Add → Download from: http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/
- Select the Xtext SDK from the category Xtext and complete the wizard by clicking the Next button until you can click Finish.
- After a quick download and a restart of Eclipse, Xtext is ready to use

After the installation, in case there's the necessity of importing a project, follow those steps:

- Download The project
- Copy the project into the work-space
- Import The Xtext project in eclipse: File → Import → General → Existing Project into workspace.

The project imported will be now correctly imported and usable within eclipse ide.
In case a new project needs to be created, follow those steps: Use the Eclipse wizard to do so:

- File → New → Project... → Xtext → Xtext project.
- Choose a meaningful project name, language name and file extension.
- Click on Finish to create the project.

3.5.3 Xtend

Xtend is a general-purpose high-level programming language for the Java Virtual Machine. Syntactically and semantically Xtend has its roots in the Java programming language but focuses on a more concise syntax and some additional functionality such as type inference, extension methods, and operator overloading. Being primarily an object-oriented language, it also integrates features known from functional programming like lambda expressions, in fact, it works well with Java 8 APIs and successors, as it does the same kind of target typing coercion for lambdas. Xtend is statically typed and uses Java's type system without modifications. It is compiled to Java code and thereby seamlessly integrates with all existing Java libraries. Unlike other JVM languages Xtend has zero interoperability issues with Java: Everything written interacts with Java exactly as expected. At the same time Xtend is much more concise, readable and expressive. Xtend's small library is just a thin layer that provides useful utilities and extensions on top of the Java Development Kit (JDK).
Of course, it's possible to call Xtend methods from Java, too, in a completely transparent way. Furthermore, Xtend provides a modern Eclipse-based IDE closely integrated with the Eclipse Java Development Tools (JDT), including features like call-hierarchies, rename refactoring, debugging and many more. The language Xtend and its IDE is developed as a project at Eclipse.org and participates in the annual Eclipse release train. The code is open source under the Eclipse Public License. Yet, the language can be compiled and run independently of the Eclipse platform. [44] It's at least, important to say, that, Xtend, like Java, is a statically typed language. In fact it completely supports Java's type system, including the primitive types like int or boolean, arrays and all the Java classes, interfaces, enums and annotations that reside on the class path.

4. Implementation

In this chapter it's shown how the implementation of the tools toke place, starting from the grammar definition, reaching the code trasformation

4.1 The grammar

In this section, is evaluated the grammar that has been written for work with the XML provided by Camunda. The explanation of the proposed grammar, is divided in three parts,in the first part is showed how Xtext use to manage the grammar and which are its main commands/constructs; in the second part, are evaluated the rules that allow the software to read the XML file, meanwhile in the second part, is showed the grammar representing a language that can allow the modeler/programmer (depending on the situation) to specify additional information for the IoT.

For explain how Xtext use to act, let's see an example:

Code 4.1: An example Grammar for an Xtext project

```
grammar org.example.domainmodel.Domainmodel with
                                          org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Domainmodel:
    (elements+=Type)*;

Type:
    DataType | Entity;

DataType:
    'datatype' name=ID;

Entity:
    'entity' name=ID ('extends' superType=[Entity])? '{'
        (features+=Feature)*
    '}';

Feature:
    (many?='many')? name=ID ':' type=[Type];
```

In the Figure above, an example grammar for a basic Xtext project is given, Let's have a more detailed look at what the different grammar rules mean:

1. The first rule in a grammar is always used as the start rule. It says that a Domainmodel contains an arbitrary number (*) of Types which are added (+=) to a feature called elements.

Code 4.2: Example of an Xtext grammar pt1

```
Domainmodel:
(elements+=Type)*;
```

2. The rule Type delegates to either the rule DataType or (|) the rule Entity.

Code 4.3: Example of an Xtext grammar pt2

```
Type:
DataType | Entity;
```

3. The rule DataType starts with a keyword 'datatype', followed by an identifier which is parsed by a rule called ID. The rule ID is defined in the super grammar org.eclipse.xtext.common.Terminals and parses a single word, a.k.a identifier. It's possible to navigate to the declaration by using F3 on the rule call. The value returned by the call to ID is assigned (=) to the feature name.

Code 4.4: Example of an Xtext grammar pt3

```
DataType:
'datatype' name=ID;
```

4. The rule Entity again starts with the definition of a keyword followed by a name. Next up there is the extends clause which is parenthesized and optional (?). Since the feature named superType is a cross reference (note the square brackets), the parser rule Entity is not called here, but only a single identifier (the ID-rule) is parsed. The actual Entity to assign to the superType reference is resolved during the linking phase. Finally between curly braces there can be any number of Features, which invokes the next rule.

Code 4.5: Example of an Xtext grammar pt4

```
Entity :
'entity' name=ID ('extends' superType=[Entity])? '{'
    (features+=Feature)*
'}';
```

5. The keyword many shall be used to model a multi-valued feature in this DSL. The assignment operator (?=) implies that the feature many is of type boolean.

Code 4.6: Example of an Xtext grammar pt5

```
 Feature:
(many?='many')? name=ID ':' type=[Type];
```

In the following part it's explained in detail the grammar defined for the XML reading:

1. The grammar start with those rules. In particular, the first rule it's just the start rule, in which it's specified that the grammar is used for read an XML file. The second rule, is used for specified in an high level of abstraction, the main components of an XML file. Those are, the prolog/intestation and the elements/tag that compose it.

   Code 4.7: XML grammar pt1

   ```
   Model:
       model += Xml
   ;

   Xml:
       {Xml}  prolog? elements+=element*
   ;
   ```

2. The prolog rule, indicates the syntax of a camunda XML header. In particular, a prolog is composed by: the terminal symbols "<?", a terminal rule called HEAD that is debated later, the terminal word "version=" followed by a string terminal type and the terminal word "encoding=" that, again is followed by a string.
   The elements rule, is used for specified the structure of an XML file after the default header. In practice, is specified, that an element (that is repeated from 0 to n time, since there's the '*' symbol in the previous rule) can be of two different types: open tag, which will must have content inside it followed by a close tag or a singleton tag (with singleton tag, is intended a tag that is open and closed without having "children").

   Code 4.8: XML grammar pt2

   ```
   prolog:
       "<?" HEAD "version=" STRING "encoding="STRING "?>"
    ;

    element      :   (open += Open
                 contents+=content  close_tag+=Close)
              |  singleton_tag+=Singleton
              ;
   ```

3. Let's see in detail how an open tag is defined: every open, starts with the symbol "<" followed by an HEAD terminal rule that in turn, is followed by a terminal symbol ":" and a keyword. After the header of the tag, there is a not specified number that goes from 0 to n of HEAD or HEAD followed by keywords, "=" and a string. of course an open tag ends with the symbol ">".

Code 4.9: XML grammar pt3

```
Open:

        ("<" head+= HEAD ":" keywords += KEYWORDS  ((head1 += HEAD
            ":") | ((head1 += HEAD|keywords1 += KEYWORDS) "="
            value+=STRING))* ">")

;
```

4. The content of an open tag it's a little bit more complex of the previous one. In practice, in the content we can have the specification of a type, that is explained later since it doesn't regard the XML structure itself, but is part of the additional language. Inside the content, we can have another element, this, implies that a recursive call to the rule element will be performed. Another possibility is to have a string, or body (a particular type of string) or a keyword. Since we can have more than one content and a content it's defined by one or more of the possibility above, all this rule is under the star symbol.

Code 4.10: XML grammar pt4

```
content:
    {content}
    ( type += ("_TASK"|"_GATEWAY") "{" codex += codex "}"|
        element+=element | body+=preconditions*  body+=(BODY
        |variables) body+= conditions*
        |    keywords+=KEYWORDS  |  data+=STRING
    )*
;
```

5. The Singleton rule has a syntax similar to the Open rule, but with less possibility. In a singleton rule, it's always present an HEAD and a keywords, followed by another HEAD or an HEAD followed by a keyword or a STRING. A singleton rule terminates with the "/>" symbol.
   The closure tag meanwhile, it's just composed just from "</" an HEAD and a keyword. The closure terminate with a ">" symbol.

Code 4.11: XML grammar pt5

```
    Singleton:
        {Singleton} ("<" HEAD ":" keywords+=KEYWORDS ((HEAD ":") |
            (keywords1+= KEYWORDS "=" value+=STRING))*)
        "/>"
    ;

    Close:

        {Close} ("</" HEAD ":" KEYWORDS ">")
    ;
```

6. In the end, let's see the terminal rules that composed the grammar.
   The first terminal rule, it's "HEAD", in this terminal rule, are defined the possible word that Camunda can use in the first part of a tag. The second terminal rule, it's "KEYWORDS", and as well as the previous one, indicated a list of words that can be used in a tag. The last rule, it's "BODY", this terminal rule, it's a particular type of string, that include also other symbols that are not inside the STRING default Xtext's terminal rule. In this sense a first intuition could have been to use the or operator "|", making the compiler choose between a string or a set of missing symbols, but this kind of things, are not recommended, since the compiler doesn't have back-tracking and this would have created an ambiguous grammar. Xtext allow the programmer to enable the back-tracking, but it's not recommended for optimization.
   The BODY rule it's a particular one, since, it has a return "ecore::EString": this kind of rule it's called "datatype rule".

Code 4.12: XML grammar pt6

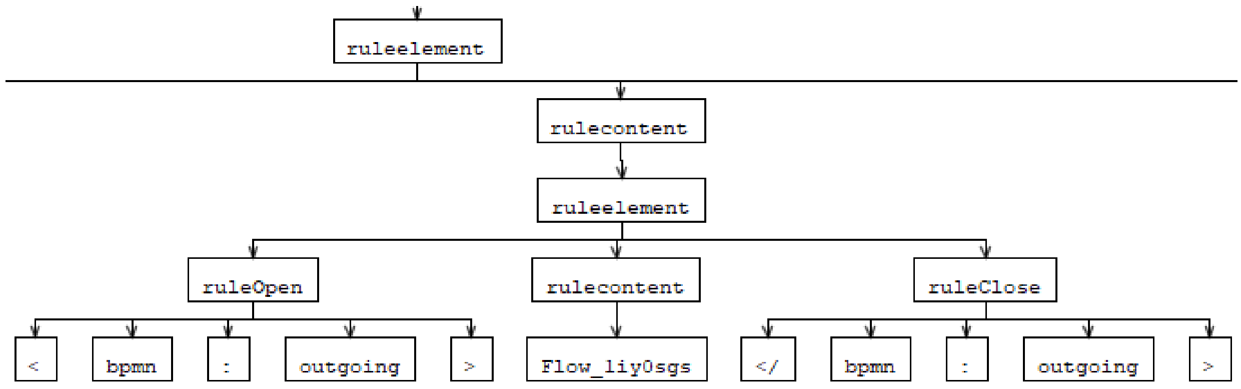
```
     terminal HEAD:
        ("bpmn"|"bpmndi"|"camunda"|"xsi"|"xml"|"xmlns"|"dc"|"di")
    ;

    terminal KEYWORDS: "id" | "name" |  "isExecutable" | "sourceRef"
        | "processRef" | "targetRef"
        | "calledElement" | "type" | "expression" | "value" |
          "resultVariable" | "asyncBefore" | "intermediateThrowEvent"
        | "class" | "event" | "startEvent"| "task" |
          "messageEventDefinition" | "sequenceFlow" | "isExpanded"
        | "condition" | "association" | "outgoing" | "serviceTask" |
          "process" | "standardLoopCharacteristics"
        | "incoming" | "intermediateCatchEvent" |
          "conditionalEventDefinition"|"isMarkerVisible" |
          "terminateEventDefinition"
        | "endEvent" | "textAnnotation" | "text" |
          "dataStoreReference" | "bpmnElement" |
          "dataObjectReference"
        | "callActivity" | "laneSet" | "lane" | "flowNodeRef"
          |"definitions" | "userTask" | "documentation"
        | "dataOutputAssociation" | "exclusiveGateway" | "waypoint" |
          "BPMNLabel" | "diagramRelationId"
        | "extensionElements" | "inputOutput" | "list" |
          "inputParameter" | "height" | "messageFlow" |
          "dataObjectRef"
        | "outputParameter" |  "properties" | "property" |
          "BPMNShape" | "Bounds" | "subProcess" | "cancelActivity"
        | "field" | "string" | "scriptTask" | "script" |"BPMNPlane" |
          "BPMNEdge" | "sendTask" | "boundaryEvent"
        | "executionListener" | "timerEventDefinition" |
          "timeDuration" | "width" | "dataInputAssociation"
        | "parallelGateway" | "collaboration" | "participant" |
          "targetNamespace" | "dataObject" | "signalEventDefinition"
        | "BPMNDiagram"     | "exporter" | "exporterVersion" | "x" |
          "y"| "isHorizontal" | "attachedToRef"
        | "conditionExpression" | "receiveTask" | "messageRef" |
          "initiator" | "message" | "inclusiveGateway";


    terminal BODY returns ecore::EString :
        ('a'..'z'|'A'..'Z'|'è'|'ò'|INT|'_'  )* ;
```

An important aspect of the grammar just explained regards the usage of particular constructs that are present sometimes in the rule. Those are the actions (an action has the form: action), that are used for specify explicitly the creation of returned EObject.
The last thing to say about the XML grammar just specified, regards the modality of how the words presents inside the terminal rules has been selected. In particular, different steps has been followed. Firstly has been tried to find all the possible alternative manually (trying manually with camunda to create all the possible type of constructs and add keywords after every test), but for abvious reasons, this approach was wrong, since there's an high possibility to miss some particular keywords making the parser unable to read some particular type of BPMN that maybe has not been considered. Since that, the possible keywords has been enrich adding the missing words founded in the BPMN documentation. Of course there's still the possibility of the error, since this step too has been made manually, but the set of keywords has been surely increased.
In this sense, another possibility could have been to not define the a list of words that the user can adopt in a BPMN, but this would have been that the compiler would have been unable to understand if inside a tag there are wrong keywords.

In the next two Figure are shown a part of the ANTLR grammar generated from Xtext and a part of the parse tree:

Code 4.13: Part of the generated ANTLR grammar

```
// Rule Model
ruleModel:
    ruleXml
;

// Rule Xml
ruleXml:
    ruleprolog?
    ruleelement
    *
;

// Rule prolog
ruleprolog:
    '<?'
    RULE_HEAD
    'version='
    RULE_STRING
    'encoding='
    RULE_STRING
    '?>'
;

// Rule element
ruleelement:
    (
        ruleOpen
        rulecontent
        ruleClose
            |
        ruleSingleton
    )
;
```



Picture 4.1: A part of the parse tree generated for an XML file.

In this last part of the section, is explained the the grammar regarding the language that is used for specify the additional data of the IoT that can't be defined in a standard BPMN diagram. As explained in the previous part, this particular grammar, will be triggered if it's written inside the content of a particular XML tag. The following rules, can be of two types: task or gateway; in the first case is specified what can be done in a particular activity, meanwhile in the second case, it's specified the conditions option of a gateway.

1. Codex rule, represents the starting rule of the embedded language. In this particular rule, it's possible to specify information regarding the device that will be used (Arduino ecc...) through the rule "device", the network protocol or the network protocols and the sensors if used.

Code 4.14: IoT grammar pt1

```
codex:
        ( device_code += device protocol += protocol* sensor_code
            += sensor* )
;
```

2. This rule, allow the user to specify the information regarding the IoT device that is intended to be used. In particular, the informations stored are the id of the device (in this sense the modeler can specify the usage of more than one arduino in the same business process) and the device name, that it's just the type of the device that is intended to be used.

Code 4.15: IoT grammar pt2

```
device:
    "DEVICE" ":" device += STRING
    "NAMEID" ":" id += STRING
;
```

3. This rule is used for reading the information regarding the network protocol that is intended to be used for the selected device. in particular, it's possible to specify which protocol has been choose just specifying it with the terminal words "HTTP" or "MQTT" followed by the information required.

Code 4.16: IoT grammar pt3

```
protocol:
    ((pname += ("MQTT") ) "{" (mqtt_data += mqtt_data
        mqtt_device += protocol_device) "}") |
    ((pname += ("HTTP") ) "{" (http_data += http_data
        http_device += protocol_device) "}")
;
```

4. Both rule are used for specifying detailed information regarding the selected network protocol. In the first case, is intended that in that particular task there's the necessity of using the http protocol. With this rules, the user can specify information that will be used for the code generation. In particular, in the first rule, since the selected protocol its the http, the information needed are: name of the protocol, server ip address, request type, content type, header, additional data and network information. In case of MQTT, other information are required: name of the protocol, broker user, broker password, broker, subtopics, pubtopics, additional data and network information. The fields above are explained in detail in the chapter 5.

Code 4.17: IoT grammar pt4

```
http_data:
    {http_data}
    ("NAME" "=" pname+=STRING | "SERVER_IP" "="
        server_ip+=STRING
        | "REQUEST_TYPE" "=" request_type+=STRING
        | "CONTENT_TYPE" "=" content_type += STRING
        | "HEADER" "=" header += STRING
        | "DATA" "=" data += STRING
        | "NETWORK" "{" mqtt_network_data +=
            protocol_network_data* "}"
    )*
;
mqtt_data:
    {mqtt_data}
    ("NAME" "=" pname+=STRING | "BROKER_USER" "="
        broker_user+=STRING
        | "BROKER_PASSWORD" "=" broker_password+=STRING |
            "BROKER" "=" broker += STRING
        | "NETWORK" "{" mqtt_network_data +=
            protocol_network_data* "}"
        | "SUBTOPICS" "{" ("TOPIC_NAME" "=" subtopics += STRING)*
            "}"
        | "PUBTOPICS" "{" ("TOPIC_NAME" "=" pubtopics += STRING*
        | "DATA" "=" value+=(STRING|variables))*"}"
    )*
;
```

5. This part of the grammar regards the network information and the protocol device. The first rule, regard the possible users and password necessary for connecting to the access point. The second rule, it's used for get information about the network shield that is will be used; in particular, it's required the name of the intended device.

Code 4.18: IoT grammar pt5

```
    protocol_network_data:
        "SSID" "=" ssid+=STRING
        "PASSWORD" "=" password+=STRING
    ;
    protocol_device:
        "PROTOCOL_DEVICE" "{" "NAME" "=" dname += STRING "}"
    ;
```

6. The last part of the grammar regarding the tasks, is used for specify the information regarding the possible sensors connected to the IoT device. The first rule specify the possible categories of the sensors that can be used; for every category it's possible to define the specified information, those are: the name of the sensor, the ID of the sensor and the pins which the sensor it's connected.

Code 4.19: IoT grammar pt6

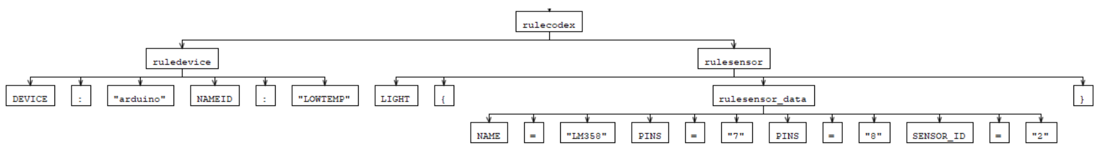
```
   sensor:
        (sname += "TEMPERATURE" "{" sensor += sensor_data "}")|
        (sname += "DISTANCE" "{" sensor += sensor_data "}")|
        (sname += "GAS" "{" sensor += sensor_data "}")|
        (sname += "LIGHT" "{" sensor += sensor_data "}")|
        (sname += "LED" "{" sensor += sensor_data "}")
    ;
    sensor_data:
        {sensor_data}("NAME" "=" pname+=STRING
            | "PINS" "=" pins+=STRING | "SENSOR_ID" "=" sensor_id +=
                STRING | "VALUE" "=" sensor_init_value += STRING
        )*
    ;
```

7. In relation of the conditions, the user can specify a condition following the syntax expressed in those rules. In practice it's possible to define a preconditions, that it's given by the "!" (not) symbol or the open bracket. The condition will then follow the standard syntax of a the java programming language. The words "&le", "&lt", "&ge", "&gt" and "&amp" are in order: less then (<), less then or equal (<=), grater then (>). grater than or equal (>=), &.

Code 4.20: IoT grammar pt7

```
    conditions:
        ("&lt;" | "&le;" | "&ge;" | "&gt;" | "=" | "&amp;" | "||" |
            ")" | "!=")
    ;
    preconditions:
        ("!"|"(")
    ;
```

Once a file with the corrected syntax is red, the part of the parse tree regarding the IoT information will look as follow:

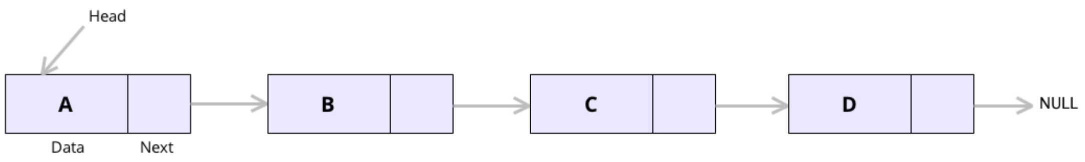


Picture 4.2: IoT grammar parse tree

This conclude the explanation of the grammar that has been adopt in the software for parsing the XML provided by Camunda plus the additional information used for generating code.
In the next part of the section it's explained how the information red from the XML are transformed/compiled into code, for this purpose has been used Xtend.

4.2 Code generation

In this section is explained how the parse tree created using the grammar explained in the previous section it's transformed into code compatible with IoT devices. For the transformation of the data into code compatible with the IoT, the used tool is Xtend. The first problem regarding the transformation, regards the order which the information needs to be compiled. This issue occurs, because Camunda use to generate the XML code dynamically, so when the user insert inside the BPMN any possible thing, the XML code regarding it, is inserted into the XML. This by the way, implies that if for example the user create a task, called for example "X" and, after the creation, he create another task "Y", that in a temporally line of reasoning occurs before "X", in the XML, "X" will be in the file, and as consequence in the parse tree before "Y".
This problem grows in relation of the complexity of the BPMN, since more complexity and as consequence more elements in the diagram, will produce more ambiguity.
For this reason, traversing the parse tree without any constriction will produce code that has no logic consequence; if for example the intentions it's to read from a sensor and after that, send the data using a supported protocol via a supported shield, the result could be wrong. It's possible that the result is wrong, indeed if the previous issue occurs, the device will first send the data and after that read from the sensor.



Picture 4.3: Figure of a generic list

A simple BPMN diagram, without particular branching, has a similar structure of a code list; where every task represent a node of the list and every arrow it's the point of the successor/successors. The idea in this case it's to traverse the parse tree multi time once, creating a list (since we are talking about java this will be an arraylist for comfort) and insert in it all the start event that will represent the head of the list or better the first node that occurs. For doing it, it's important to explain that the camunda's XML has inside it, data about the type of

elements, for example, a start event, can have the semantic: "<bpmn:startEvent id="StartEvent1">". In this case it's possible to recognize the start event from the first keyword. Every times, a tag with a syntax similar to the one just explained, that represent a start event, it's found during the traversing, it's inserted in the arraylist. later one it's showed and explained the function that allow the software to create the arraylist.
The method "FillEvent" will traverse all the parse tree one time. During the exploration, if a tag that represent a start event it's founded, the id of that particular element it's inserted as String in the arraylist. In case it's reached a start event that it's inside a sub-process it's not inserted in the starting list, since, it doesn't represent an initial starting point for the code generation.

Code 4.21: fillEvent method

```
//This method it's used for filling all the start events in the bpmn
def FillEvent(Resource r)
{
    for (Element : r.allContents.toIterable.filter(element))
    {
        for(Open : Element.open)
        {
            if (Open.keywords.get(0).equals("startEvent"))
            {
                if (InSubProcess(getID(Open),r) == false)
                {
                    start_events.add(getID(Open));
                }
            }
        }
    }
}
```

Once all the starting point of the BPMN are stored, the next step it's to start from the first start event, that in a line of reasoning represent the starting point, or the first pointer for the first element that needs to be transformed. From this point, still staying in the point of view of a simple BPMN diagram, for each head, it's needed to find the successors and store it in another arraylist, that represent the order which the information need to be take in care. The new arraylist, will be composed by the a list that represent the first "flow" of the business process concatenated with the remains "flows".
Let's consider now the case in which the diagram can be more complex, so when, for example we can have multiple "flows" that from an element goes into multiple elements. In this case, the previous logic it's still valid, but since there's the necessity to explore all the possible path for every element, the best solution in this case is the recursion. Applying the recursion it's possible to explore a particular path (the first that it's founded) until it ends (it ends when an end event occurs or when there's no successors) and then go back in a "back-tracking" way looking for every element if another successors is present. In this kind of situation, there's also the necessity to change the order of an element if it's repeated; in case in the BPMN it's present a gateway with three outgoing arrow that in the end, with a join will merge into a single path, the elements in it will be repeated three times. The solution for this issue it's simple, since the only thing to do it's remove the element present in position "N" and add the element again in the last index; in this way, it's possible to have for example more conditions nested or consecutive and the flow of the code will be correct. Let's see now the method written in Xtend that allow the software to create the structure representing the order of compiling for a BPMN. Since this method it's very long for obvious reasons, just a part of it it's reported.

The method starts with a research in the parse tree. In particular what it's performed it's that for every element (where an element can be a singleton tag, an open tag, a close tag or the content), are explored the keywords that represent the type of element on which the navigation has arrived. In case of the keyword it's "sequenceFlow" this means that this particular part of the parse tree represent an outgoing arrow. Now, there's the necessity to determine if that particular flow start from the current element or not. For doing it, it's performed a control of all the words that are stored in this particular tag, and if it's founded something like "sourceRef=id of this element", this means that all the ids into the "targetRef" are successors of this particular element. This operation it's performed visiting all the parse tree and as consequence all the BPMN. One a successor it's founded different controls are done. First of all it's checked if the successor it's a sub process or not; in a positive situation, will be called recursively the method for creating the arraylist containing the compiling order, since, the sub process can be view as a BPMN that start from this point and end at the next successor.

Code 4.22: fillSuccessors method pt1

```
for (Element : r.allContents.toIterable.filter(element))
{
    for(Singleton : Element.singleton_tag)
    {
        if (Singleton.keywords.get(0).equals("sequenceFlow"))
        {
            for(keywords : Singleton.keywords1)
            {
                if (keywords.equals("sourceRef"))
                {
                    if (Singleton.value.get(n).equals(my_id))
                    {
                        for(keywords1 : Singleton.keywords1)
                        {
                            if (keywords1.equals("targetRef"))
                            {
                                //In case the actual event it's a
                                    subprocess
                                if(isSubProcess(Singleton.value.get(j),r))
                                {
                                    getSubStartEvents(Singleton.value.get(j),r);
                                    var app = j;
                                    for (subStart :
                                        subStarts.get(subStarts.size()-1))
                                         fillSuccessors(subStart,r);
                                    j=app;
                                }
```

In case the founded element it's an inclusive condition or an exclusive condition, it's checked if this particular element it's a join gateway or not. In case it's not, nothing will happen, meanwhile if it's true, this means that in this part of the BPMN there's the end of a condition already opened before this element. The "false closure" number it's just used for indicate that a particular condition has already been closed and that there's no necessity to close again a condition when an end event it's reached. The "tread condition" number it's used in case there's a condition inside a thread.

Code 4.23: fillSuccessors method pt2

```
//If i have a gateway
if (getGatewayType(my_id, r).equals("exclusive_condition") ||
    getGatewayType(my_id, r).equals("inclusive_condition"))
{
    if (isForkGateway(my_id, r) && getCondition(Element).equals(""))
    {
        false_closure++;
        successors.add("end_condition");
        if (threadNumber > 0)
        {
            thread_conditions--;
        }
    }
}
```

When the element it's a condition written with the right syntax (The syntax of the language it's explained later), the operation that it's performed it's just add this particular condition into the arraylist. If a condition it's already open, this means that this condition it's an else condition (in case the element it's an inclusive gateway, the else condition is treated in the same manner of the else condition). If a condition it's in a loop, this means that in this particular situation there's the necessity to insert a loop, or better a while construct.

Code 4.24: fillSuccessors method pt3

```
//If there's a condition
if (!getCondition(Element).equals(""))
{
    if (threadNumber > 0)
    {
        thread_conditions++;
    }
    if (str2.equals(""))
    {
        str2+=getGatewayType(my_id,r)+"="+getCondition(Element);

    }
    else
    {
        if (false_closure > 0)
        {
            false_closure--;
        }
        else
            successors.add("end_condition");
        str2 = getGatewayType(my_id,r)+"="+getCondition(Element)+"_else";
    }
    //While statement in case i have a condition with loop
    if (hasLoop(my_id,r,str2))
    {
        setLoop(loop_variable);
        return;
    }
    else
    {
        successors.add(str2);
    }

}
```

In this part of the method it's managed the part regarding the parallel gateways. In particular, in case there's a parallel condition, it's performed a check in the same way of a normal gateway, so, it's checked if this particular gateway it's a join gateway or not. In case it's not a join gateway, the number of the opened thread will be incremented and a "parallel condition" it's inserted. In the opposite manner, an "end parallel condition" it's inserted.

Code 4.25: fillSuccessors method pt4

```
if (!hasLoop(my_id,r,""))
{
    //Check if i have a parallel condition
    if (getGatewayType(my_id,r).equals("parallel_condition"))
    {
        if (isForkParallel(my_id,r))
        {
            if (thread_conditions <= 0)
            {
                successors.add("end_parallel");
                str3 = "";
                fork = true;
                threadNumber--;
            }
        }
        else
        {
            if (str3.equals("") || fork)
            {
                fork = false;
                str3 = "parallel_condition";
                successors.add(str3);
                threadNumber++;
            }
            else
            {
                successors.add("end_parallel");
                successors.add(str3);
                threadNumber--;
            }
        }
    }
```

In this part of the method, if the next element it's already into the arraylist, means that this part of the BPMN has already been explored, otherwise, the element it's inserted into the arraylist, and the method it's called recursively.

Code 4.26: fillSuccessors method pt5

```
//Repeated values
if (!successors.contains(Singleton.value.get(j)) ||
    getGatewayType(Singleton.value.get(j),r).contains("_condition"))
{

    successors.add(Singleton.value.get(j));
    fillSuccessors(Singleton.value.get(j),r);
}
```

Those operations are performed also for the opening tag, in a similar way, but since this would have been a repetition, of the explanation just given, this part of the code will not be reported. At the end of the method, so when the array-list it's completely defined, different other methods will be called for adjusting some order mistakes in case the BPMN has a particular structure, or for removing some not appropriate elements. Once this procedure and the various "adjusting methods" have been called, the next step it's the elimination of the not needed things inside the arraylist. With not need things it's intended the tasks or object that are part of the BPMN but that doesn't regards the IoT or the sensing. For doing that, the arraylist just created it's explored and the usable things are inserted into a data structure composed by object of type Elements, from which the different type of tasks, gateways, conditions ecc... will inherit. In this last data structure that it's an arraylist too, will be inserted all the possible object that can be converted to code and also the data regarding the language specifying the possible things that has to be performed in that particular point of the flow. For every kind of data that regards the IoT something it's performed: In case of parallel condition an instance of the Parallel class it's created, in case of gateway condition an instance of the Condition class it's created.

Code 4.27: setDatas method pt1

```java
if (successor_id.contains("parallel_condition"))
{
    thread = new Parallel();
    elements.add(thread);
}
if (successor_id.contains("end_parallel"))
{
    thread = new Parallel(true);
    elements.add(thread);
}
if (successor_id.contains("condition="))
{
    conditions++;
    cond = new Condition (successor_id);
    opened_conditions.add(cond.getId());
    elements.add(cond);
    return;
}
if (successor_id.equals("end_condition") && opened_conditions.size() > 0)
{
    cond = new Condition(false,
        opened_conditions.get(opened_conditions.size()-1));
    opened_conditions.remove(opened_conditions.size()-1);
    elements.add(cond);
}
if (successor_id.equals("end_condition_end") )
{
    cond = new Condition(true,
        opened_conditions.get(opened_conditions.size()-1));
    opened_conditions.remove(opened_conditions.size()-1);
    elements.add(cond);
}
```

When more complex data that doesn't determine the order of execution it's found,more complex operation are performed. In this example, it's shown a part of the case in which in a particular task, has been decided to use the HTTP protocol for sending some data. In this particular case, an instance of the class HTTP it's created and every possible attribute if given it's used during the initialization of it. After the initialization that will change in every case it's different, because, in relation of what it's intended to do a different object with different attribute it's created.

Code 4.28: setDatas method pt2

```
netdata1 = new HTTP();
elements.add(netdata1);
netdata1.setType("http-get");
netdata1.setName(getName(Element));
for(Device : Codex.device_code)
{
    netdata1.getDatas().setDevice(Device.device.get(0));
    cpp_gen.setDevice(Device.device.get(0));
    netdata1.setId(Device.id.get(0));
}
for(http : Protocol.http_data)
{
    h_gen.setNetwork_protocol(http.pname.get(0).toLowerCase().replaceAll("\\s+",""));
    cpp_gen.setNetwork_protocol(http.pname.get(0).toLowerCase().replaceAll("\\s+",""));
    netdata1.getDatas().setServer_ip(http.server_ip.get(0));
    netdata1.getDatas().wifi_ssid.clear();
    netdata1.getDatas().wifi_pass.clear();
    for(http_network_data : http.mqtt_network_data)
    {

        netdata1.getDatas().wifi_ssid.add(http_network_data.ssid.get(0))
        netdata1.getDatas().wifi_pass.add(http_network_data.password.get(0))
    }
    if
        (http.request_type.get(0).replaceAll("\\s+","").toLowerCase().equals("post"))
    {
        netdata1.setType("http-post");
        netdata1.getDatas().setContent_type(http.content_type.get(0))
        netdata1.getDatas().setHeader(http.header.get(0));
        netdata1.getDatas().getDatas().clear();
        for (data : http.data)
            netdata1.getDatas().getDatas().add(data);
    }
```

Once this last data structure it's ready with all the information needed for generating code, it's performed the operation of generation. This operation will create from 3 to N files in relation of how many devices are used (this because if there's for example four arduinos that makes different operation, every device must have his code). The firstly thing that will be generated it's a library that contains all the method that allow the device to work. This library it's composed by two elements, where, one it's the ".h" file, or better the header of the library, which will contains all the methods and information regarding the library. The second element that it's generated it's the ".cpp" file, or better the source file, in which all the required methods are created depending on the situation. Both files are global, in the sense that, both of the are generated just one time and will be usable just importing them by all the arduino that are in the BPMN. Of course, those file are not static, but will be generated in function of what it's specified inside the source diagram (if for example in a task it's written that the intention it's to use a DHT22 temperature sensor, all the methods that allow the user to work with it will be generated). The main file, it's pretty different. In this case, as mentioned above, for every single device an ".ino" file it's created. For all the three possible solution a different class will produce code; let's see in detail how the main part of the code it's produced.
The following class will produce three different types of code: the first part it's the part regarding the variables that are needed for a particular sensor or network protocol. In the following code, there's the algorithm used for generating variables for a gas sensor: in this situation, the first variables need it's a float in which the red result will be stored, meanwhile, the rest of the variables are the pins which the sensor it's connected to. The variables are unique, since in case, for a particular sensor the "result" variables are created with a static name, plus the sensor id that has to be unique for every sensor, so, it's impossible to have two variables in which it's stored the result having the same name. Something similar it's done with the pins, but in addition to the information above, since a sensor can be connected on more than one pin, the variable name it's enriched with the pin that it's a unique value.

Code 4.29: .ino file code generation pt1

```
if (elements.get(n).getType().equals("mq9"))
{
    GasSensor app = (GasSensor) elements.get(n);

    if (!sens_variables.contains("float gas"+ app.getSensorId()))
        sens_variables += "float gas"+app.getSensorId()+"; //Stores
            value\r\n";
    for (int y = 0; y < app.getPins().size() ; y++)
    {
        if (!sens_variables.contains("int pin"+ app.getSensorId() + "= "
            + app.getPins().get(y)))
            sens_variables += "int pin"+ app.getSensorId() +
                app.getPins().get(y) + "= " + app.getPins().get(y)+";\n";
    }

}
```

The following piece of code, it's a part of the method that will produce code for a particular sensor into the setup part. In particular, it's shown the case in which in the task, it's intended to use a dht22 temperature sensor. In this case, the code which it's used for initialize the sensor it's invoked. The method "getVariableName()" will return the name of a variable that has been declared before given its value.

Code 4.30: .ino file code generation pt2

```
if (elements.get(n).getType().equals("dht22"))
{
    if (elements.get(n).getId().equals(elements.get(i).getId()))
    {
        TemperatureSensor app = (TemperatureSensor) elements.get(n);
        setup_code+="\tSerial.begin(9600);\n";
        setup_code+="\tInitDHT22("+getIntVariableName(app.getPins().get(0))+");\n";

    }
}
```

In this part of the code, it's shown the process that it's used for writing code into the loop part in case the specified sensor it's a dht22 temperature sensor. The first part of the produced code it's used for reading the temperature and the humidity from the sensor. After that, a list of tabulation it's inserted into the final result, in relation of the rest of the code. Finally the result it's inserted inside the loop variable that represent the data that will be written into the arduino's loop or in case there's a task opened, this part of code will take place inside the task.

Code 4.31: .ino file code generation pt3

```
if (elements.get(n).getType().equals("dht22"))
{
    TemperatureSensor app = (TemperatureSensor) elements.get(n);
    temp+= "\tdelay(2000);\r\n"
            + "\t//Read data and store it to variables\r\n"
            + "\thum"+app.getSensorId()+"= dht.readHumidity();\r\n"
            + "\ttemp"+app.getSensorId()+"= dht.readTemperature();\n";
    for (int k = 0; k < tabulations;k++)
    {
        temp = temp.replaceAll("(?m)^", "\t");
    }
    if (opened_threads.size() > 0)
        opened_threads.get(opened_threads.size()-1).addBody(temp);
    else
        loop_code+= temp;
    temp = "";
}
```

After the creation of all the code needed for this particular arduino, the final code will be inserted into an arraylist that will contains in every position the complete code generated for a device. The final operation performed it's the creation of the files with their content. In this case Xtend makes a very good job, rendering the creation and writing very easy to do. In practice, with a method of the class “IFileSystemAccess2”, called “generateFile”, there's just the necessity to specify the file name and the content of it that after the process just explained will be ready to be written.

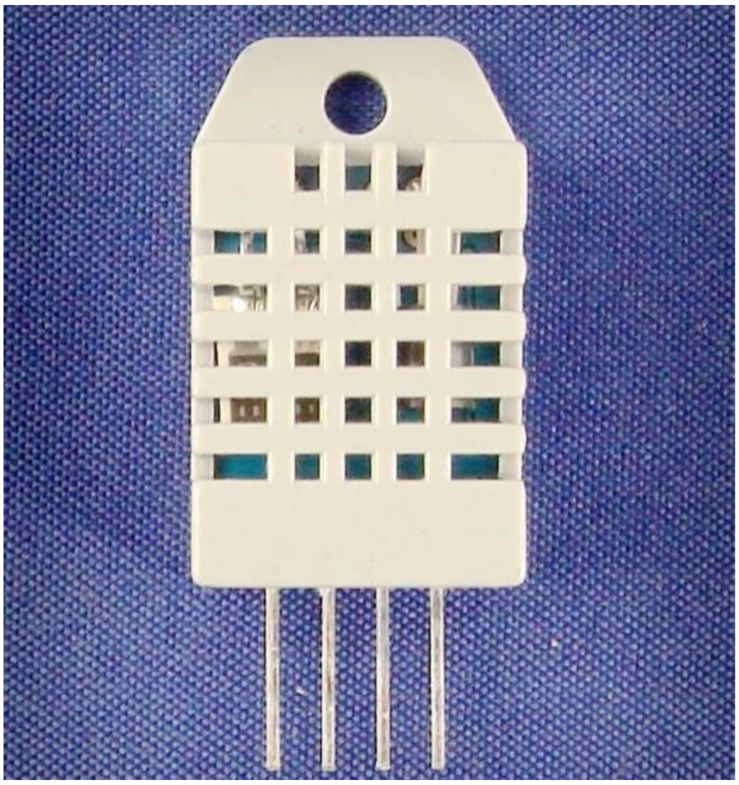
5. Supported devices and software usage

In this chapter are listed all the supported sensor, protocols and the correct way for writing the data that will be red using the language used for expressing IoT information, explained in the section 4.1. The language structure it's composed by a list of couple, keywords-value separated by the equals symbol ("=") and moreover the main keywords type that are composed by multiple values has recursively inside them a couple list with the form just explained eclosed into curly brackets. Those data can be concerned to a particular task, or a condition. At least, at the end of the chapter there will be also a part regarding the software usage.

5.1 Sensors, protocols and language

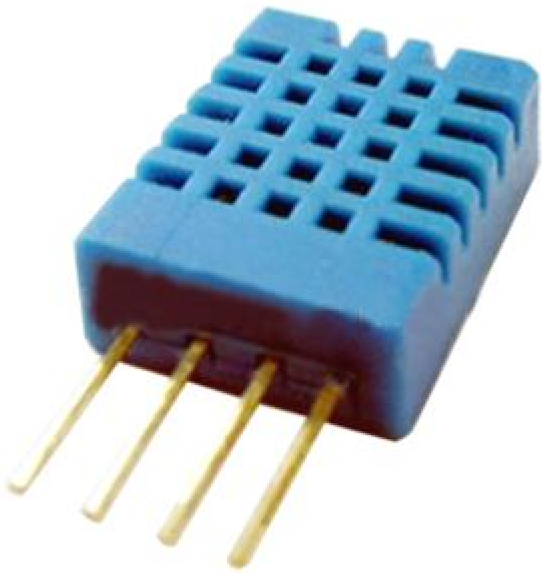
Shown below, in order, the list of the supported sensor and protocols with the related language.

- DHT22 temperature sensor: The DHT22 is a basic, low-cost digital temperature and humidity sensor. It uses a capacitive humidity sensor and a thermistor to measure the surrounding air, and spits out a digital signal on the data pin (no analog input pins needed). It's fairly simple to use, but requires careful timing to grab data. For using it, simply connect the first pin on the left to 3-5V power, the second pin to the data input pin and the rightmost pin to ground.



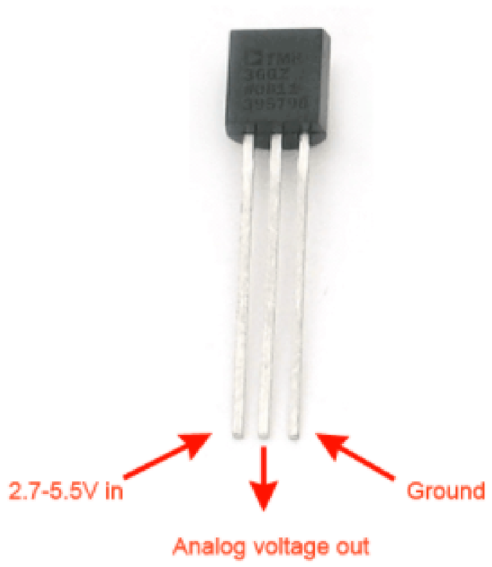
Picture 5.1: Figure of a dht22 sensor [45]

- DHT11 Temperature sensor: The DHT11 sensor, it's part of the same family of the previous sensor, but it's ultra low-cost; the language configuration in this case, it's in fact equal, the only difference it's in name field in which will be specified that the needed sensor it's the DHT11.

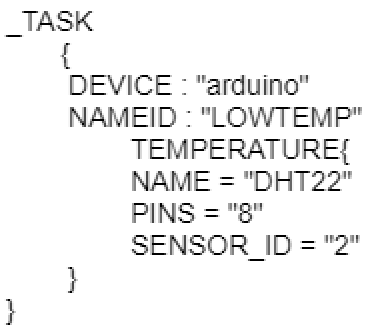


Picture 5.2: Figure of a dht11 sensor [46]

- TMP36 Temperature sensor: this sensor, its a chip that tells what the ambient temperature is. These sensors use a solid-state technique to determine the temperature based on the voltage, in fact, as temperature increases, the voltage across a diode increases at a known rate.
  Whether there's the necessity of using this particular sensors, the configuration should have the following syntax:
  For those sensors, the name field corresponds to the name of sensor, the field sensor id correspond to the id of this particular sensor (every sensor on the same device needs to have a different id), at least there's just one pin in this case that it's used for getting the red environment information.



Picture 5.3: Figure of a TMP36 sensor [47]



```
_TASK
    {
    DEVICE : "arduino"
    NAMEID : "LOWTEMP"
         TEMPERATURE{
         NAME = "DHT22"
         PINS = "8"
         SENSOR_ID = "2"
    }
}
```

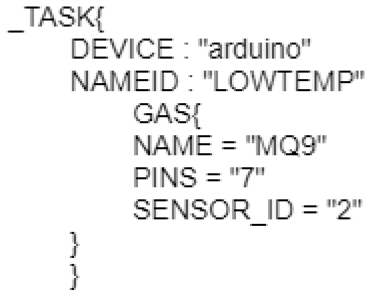
Picture 5.4: Example language used for generating code of a DHT22 sensor

- MQ9 Gas Sensor: A gas sensor is a device that detects the presence of one or more types of gas in the environment. These sensors have wide applications such as security systems of refineries, industrial centers, and even homes. These sensors can detect combustible gas, toxic gas, pollutant gas, and so on. There are several methods for gas detection, the most commonly used is electrochemical sensors. These sensors measure the concentration of a specific gas by performing a chemical reaction on their heated electrodes and measuring the resulting electric current. The MQ series presents different sensor model, that can red different gas type; the MQ9 it's able to determine the presence of carbon monoxide and flammable gasses.

  In this case the configuration it's similar to the one given for the temperature sensors, but in this case, the "keyword" that indicates that the selected

Picture 5.5: Figure of an MQ9 sensor [48]

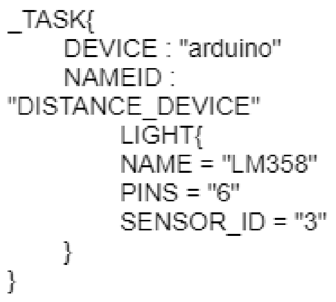
sensor it's part of the "gas" family it's "GAS":


```
_TASK{
     DEVICE : "arduino"
     NAMEID : "LOWTEMP"
          GAS{
          NAME = "MQ9"
          PINS = "7"
          SENSOR_ID = "2"
     }
     }
```

Picture 5.6: Example language used for generating code of an MQ9 sensor

- LM358 Light Sensor: The Grove Light sensor integrates a photoresistor (light dependent resistor) to detect the intensity of light. The resistance of photoresistor decreases when the intensity of light increases. A dual OpAmp chip LM358 on board produces a voltage corresponding to the intensity of light (based on resistance value). The output signal is an analog value. When the light is brighter, the value becomes larger. This sensor it's an 8 pin double operational amplifier IC, integrated with two Op-Amps in a solitary package. It has a wide power supply Range. Single supply ranges from 3V to 32V while dual supply ranges from ±1.5V to ±16V. It has a low supply current of 700μA. Also in this case the configuration language will be the same of the previous, but the keyword will be "LIGHT". The sensor language follows the same reasoning of the previous. As follows, the configuration language of the LM358

Picture 5.7: Figure of an LM358 sensor [54]

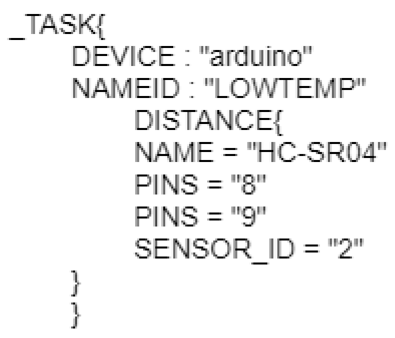


```
_TASK{
      DEVICE : "arduino"
      NAMEID :
"DISTANCE_DEVICE"
            LIGHT{
            NAME = "LM358"
            PINS = "6"
            SENSOR_ID = "3"
      }
}
```

Picture 5.8: Example language used for generating code of an LM358 sensor

- HC-SR04 ultrasonic sensor: The HC-SR04 uses non-contact ultrasound sonar to measure the distance to an object, and consists of two ultrasonic transmitters (basically speakers), a receiver, and a control circuit. The transmitters emit a high frequency ultrasonic sound, which bounce off any nearby solid objects, and the receiver listens for any return echo. That echo is then processed by the control circuit to calculate the time difference between the signal being transmitted and received. This time can subsequently be used, along with some clever math, to calculate the distance between the sensor and the reflecting object.
  In this case, the keyword will be “DISTANCE”, and since this sensor use multiple pins for reading data from the environment, the keywords “PINS” are repeated multiple times, one for every pin.

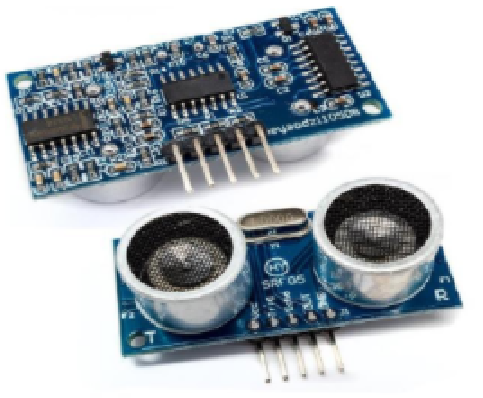
Picture 5.9: Figure of an HC-SR04 sensor [48]



```
_TASK{
      DEVICE : "arduino"
      NAMEID : "LOWTEMP"
            DISTANCE{
            NAME = "HC-SR04"
            PINS = "8"
            PINS = "9"
            SENSOR_ID = "2"
      }
      }
```

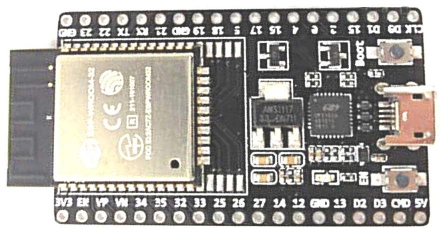
Picture 5.10: Example language used for generating code of an HC-SR04 sensor

- HY-SRF05 ultrasonic sensor: This module it's a longer range, precise ultrasonic sensor. These sensors use a pulse of ultrasonic sound and listen for a response. Since the speed of sound is relatively stable, we can measure the time between the pulse and the echo to determine distance. The HY-SRF05 it's a bit better at detecting objects at longer distances than the HC-SR04 in fact, this module it's the evolution of the HC-SR04. The SRF05 will detect a person sized object out to approximately 4 meters; walls and other large flat surfaces can be measured over 5 meters away. These sensor it's code compatible with any of the HC-SR04. Since that, the language to use in this case will be the same of the previous one used (of course the name of the sensor it's different).



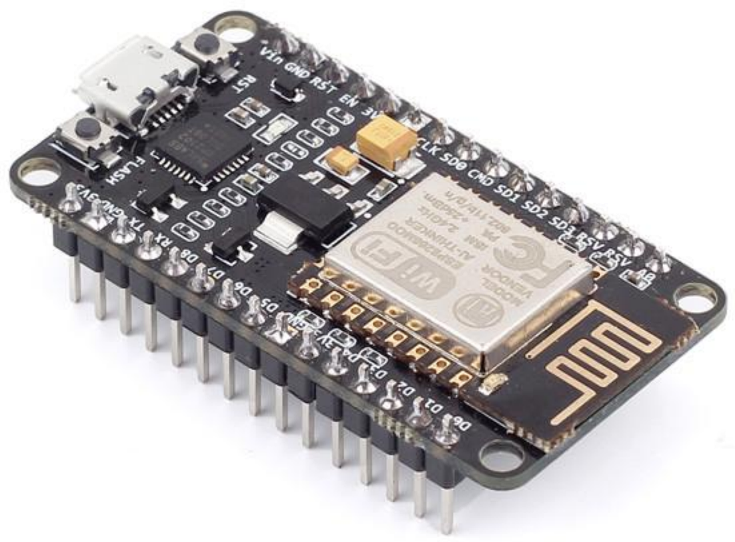
Picture 5.11: Figure of an HC-SRF05 sensor [49]

- ESP32: ESP32 is a series of low-cost, low-power system on a chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth. The ESP32 series employs a Tensilica Xtensa LX6 microprocessor in both dual-core and single-core variations and includes built-in antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power-management modules. ESP32 is created and developed by Espressif Systems, a Shanghai-based Chinese company, and is manufactured by TSMC using their 40 nm process. This module it's the successor of the ESP8266 microcontroller.



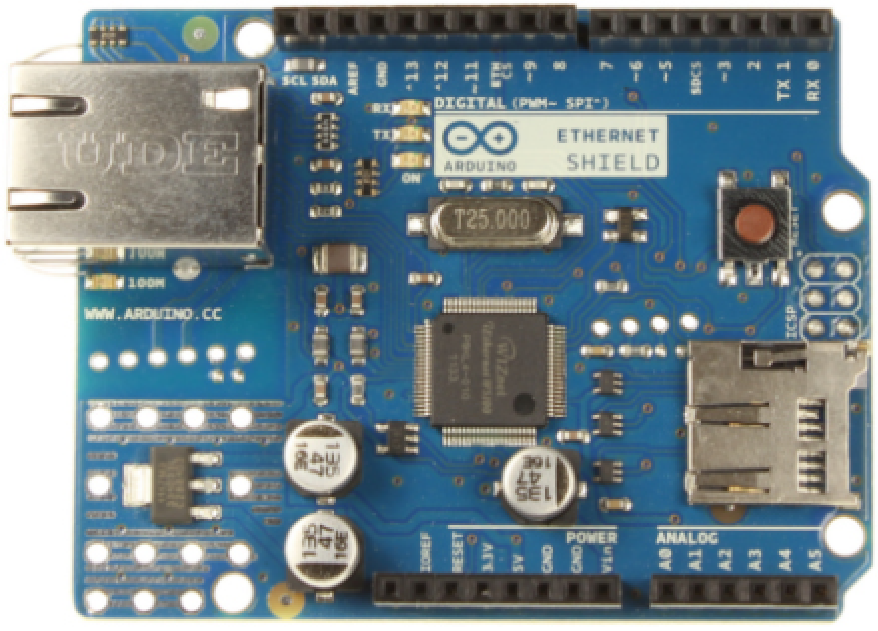
Picture 5.12: Figure of an ESP32 [50]

- ESP8266: ESP8266 is a low-cost WiFi module that belongs to ESP's family which can be uses to control electronics projects anywhere in the world. This sensor is a System on a Chip (SoC), manufactured by the Chinese company Espressif. It consists of a Tensilica L106 32-bit micro controller unit (MCU) and a Wi-Fi transceiver. It has 11 GPIO pins* (General Purpose Input/Output pins), and an analog input as well. This means that it's possible to program it like any normal Arduino or other micro-controller. This module has an in-built micro-controller and a 1MB flash allowing it to connect to a WiFi. The TCP/IP protocol stack allows the module to communicate with WiFi signals. The maximum working voltage of the module is 3.3v so it's not possible to supply 5v as it will fry the module.



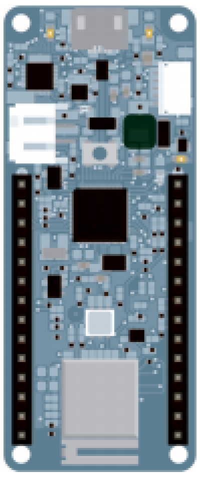
Picture 5.13: Figure of an ESP8266 [51]

- The W5100 is a versatile single-chip network interface chip with a 10/100Mbs Ethernet controller integrated internally. It is mainly used in a high integrated, high stable, high performance and low cost embedded system. The W5100 enables internet connection without operation system. It is also compatible with IEEE802.3 10BASE-T and 802.3u 100BASE-TX. W5100 integrates a market-proven full-hardware TCP/IP protocol stack inside, Ethernet media access control (MAC) layer and physical layer (PHY). The hardware TCP/IP protocol stack supports the following protocols: TCP, UDP, IPV4, ICMP, ARP, IGMP and PPoE, which have been tested by the market for years in many fields. In addition, W5100 also integrates 16KB memory for data transmission. For W5100, it's not necessary to consider the control of the Ethernet; what it's needed to do is socket programming. The W5100 has three interfaces: direct parallel bus, indirect parallel bus and SPI bus. It is easy to connect it with an MCU, just like accessing an external memory.



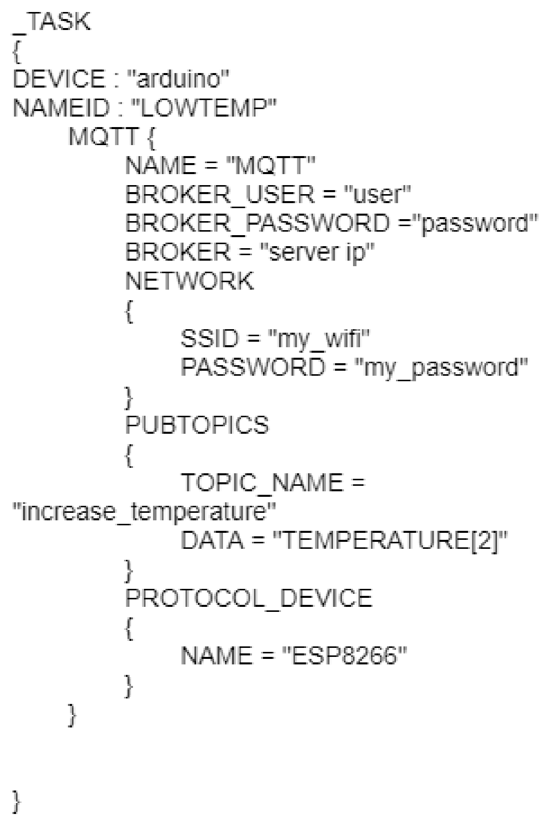
Picture 5.14: Figure of a W5100 [52]

- MKR 1010: The Arduino MKR WiFi 1010 is the easiest point of entry to basic IoT and pico-network application design. Whether it's intended to build a sensor network connected to an office or home router, or if there's the necessity to create a BLE device sending data to a cellphone, the MKR WiFi 1010 is a one-stop-solution for many of those basic IoT application scenarios. The board's main processor is a low power Arm Cortex-M0 32-bit SAMD21, like in the other boards within the Arduino MKR family. The WiFi and Bluetooth® connectivity is performed with a module from u-blox, the NINA-W10, a low power chipset operating in the 2.4GHz range. On top of those, secure communication is ensured through the Microchip ECC508 crypto chip.



Picture 5.15: Figure of an MKR1010 [53]

For all the shield above, the syntax language it's similar since those are all shields. The differences between the shields in relation of the same protocol, it's just the protocol device. In the next figures are shown a case in which there's the necessity of using MQTT protocol and HTTP protocol.



```
_TASK
{
DEVICE : "arduino"
NAMEID : "LOWTEMP"
      MQTT {
            NAME = "MQTT"
            BROKER_USER = "user"
            BROKER_PASSWORD ="password"
            BROKER = "server ip"
            NETWORK
            {
                  SSID = "my_wifi"
                  PASSWORD = "my_password"
            }
            PUBTOPICS
            {
                  TOPIC_NAME =
"increase_temperature"
                  DATA = "TEMPERATURE[2]"
            }
            PROTOCOL_DEVICE
            {
                  NAME = "ESP8266"
            }
      }


}
```

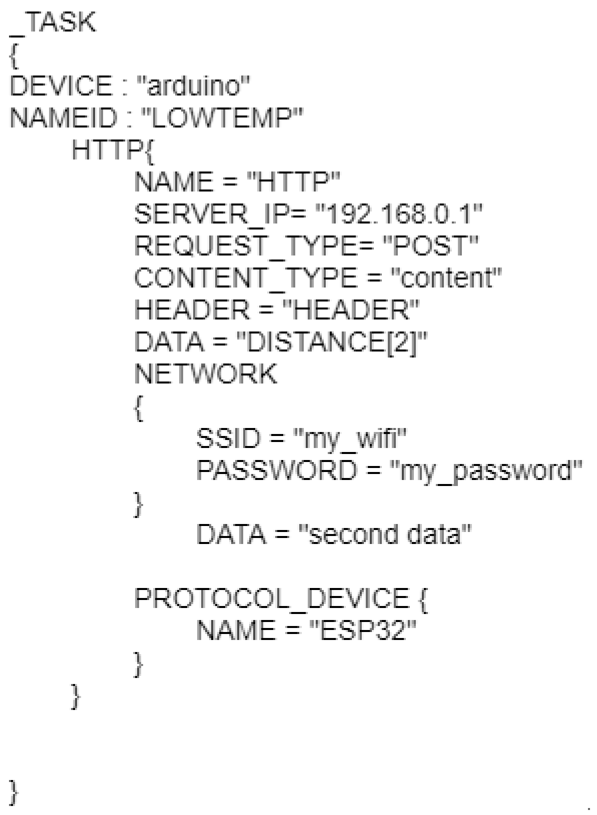
Picture 5.16: Language used for MQTT protocol on ESP8266

For this protocol, the required data have the syntax specified above, where:

- NAME : indicates the protocol name
- BROKER_USER : indicates the broker user
- BROKER_PASSWORD: indicates the broker password
- BROKER: indicates the ip address/link/name of the broker
- The subdata NETWORK indicates the information required for connecting to the access point
- the subdata PUBTOPICS it's used for specify the information regarding a topic where it's intended to publish something. In particular TOPIC_NAME indicates the topic name and DATA indicates the data that it's intended to be sent.

– the subdata SUBTOPICS it's used for specify the information regarding a topic where it's intended to subscribe. In particular TOPIC_NAME indicates the topic name.

– the subdata PROTOCOL_DEVICE it's used for specify which device it's used for enstablish the connection.



```
_TASK
{
DEVICE : "arduino"
NAMEID : "LOWTEMP"
     HTTP{
          NAME = "HTTP"
          SERVER_IP= "192.168.0.1"
          REQUEST_TYPE= "POST"
          CONTENT_TYPE = "content"
          HEADER = "HEADER"
          DATA = "DISTANCE[2]"
          NETWORK
          {
               SSID = "my_wifi"
               PASSWORD = "my_password"
          }
               DATA = "second data"

          PROTOCOL_DEVICE {
               NAME = "ESP32"
          }
     }


}
```

Picture 5.17: Language used for HTTP protocol (post request) on ESP8266

For this protocol, the required information have the syntax specified above, where:

– NAME : indicates the protocol name

– SERVER_IP: indicates the ip address/link/name of the server

– REQUEST_TYPE: indicates the request type, that can be GET or POST

– CONTENT_TYPE: indicates the content type of the request in case it's POST

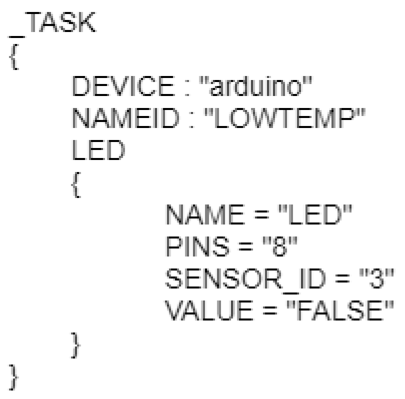
– HEADER: indicates the header of the request in case it's POST

  - The subdata NETWORK indicates the information required for connecting to the access point
  - DATA: indicates the data that it's intended to be sent in case the request type it's POST.
  - the subdata PROTOCOL_DEVICE it's used for specify which device it's used for establish the connection.

- Led: A light-emitting diode (LED) is a semiconductor light source that emits light when current flows through it. Electrons in the semiconductor recombine with electron holes, releasing energy in the form of photons. The color of the light (corresponding to the energy of the photons) is determined by the energy required for electrons to cross the band gap of the semiconductor. LEDs have many advantages over incandescent light sources, including lower energy consumption, longer lifetime, improved physical robustness, smaller size, and faster switching.

Picture 5.18: Figure of some leds turned on

The language that allow the usage of this semiconductor will be the following: where, in particular, the field value, can be true or false, in relation of the action performed by led (true indicates on, false indicates off).



```
_TASK
{
	DEVICE : "arduino"
	NAMEID : "LOWTEMP"
	LED
	{
		NAME = "LED"
		PINS = "8"
		SENSOR_ID = "3"
		VALUE = "FALSE"
	}
}
```

Picture 5.19: Language used for Led Managing

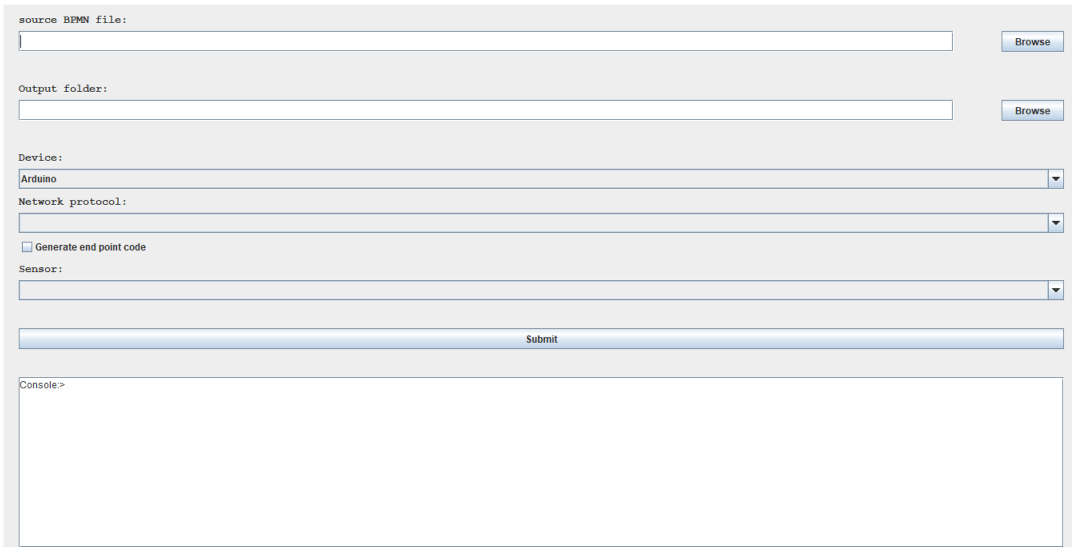
Following the flow of code there's of course the possibility of finding a condition, in this case the user that is writing the language, can specify it using the c programming language syntax. Another important feature it's given by the fact that the user can also interact with the expected data of the code. For doing it, it's necessary to use one of the following keyword, which correspond to the respective data;

- TEMPERATURE → red temperature of a temperature sensor
- GAS → red value of a gas sensor
- DISTANCE → red distance of a distance sensor
- LIGHT → red value of light sensor

those information in some cases (for example if it's intended to use a data red from a particular sensor connected to a particular device) needs to be enriched with one or two additional information, which are, the device id and the sensor id. The information will have then a syntax like: TEMPERATURE[deviceID , sensorID]. So, a condition that use to adopt it, will be like: TEMPERATURE [LOWTEMP , 2] <= 15, where the first operand it's the temperature value red from the sensor with id '2' connected to the arduino with id "LOWTEMP".

5.2 How to use the software

The software will also allow the programmer/user to work without creating a BPMN, but just specifying into the GUI the selected protocol or the selected sensor that it's intended to be used; in this particular case the result will be just the libraries, since there's not a sequence flow to follow. Otherwise if there's a BPMN as input, there will be just the necessity to select the XML file reported into the translation file as input, select an output folder and click the submit button; this will makes the software creates all the code for a specified diagram. In this case, the generated code will generate from three to N files, where two files are the library and the remaining are the main files, one for every device; so, increasing then the number of IoT systems in the diagram, will produce an increment of the output files. finally, all the information that in a normal situation would have been reported into the eclipse console has been intercepted and written inside a "MessageConsole", in this way, there's the possibility of seeing the various error or successfully messages directly from it without opening eclipse.



Device: Arduino

Console:>

Picture 5.20: software tool's GUI

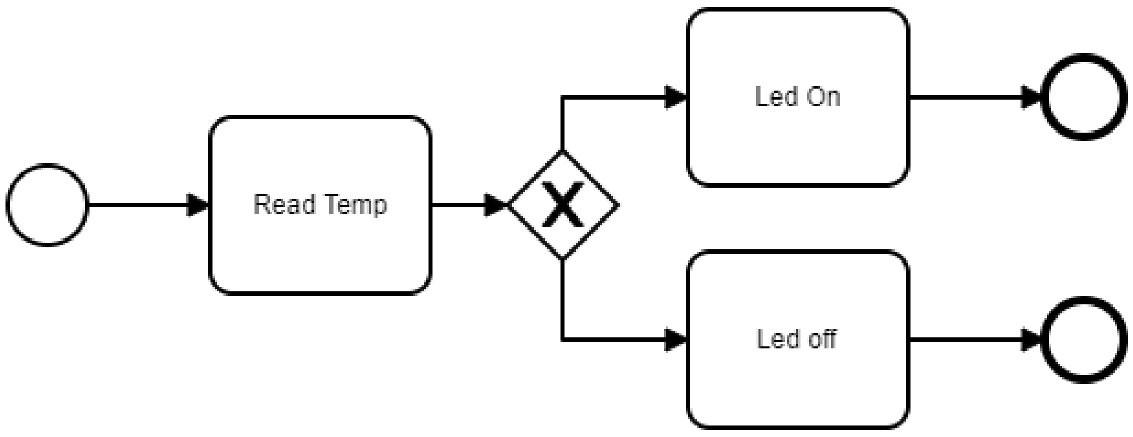
In case compiler gives errors during the compilation, it's possible to perform a syntax analysis of the XML content, for doing that, it's needed to open the eclipse IDE, right click the project and selected "run as eclipse instantiation"; after that a new instance of eclipse will be opened. After that, creating a new generic project, and converting it following the on screen instructions, will be possible to perform the analysis and check if there are errors in the file, where and which.

6. Framework Testing

In this chapter are reported the different tests that has been made. In particular, the explanation will start from a base case that will be expanded for every test in term of complexity.

6.1 Test 1

The first test consist of a simple example, in which it's represented a BPMN concerning a temperature monitoring. In particular in this case has been considered the usage of an arduino as device, which mount a DHT22 and a led. The logic in this case it's that, if the temperature red from the sensor it's higher then a specified value, the device will turn on a led meanwhile in case the value it's less than a specific value the led will turn off. In order, are shown, the BPMN Figure, the configuration language and the result code generated by the tool



Picture 6.1: Temperature test BPMN

In the Task “Read Temperature” it’s specified that the used sensor it’s a TEMP36, which it’s connected to the pin 7, having ID 2.

Code 6.1: Read Temp task specification language

```
_TASK
{
    DEVICE : "arduino"
    NAMEID : "LOWTEMP"
    TEMPERATURE
    {
        NAME = "TMP36"
        PINS = "7"
        SENSOR_ID = "2"
    }
}
```

The conditions written into the gateway are the following:

Code 6.2: Condition 1

```
TEMPERATURE [LOWTEMP , 2] > 10
```

Code 6.3: Condition 2

```
TEMPERATURE [LOWTEMP , 2] <= 10
```

In case the first condition it’s true, the task Led On” will be triggered, and the led will turn on. The led it’s connected to the pin 8 and the sensor id it’s 3, the field ”value” indicates the led state, where, true means that the led will turn on and false that the led will turn off. The configuration language in this case it’s:

Code 6.4: Led on

```
_TASK
{
    DEVICE : "arduino"
    NAMEID : "LOWTEMP"
    LED
    {
        NAME = "LED"
        PINS = "8"
        SENSOR_ID = "3"
        VALUE = "TRUE"
    }
}
```

Whether the second condition it's true, the led will turn off and the configuration it's:

Code 6.5: Led off

```
_TASK
{
    DEVICE : "arduino"
    NAMEID : "LOWTEMP"
    LED
    {
        NAME = "LED"
        PINS = "8"
        SENSOR_ID = "3"
        VALUE = "FALSE"
    }
}
```

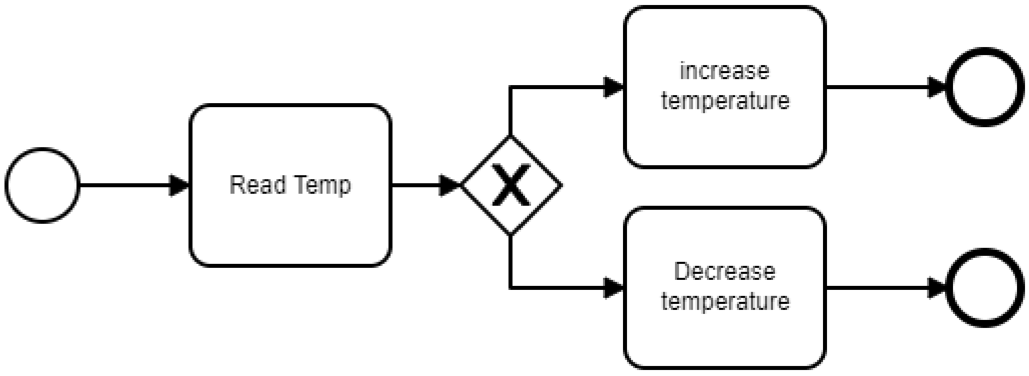
Running this configuration using the software tool, no library will be produced, since, the leds and that particular sensor has not particular methods that should be used; in this case then, the ".ino" file produced will be:

Code 6.6: .ino file

```
#include<GeneratedLib.h>

GeneratedLib my_lib;
int val_adc2; //adc value readed
float voltage2; //Voltage value
float temp2; //Stores temperature value
int pin2= 7;
int led_pin3= 8;
void setup()
{
    pinMode(led_pin3, OUTPUT);
}
void loop()
{
    //Read data and store it to variables
    val_adc2= analogRead(pin2);
    voltage2= ( val_adc2/ 1024.0) * 5.0;
    temp2= (voltage2- .5) * 1000;
    if(temp2 <=10)
    {
        //Turn off the led
        digitalWrite(led_pin3, LOW);
    }
    else if(temp2 >10)
    {
        //Turn on the led
        digitalWrite(led_pin3, HIGH);
    }
}
```

6.2 Test 2

In the second test, the device will act in a way similar to the first one, but in this case has been introduced the network protocol. In practice, this time, instead of turning on a led, the device will send the information red using the MQTT and an ESP8266 as shield to a specific broker's topic in relation of the information gets. If the temperature it's less or equal fifteen, the information will be send into the topic "Increase Temperature", meanwhile if the temperature it's higher or equal twenty it will be send in a topic called "Decrease Temperature".



In the following, it's written in order, the configuration language for "increase temperature" task and "decrease temperature" task.

Code 6.7: increase temperature

```
_TASK{
DEVICE : "arduino"
NAMEID : "LOWTEMP"
    MQTT {
        NAME = "MQTT"
        BROKER_USER = "user"
        BROKER_PASSWORD = "password"
        BROKER = "server ip"
        NETWORK{
            SSID = "my_wifi"
            PASSWORD = "my_password"
        }
        PUBTOPICS{
            TOPIC_NAME = "increase_temperature"
            DATA = "TEMPERATURE[2]"
        }
        PROTOCOL_DEVICE {
            NAME = "ESP8266"
        }
    }
}
```

Code 6.8: decrease temperature

```
_TASK{
DEVICE : "arduino"
NAMEID : "LOWTEMP"
    MQTT {
        NAME = "MQTT"
        BROKER_USER = "user"
        BROKER_PASSWORD = "password"
        BROKER = "server ip"
        NETWORK{
            SSID = "my_wifi"
            PASSWORD = "my_password"
        }
        PUBTOPICS{
            TOPIC_NAME = "decrease_temperature"
            DATA = "TEMPERATURE[2]"
        }
        PROTOCOL_DEVICE {
            NAME = "ESP8266"
        }
    }
}
```

In this case, the software tool will produce three files, that in order are: header library, source library and main file:

Code 6.9: Header file

```
#ifndef GeneratedLib_H //tests if RACom_H has not been defined
#define GeneratedLib_H //define GeneratedLib_H
#include "Arduino.h"         //includes the library Arduino.h
#include "SoftwareSerial.h" //Includes the library SoftwareSerial.h
#include <ESP8266WiFiMulti.h>
#include <ESP8266WiFi.h>
#include <PubSubClient.h>
#include <WiFiClient.h>

class GeneratedLib
{
private:
void InitDHT22(int pin);
void setupWiFi(char* ssid, char* password);
void Subscribe(char* topic);
void reconnect(int id, char* brokerUser, char* brokerPass,char*
    broker);
void callback(char* topic, byte* payload, unsigned int len);
void InitNetwork(char* broker, char* ssid, char* password);
void sendInPubTopic(char* pubTopic, char* datas);
};
#endif
```

Code 6.10: Source library

```
#include<GeneratedLib.h>
ESP8266WiFiMulti wifiMulti;
WiFiClient espClient;
PubSubClient client(espClient);
long currentTime, lastTime;
void InitDHT22(int pin){
    DHT dht(pin, DHTTYPE); // Initialize DHT sensor for normal 16mhz
        Arduino
    dht.begin();
}
void setupWiFi(char* ssid, char* password){
    delay(100);
    wifiMulti.addAP(ssid, password);
    Serial.println("Connecting ...");
    while (wifiMulti.run() != WL_CONNECTED){
        delay(250);
        Serial.print('.');
    }
    Serial.println('\n');
    Serial.print("Connected to:\t");
    Serial.println(WiFi.SSID());
    Serial.print("IP address:\t");
    Serial.println(WiFi.localIP());
}
void Subscribe(Char* topic){
    if(!client.connected()){
        reconnect(id, brokerUser, brokerPass, broker);
    }
    client.subscribe(topic);
}
void reconnect(int id, char* brokerUser, char* brokerPass,char*
    broker){
    while(!client.connected()){
        Serial.print("\nConnecting to ");
        Serial.println(broker);
        if(client.connect(id, brokerUser, brokerPass)){
            Serial.print("\nConnected to ");
            Serial.println(broker);
        }
        else{
            Serial.println("Connecting");
            delay(2500);
        }
    }
}
void callback(char* topic, byte* payload, unsigned int len){
    Serial.print("Received messages: ");
    Serial.println(topic);
    for(unsigned int i=0; i<len; i++){
        Serial.print((char) payload[i]);
    }
    Serial.println();
}
void InitNetwork(char* broker, char* ssid, char* password){
    Serial.begin(115200);
    setupWiFi(ssid, password);
    client.setServer(broker, 1883);
    client.setCallback(callback);
}
```

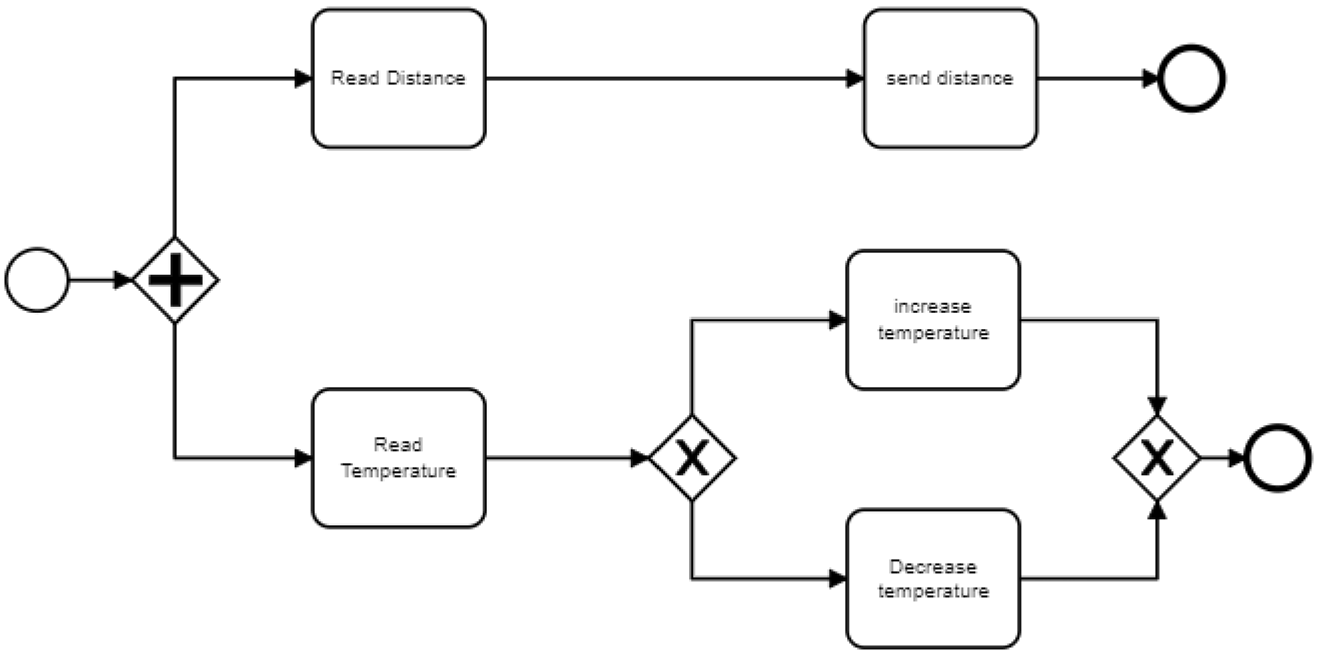
```
void sendInPubTopic(char* pubTopic, char* datas){
    if(!client.connected()){
        reconnect();
    }
    currentTime = millis();
    client.publish(pubTopic, datas);
    lastTime = millis();
}
```

Code 6.11: .ino file

```
#include<GeneratedLib.h>

GeneratedLib my_lib;

float hum2; //Stores humidity value
float temp2; //Stores temperature value value
int pin2= 7;
char* ssid2= "my_wifi";
char* wifi_password2= "my_password";
char* broker2= "server ip";
char* broker_user2= "user";
char* broker_password2= "password";
char* pubtopic20= "increase_temperature";
char* pubtopic50= "decrease_temperature";
char* pubtopicdata20= dtostrf(temp2, 6, 2, pubtopicdata20);

void setup()
{
    Serial.begin(9600);
    InitDHT22(pin2);
}

void loop()
{
    delay(2000);
    //Read data and store it to variables
    hum2= dht.readHumidity();
    temp2= dht.readTemperature();
    if(temp2 <=15)
    {
        my_lib.InitNetwork(broker2,ssid2,wifi_password2);
        my_lib.reconnect("device id", broker_user2, broker_password2,
            broker2);
        dtostrf(temp2, 6, 2, pubtopicdata20);
        my_lib.sendInPubTopic(pubtopic20, pubtopicdata20);
    }
    else if(temp2 >=20)
    {
        my_lib.InitNetwork(broker2,ssid2,wifi_password2);
        my_lib.reconnect("device id", broker_user2, broker_password2,
            broker2);
        dtostrf(temp2, 6, 2, pubtopicdata20);
        my_lib.sendInPubTopic(pubtopic50, pubtopicdata20);
    }
}
```

6.3 Test 3

In the third test it's still present a single device, but this time, it will run multiple thread. The first thread will read the distance using the HC-SR04 distance sensor and after that using the HTTP protocol, will send to a server the data red. The second thread meanwhile will act as the previous example.



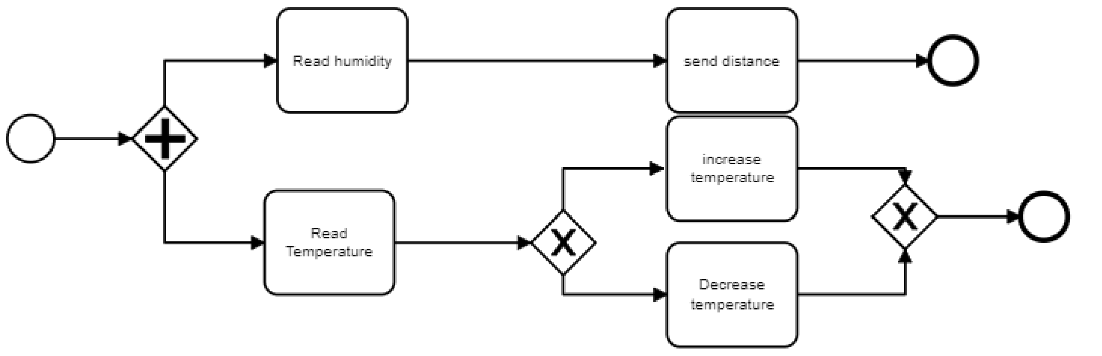
For space reason, just the main file code it's reported:

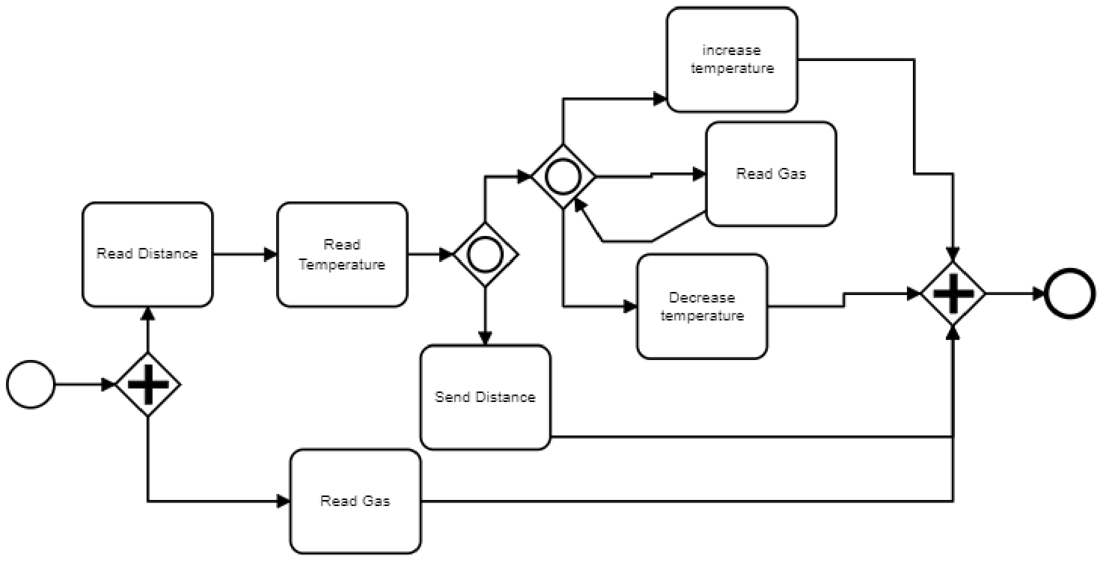
Code 6.12: .ino file

```
#include<GeneratedLib.h>
GeneratedLib my_lib;
int distance2; //Stores distance value
int pin28= 8;
int pin29= 9;
float hum2; //Stores humidity value
float temp2; //Stores temperature value value
int pin2= 8;
char* ssid2= "my_wifi";
char* wifi_password2= "my_password";
char* broker7= "server ip";
char* broker_user7= "user";
char* broker_password7= "password";
char* pubtopic70= "decrease_temperature";
char* pubtopic100= "increase_temperature";
char* pubtopicdata70= dtostrf(temp2, 6, 2, pubtopicdata70);
char* server_ip2= "192.168.0.1";
char* header2= "HEADER";
char* content_type2= "content";
char* send_data20= dtostrf(distance2, 6, 2, send_data20);
void task4(void *pvParameters){
    //Read data and store it to variables
    my_lib.InitHCSR04(pin28, pin29);
    distance2 = my_lib.ReadDistanceHCSR04(pin28, pin29);
    my_lib.setupWiFiHTTP(ssid2, wifi_password2);
    dtostrf(distance2, 6, 2, send_data20);
    my_lib.sendPost(server_ip2, content_type2, header2, send_data20);
    my_lib.sendPost(server_ip2, content_type2, header2, send_data21);
}
void task5(void *pvParameters){
    delay(2000);
    hum2= dht.readHumidity();
    temp2= dht.readTemperature();
    if(20<=temp2 ){
        my_lib.InitNetwork(broker7, ssid2, wifi_password2);
        my_lib.reconnect("device id", broker_user7, broker_password7,
            broker7);
        dtostrf(temp2, 6, 2, pubtopicdata70);
        my_lib.sendInPubTopic(pubtopic70, pubtopicdata70);
    }
    else if(15>=temp2 ){
        my_lib.InitNetwork(broker7, ssid2, wifi_password2);
        my_lib.reconnect("device id", broker_user7, broker_password7,
            broker7);
        dtostrf(temp2, 6, 2, pubtopicdata70);
        my_lib.sendInPubTopic(pubtopic100, pubtopicdata70);
    }
}
void setup(){
    Serial.begin(9600);
    InitDHT22(pin28);
}
void loop(){
    xTaskCreate(task4, "task4", 128, NULL, 1, NULL);
    xTaskCreate(task5, "task5", 128, NULL, 1, NULL);
    vTaskStartScheduler();
}
```

6.4 Test 4

For the last test has been added to the previous example another device, that will act independently from the other (take in mind that every device has his own id, so adding a task somewhere in which are specified information for a device with a different id, means that another device will be considered by the software), the first device it's represented by the first business process, meanwhile the second it's represented by the second business process. The first device will act as shown in the third example, meanwhile, the second device will have a different behaviour. The first thread will read the distance and then the temperature, after that some conditions and a loop are present. Second task will just read data from the gas sensor.





Shown below, the produced code for the second device:

Code 6.13: .ino file

```
#include<GeneratedLib.h>

GeneratedLib my_lib;

int distance1; //Stores distance value
int pin17= 7;
int pin18= 8;
float hum2; //Stores humidity value
float temp2; //Stores temperature value value
int pin2= 9;
float light4; //Stores value
int pin4= 5;
float gas3; //Stores value
int pin3= 6;
char* ssid18= "my_wifi";
char* wifi_password18= "my_password";
char* broker18= "server ip";
char* broker28= "server ip2";
char* broker_user18= "user";
char* broker_password18= "password";
char* pubtopic180= "increase_temperature";
char* pubtopic210= "decrease_temperature";
char* pubtopic280= "primo topic2";
char* pubtopicdata180= dtostrf(temp2, 6, 2, pubtopicdata180);
char* pubtopicdata280= "prova";
void task2(void *pvParameters)
{
    //Read data and store it to variables
    my_lib.InitHCSR04(pin17, pin18);
    distance1 = my_lib.ReadDistanceHCSR04(pin17, pin18);
    delay(2000);
    //Read data and store it to variables
    hum2= dht.readHumidity();
    temp2= dht.readTemperature();
    if(distance1 <=10)
    {
        if(temp2 !=10)
        {
            my_lib.InitNetwork(broker18,ssid18,wifi_password18);
            my_lib.reconnect("device id", broker_user18,
                broker_password18, broker18);
            dtostrf(temp2, 6, 2, pubtopicdata180);
            my_lib.sendInPubTopic(pubtopic180, pubtopicdata180);
        }
        if(temp2 ==10)
        {
            my_lib.InitNetwork(broker18,ssid18,wifi_password18);
            my_lib.reconnect("device id", broker_user18,
                broker_password18, broker18);
            dtostrf(temp2, 6, 2, pubtopicdata180);
            my_lib.sendInPubTopic(pubtopic210, pubtopicdata180);
        }
        while(temp2 >=10)
        {
            //Read data and store it to variables
            light4= my_lib.Readlm358(pin4);
        }
    }
    if(distance1 >=10)
    {
        my_lib.InitNetwork(broker28,ssid18,wifi_password18);
        my_lib.reconnect("device id", broker_user18, broker_password18,
            broker28);
        my_lib.sendInPubTopic(pubtopic280, pubtopicdata280);
    }
}
void task3(void *pvParameters)
{
    //Read data and store it to variables
    gas3= my_lib.ReadMQ9(pin3);
}
void setup()
{
    Serial.begin(9600);
    InitDHT22(pin2);
}

void loop()
{
    xTaskCreate(task2, "task2", 128, NULL, 1, NULL);
    xTaskCreate(task3, "task3", 128, NULL, 1, NULL);
    vTaskStartScheduler();
}
```

7. Conclusion

The proposed work has tried to solve two main issues regarding the developing of a model-driven based project. The problems taken in exam are: Firstly, the difficulty of representing the low-level information regarding the IoT systems in the diagram and secondly the fact that the implementation use to deviates from the intentions and expectations of the business analysts because cross-domain communication is prone to misunderstandings. Since the various misinterpretation can cause a substantial waste of time, it's also proposed a tool that help for the diagram interpretation. For solving the issues has been firstly made a long research using different academic articles investigations and surveys that are reported in the references. Those papers gave a general overview of the problem, making easier the reasoning for find a solution. The final result of the work, propose a framework based on model-driven approach that enriched with the techniques expressed in the section 2.2 allow the modeler to represent using a BPMN diagram the IoT environment. Since the Internet Of Things taken in exam needs additional information for being represented, the approach proposed, allow the modeler to enrich the diagram with a defined language expressed in section 5.1. At least the framework consists also in a tool java based, that taken as input a BPMN diagram in which it's represented one or more IoT scenarios , will produce code that can simplify substantially the work of the programmer and reduce essentially the possibility of a misunderstanding. It's possible to affirm then, that the framework solves the majour issues discussed above. It's important call to mind that the proposed software it's a first prototype and that it's not compatible with all the possible situations that can occurs. In this sense, a possible future development, can be the expansion of the framework domain in every direction, adding to the supported sensor and protocols, other possibilities or improving the code optimization and the language.

References

[1] Cognini, R., Corradini, F., Gnesi, S., Polini, A., & Re, B. (2018). *Business process flexibility-a systematic literature review with a software systems perspective. Information Systems Frontiers,* . 20(2), 343-371.

[2] Caracaş, A., & Bernauer, A. (2011, June). *Compiling business process models for sensor networks. In 2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)* (pp. 1-8). IEEE.

[3] Meyer, S., Sperner, K., Magerkurth, C., & Pasquier, J. (2011, June). *Towards modeling real-world aware business processes. In Proceedings of the Second International Workshop on Web of Things* (pp. 1-6).

[4] Friedow, C., Völker, M., & Hewelt, M. (2018, June). *Integrating IoT devices into business processes. In International Conference on Advanced Information Systems Engineering* (pp. 265-277). Springer, Cham.

[5] Ferreira, P., Martinho, R., & Domingos, D. (2014). *Process invariants: an approach to model expected exceptions. Procedia Technology,* 16, 824-833.

[6] Braun, R. (2015, February). *Behind the scenes of the bpmn extension mechanism principles, problems and options for improvement. In 2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)* (pp. 1-8). IEEE.

[7] Martins, F., & Domingos, D. (2017). *Modelling IoT behaviour within BPMN business processes. Procedia computer science,* 121, 1014-1022.

[8] Domingos, D., & Martins, F. (2017). *Using BPMN to model Internet of Things behavior within business process. International Journal of Information Systems and Project Management,* 5(4), 39-51.

[9] Thomas, I., Kikuchi, S., Baccelli, E., Schleiser, K., Doerr, J., & Morgenstern, A. (2018). *Design and implementation of a platform for hyperconnected cyber physical systems. Internet of Things,* 3, 69-81.

[10] DaSilva, C. M., & Trkman, P. (2014). *Business model: What it is and what it is not. Long range planning,* 47(6), 379-389.

[11] Casati, F., Daniel, F., Dantchev, G., Eriksson, J., Finne, N., Karnouskos, S., ... & Voigt, T. (2012, June). *Towards business processes orchestrating the physical enterprise with wireless sensor networks. In 2012 34th International Conference on Software Engineering (ICSE)* (pp. 1357-1360). IEEE.

[12] Caracaş, A., & Bernauer, A. (2011, June). Compiling business process models for sensor networks. *In 2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)* (pp. 1-8). IEEE.

[13] Kocbek, M., Jošt, G., Heričko, M., & Polančič, G. (2015). *Business process model and notation: The current state of affairs. Computer Science and Information Systems,* 12(2), 509-539.

[14] Recker, J. (2010). *Opportunities and constraints: the current struggle with BPMN. Business Process Management Journal.*

[15] Weske, M. (2019). *Business Process Management: Concepts, Languages, Architectures. Springer.*

[16] Curtis, B., Kellner, M. I., & Over, J. (1992). *Process modeling. Communications of the ACM,* 35(9), 75-90.

[17] Mass, J., Chang, C., & Srirama, S. N. (2016, December). *Wiseware: A device-to-device-based business process management system for industrial internet of things. In 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (pp. 269-275). IEEE.

[18] Yousfi, A., Hewelt, M., Bauer, C., & Weske, M. (2018). *Toward uBPMN-based patterns for modeling ubiquitous business processes. IEEE Transactions on Industrial Informatics,* 14(8), 3358-3367.

[19] Yousfi, A., Bauer, C., Saidi, R., & Dey, A. K. (2016). *uBPMN: A BPMN extension for modeling ubiquitous business processes. Information and Software Technology,* 74, 55-68.

[20] Yousfi, A., De Freitas, A., Dey, A. K., & Saidi, R. (2015). *The use of ubiquitous computing for business process improvement. IEEE Transactions on Services Computing,* 9(4), 621-632.

[21] Ranganathan, A., & Campbell, R. H. (2003). *An infrastructure for context-awareness based on first order logic. Personal and Ubiquitous Computing,* 7(6), 353-364.

[22] Azuma, R., Baillot, Y., Behringer, R., Feiner, S., Julier, S., & MacIntyre, B. (2001). *Recent advances in augmented reality. IEEE computer graphics and applications,* 21(6), 34-47.

[23] Cook, D. J., Augusto, J. C., & Jakkula, V. R. (2009). *Ambient intelligence: Technologies, applications, and opportunities. Pervasive and Mobile Computing,* 5(4), 277-298.

[24] Parr, T. J., & Quong, R. W. (1995). *ANTLR: A predicated-LL (k) parser generator. Software: Practice and Experience,* 25(7), 789-810.

[25] Meyer, S., Ruppen, A., & Magerkurth, C. (2013, June). *Internet of things-aware process modeling: integrating IoT devices as business process resources. In International conference on advanced information systems engineering* (pp. 84-98). Springer, Berlin, Heidelberg.

[26] da Cruz, M. A., Rodrigues, J. J., Sangaiah, A. K., Al-Muhtadi, J., & Korotaev, V. (2018). *Performance evaluation of IoT middleware. Journal of Network and Computer Applications,* 109, 53-65.

[27] Scott Robert, Östberg Daniel *A comparative study of open-source IoT middleware platforms.*

[28] JS Foundation. *"Node-RED Flow-based programming for the Internet of Things".* http://nodered.org/

[29] CyberVision, Inc. *"Key Capability of the Kaa IoT platform" https://www.kaaproject.org/platform/*

[30] CyberVision, Inc. *"Architecture Overview". https://kaaproject.github.io/kaa/docs/v0.10.0/Architecture-overview/*

[31] Hussein, M., Li, S., & Radermacher, A. (2017, September). *Model-driven Development of Adaptive IoT Systems. In MODELS (Satellite Events)* (pp. 17-23).

[32] Chang, C., Srirama, S. N., & Buyya, R. (2016). *Mobile cloud business process management system for the internet of things: a survey. ACM Computing Surveys (CSUR),* 49(4), 1-42.

[33] Meroni, G., Baresi, L., Montali, M., & Plebani, P. (2018). *Multi-party business process compliance monitoring through IoT-enabled artifacts. Information Systems,* 73, 61-78.

[34] Teniente, E., & Weidlich, M. (Eds.). (2018). *Business Process Management Workshops: BPM 2017 International Workshops, Barcelona, Spain, September 10-11, 2017, Revised Papers* (Vol. 308). Springer.

[35] White, S. A. (2004). *Introduction to BPMN. Ibm Cooperation,* 2(0), 0.

[36] Sosa-Reyna, C. M., Tello-Leal, E., Lara-Alabazares, D., Mata-Torres, J. A., & Lopez-Garza, E. (2018). *A Methodology Based on Model-Driven Engineering for IoT Application Development. ICDS 2018, 45.*

[37] Schönig, S., Ackermann, L., Jablonski, S., & Ermer, A. (2018). *An integrated architecture for iot-aware business process execution. In Enterprise, Business-Process and Information Systems Modeling* (pp. 19-34). Springer, Cham.

[38] Brambilla, M., Umuhoza, E., & Acerbis, R. (2017). *Model-driven development of user interfaces for IoT systems via domain-specific components and patterns. Journal of Internet Services and Applications,* 8(1), 14.

[39] Song, R., Vanthienen, J., Cui, W., Wang, Y., Huang, L. (2019, July). *Context-Aware BPM Using IoT-Integrated Context Ontologies and IoT-Enhanced Decision Models. In 2019 IEEE 21st Conference on Business Informatics (CBI)* (Vol. 1, pp. 541-550). IEEE.

[40] Neubauer, P., Bergmayr, A., Mayerhofer, T., Troya, J., & Wimmer, M. (2015, October). *XMLText: from XML schema to Xtext. In Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering* (pp. 71-76).

[41] Neubauer, P., Bill, R., & Wimmer, M. (2017, February). *Modernizing domain-specific languages with XMLText and IntellEdit. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 565-566). IEEE.

[42] *https://circle.visual-paradigm.com/docs/introduction-and-system-requirements/introduction-of-visual-paradigm/*

[43] *https://www.eclipse.org/Xtext/download.html*

[44] *https://en.wikipedia.org/wiki/Xtend*

[45] *https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf*

[46] *https://www.mouser.com/datasheet/2/758/DHT11-Technical-Data-Sheet-Translated-Version-1143054.pdf*

[47] *https://learn.adafruit.com/tmp36-temperature-sensor*

[48] *https://www.mouser.com/catalog/specsheets/Seeed_101020045.pdf*

[49] *https://datasheetspdf.com/datasheet/HY-SRF05.html*

[50] *https://components101.com/microcontrollers/esp32-devkitc*

[51] *https://indomus.it/guide/riprogrammare-firmware-su-dispositivi-esp8266-sonoff-shelly-ecc-masterguide/*

[52] *https://www.arduino.cc/en/Main/ArduinoEthernetShieldV1*

[53] *https://www.mouser.com/datasheet/2/34/Arduino_0822018_ABX00023-1391986.pdf*

[54] *https://electronics.stackexchange.com/questions/33983/lm358-op-amp-for-a-light-sensor*

8. Acknowledgements

I would like to express the deepest appreciation to my Realtor, Andrea Polini and my Co-relator, Fabrizio Fornari for the continuous support, patience, motivation and knowledge. Their guidance helped me in all the time of research and writing of this thesis. I'm thankful for what they offered me.

I am extremely grateful to my parents, my brother, and my girlfriend who supported me with love and understanding. Without you I could have never reached this goal.

Finally, my thanks goes to the University of Camerino, for affording me the opportunity to complete my study here, to all the professors that taught me all the knowledge needed with professionalism and passion for their work, and, to everyone who played a role in my academic accomplishments.