

Graphic Language for Programming Experiments in Virtual Laboratories

Máster Universitario en Investigación en Ingeniería
del Software y Sistemas Informáticos

Universidad Nacional de Educación a Distancia

Autor: Miguel Fadrique de Pablo

Directors: Rubén Heradio Gil and Daniel Galán Vicente

Academic Year 2016-2017

February 2017



Máster Universitario en Investigación en Ingeniería
del Software y Sistemas Informáticos

Universidad Nacional de Educación a Distancia

**Graphic Language for Programming
Experiments in Virtual Laboratories**

Autor: Miguel Fadrique de Pablo

Directors: Rubén Heradio Gil and Daniel Galán Vicente

Academic Year 2016-2017

February 2017

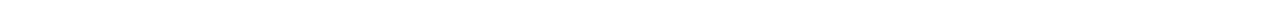
Grade Sheet:



Autorización

Autorizo a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del Autor

A handwritten signature in black ink, consisting of several stylized, overlapping loops and lines, positioned below the text 'Firma del Autor'.

Abstract

One of the main objectives of the software development is to get systems usable for everyone, no matter his/her previous knowledge. Getting more accesible systems allow to take advantage of the software products by everyone. The system described in this document follows this principle. In a simple way, it is a graphical implementation to convert blocks into code. This code will be used to execute experiments using the Experiment Editor tool created by the Department of Computer Science and Automatic Control, Universidad Nacional de Educación a Distancia and the Department of Software Engineering and Computer systems, Universidad Nacional de Educación a Distancia. The proposed solution is formed by a graphic interface to drag-and-drop the blocks and a generator to convert those blocks into code.

Resumen

Uno de los principales objetivos que se buscan a la hora de desarrollar código es conseguir que se pueda usar por cualquier persona sin importar los conocimientos que tenga. Obtener sistemas más accesibles permite el mejor aprovechamiento de los productos de software. El sistema descrito en este documento sigue este principio. De manera simple, es una interfaz gráfica que permite la conversión de bloques a código. Este código se puede emplear para desarrollar experimentos usando la herramienta "Experiment Editor" creada por el departamento de Informática y Automática de la Universidad Nacional de Educación a Distancia y el departamento de Ingeniería de Software y Sistemas Informáticos de la Universidad Nacional de Educación a Distancia. La solución propuesta se compone una interfaz gráfica para arrastrar los bloques y un generador que convierte estos bloques en código.

Keywords:

Block Programming, Visual Programming, Blockly, Java Virtual Machine, Easy javascript Simulations, Experiment Editor, Virtual Laboratories, Ruby.

Index:

1. Introduction	1
1.1.Scope.....	1
1.2.Our Approach in a Nutshell.....	2
1.3.Organization of this Work	3
2. State of the Art	5
2.1.Visual Block programming.....	5
2.1.1.Blockly	6
2.1.2.Scratch.....	8
2.1.3.Waterbear.....	9
2.2.Related Work	10
2.3.Limitations	12
2.4.Domain Specific Language.....	12
2.5.Easy Java(Script) Simulations	13
2.6.Experiment Editor	14
2.7.Blockly system.....	15
2.8.Block Factory.....	16
3. Graphic language	23
3.1.Development.....	23
3.2.Interface creation.....	28
3.3.Blocks Palettes Categories Tabs.....	30
3.4.Java Development	33
3.5.Code Button.....	35
3.6.Ruby interpreter	36
3.7.Export Code Function.....	38
3.8.Export Blocks	40
3.9.Import Blocks	41
3.10. Input Variables	43
4. Conclusions	49
4.1. Results	50

5. Future Directions	51
5.1.Import Code	51
5.2.Adapt the system	51
5.3.JVM Complete Implementation	51
5.4.DataTool implementation	52
6. References	53



Figures Index:

Figure 1. New execution flow	4
Figure 2. Block Factory Interface.....	16
Figure 3. Block to create blocks	16
Figure 4. Input value blocks	17
Figure 5. Types of blocks	18
Figure 6. Field Values.....	19
Figure 7. Automatic and External Input.....	19
Figure 8. Inline input	19
Figure 9. Connection Options.....	20
Figure 10. Color Grades.....	20
Figure 11. Preview Screen	21
Figure 12. Block Definition	21
Figure 13. Generator Stub.....	22
Figure 14. Experiments with Events Structure.....	27
Figure 15. Example of Experiment with Events	28
Figure 16. Complete Experiment with Events Function.....	28
Figure 17. Graphic Interface.....	29
Figure 18. Blocks Classification.....	30
Figure 19. Simple Experiment Block	31
Figure 20. Experiment with Events Block.....	31
Figure 21. Simple Event Block.....	32
Figure 22. Complex Event Block.....	32
Figure 23. Java Interface Buttons.....	34
Figure 24. Demonstration of code obtained in Java.....	36
Figure 25. Ruby Code Execution	37
Figure 26. Select Output File Directory Screen	39
Figure 27. Create variable block.....	45
Figure 28. Variable block	46

Code Index:

Code 1. Complex Event Block.....	24
Code 2. Block Ruby Code	26
Code 3. Experiment Text Examples	27
Code 4. Simple Experiment Code	31
Code 5. Load method call.....	34
Code 6. showCode function.....	35
Code 7. JRuby Code to Execute Ruby Code	37
Code 8. Result of an Execution	38
Code 9. Export Code Button.....	39
Code 10. exportBlocks Function.....	40
Code 11. XML File Creation	41
Code 12. importBlocks Function.....	41
Code 13. XML File Creation	42
Code 14. addElement method	43
Code 15. variables_set method	45
Code 16. variables_get block.....	45
Code 17. addElement() listener.....	47

1. INTRODUCTION

1.1.Scope

Laboratory experimentation plays an essential role in control education but the cost of getting and maintaining the instruments needed is very high. In the Universidad Nacional de Educación a Distancia EjsS virtual and remote laboratories are used to reduce the costs associated to traditional hands-on laboratories and to support online experimentation. These laboratories allow the students to create and use different experiments using only one tool, a computer.

In order to define experiments in virtual laboratories an experiment editor has been created by the Department of Computer Science and Automatic Control, Universidad Nacional de Educación a Distancia and the Department of Software Engineering and Computer systems, Universidad Nacional de Educación a Distancia. The Experiment Editor tool has been created to define experiments over laboratories developed using Easy JavaScript Simulations, EjsS. There is a huge number of repositories of projects developed using this tool so this experiment system is prepared to be applicable for all the laboratories created using EjsS.

The biggest problem when using this system is that it requires a minimum knowledge of programming in order to change the parameters or to create an experiment as it is a Ruby based system. To solve these difficulties, a graphic block programming system has been described in this document. This system allows the user to develop experiments that will be used in the Experiment Editor tool without knowledge of the application code and to do it avoiding syntax or format issues.

Many programming languages exist currently but they cannot be applicable in this specific case for several reasons:

- Common programming languages are difficult to learn without a minimum knowledge and without attending tutorials or courses.
- Programming has been always connected to several tasks as mathematical calculations which causes that some people do not learn it because its use is associated to some specific jobs.
- When programming, it is common to need support and it is not always present in these experiments as most of them are very specific.

Using the knowledge acquired in the Master studies, a new system has been created that will avoid these problems. Furthermore, it will provide an attractive and dynamic interface that will allow the users to use the system already developed taking advantage of all the functionalities easily without any previous knowledge necessary.

1.2. Our Approach in a Nutshell

To use the system currently presented it is necessary to follow some steps:

- Open the Experiment Editor.
- Open a specific experiment inside it.
- Write the code of the experiment.
- Run the experiment.

After the appliance of the changes specified in this document the functionality is going to be the same but the execution is going to be easier as the third step will have a substitute. From now there is no need to write the code because it will be done using the graphic language explained in this document.

An example is going to be explained, Figure 1, to understand the flow using the new functionality. It starts with the Experiment Editor. The following step is the opening of a new experiment. Once opened, the graphic language can be opened to drag-and-drop blocks to create the experiment code. Finally, the code generated using blocks is passed to the Experiment Editor and executed there.

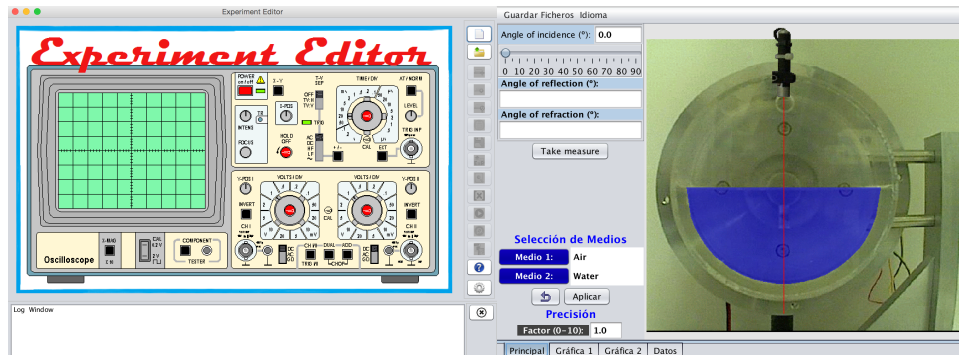
So at the end the same steps are done but the code has been generated using a system easier to use, the visual language.

1.3.Organization of this Work

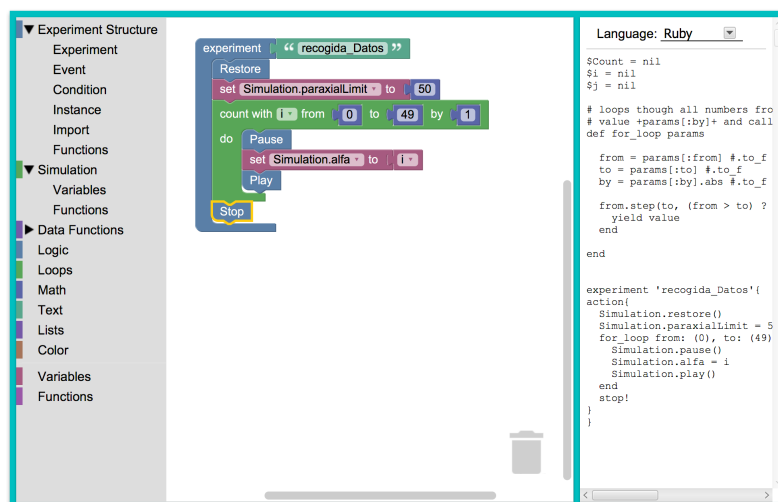
In this document the development of a new visual programming environment is explained. Chapter 2 describes the state of the art in this moment. It is explained with a brief introduction describing the visual blocks programming, the different block programming languages available and their advantages and drawbacks. At the end of chapter 2, an explanation of the restrictions that are currently present in the already developed systems is done as well as a summary of some systems already developed that are related to this one highlighting why they are not valid for this specific development. Chapter 3 focuses on the explanation of the system completely seeing all the functionalities implemented as well as some examples of how the system works and results obtained using it. Chapter 4 describes the conclusions obtained after seeing the document completely and chapter 5 explains the future work and future developments that would improve the system.

1. Experiment Editor

2. Experiment Interface



3. Graphic Language



4. Code Obtained

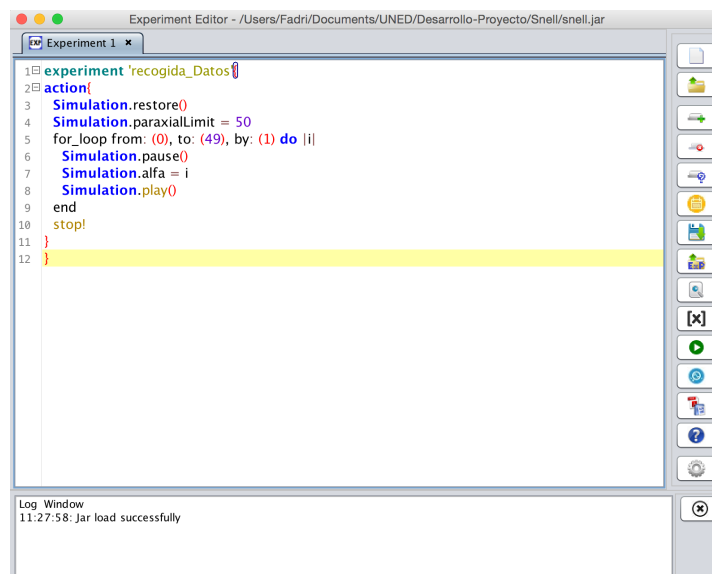


Figure 1. New execution flow

2. STATE OF THE ART

2.1. Visual Block programming

Visual block programming allows users to construct scripts by dragging and dropping language blocks and provides a visual feedback showing the execution of scripts to comprehend how they work [1].

Block programming is a kind of visual programming with the main intention of getting an easy access to the resources obtained when programming. It allows the users to create programs handling elements in a graphic way instead of specifying them by text. The block programming is the one that is developed dragging and dropping blocks connected between them as a puzzle where every type of data, event, command, fix value, boolean, etc, has their own form and one or more blanks and tabs with special shapes in which another pieces can be inserted. Because of the use of blocks, some common issues are avoided as the knowledge of the programming syntax and different structures of a programming language, conditionals or loops for example. As a result, these languages have grown in recent years allowing the inclusion of some functions and the creation of more complex programs.

As it has been seen the code is created dragging and dropping blocks and the scripts in this kind of programming languages are done joining a set of blocks in the scripting area.

The original design of these programming languages was motivated by the needs of young programming learners as it is an easy way to learn the basic concepts. Some programming languages as Scratch or Waterbear allow to create animations in the background and animations using characters getting visual results. All those functionalities were created with the intention of encourage young people to learn about programming concepts using less instructions that the ones used with other programming languages. At the beginning these

languages were used in informal learning settings such as community centers, after-school clubs, libraries, and homes, but increasingly they are used in schools as well [2].

But all these programming languages are not only created to encourage young people to learn about programming concepts but also to introduce programming to those without previous experience. This goal affects in many of the aspects of the design of those languages. Some of the decisions taken while designing them are very obvious, as the choice of a visual block language, the single-window user interface layout, and the minimal command set. Others are more difficult to imagine, as how the target audience influences the type system and the approach to error handling.

2.1.1. Blockly

Blockly is a JavaScript library with the intention of creating a visual block programming editor. Blockly is an open source project and it allows to compile in different programming languages as JavaScript, Dart and Python. Blockly is built using similar principles of Scratch programming language explained in Chapter 2.1.2.

Blockly includes a code editor for web and mobile applications. This editor uses graphic blocks to represent code concepts as variables, logic expressions, loops, etc. It allows the users to apply the programming principles without the need to know the syntax.

From user side, Blockly provides an intuitive and visual way to coding. From application side, Blockly is a set of text blocks that contain written code. There are several advantages when using Blockly [3]:

- Exportable code. Users can extract the block based programs into common programming languages. It allows a smooth transition to text based programming.

- Open source. Blockly is completely open source.
- Extensible. It is possible to modify Blockly to add customized blocks to use in the API or to remove not necessary blocks.
- Highly capable. It can be used to realize complex programming tasks. Calculate the standard deviation of a block for example.
- International. As Blockly has been translated in more than 40 languages, including right to left versions as Arabic or Hebrew.

Blockly is written using JavaScript so it is developed to work in webpages. It also generates code that can be executed independently for several text based programming languages and it is possible to create libraries to convert the code to additional programming languages.

Blockly is the language selected to develop the system explained in this document for several reasons:

- More professional interface which allows to create a more attractive and a more general programming environment.
- Exportable Code. The system explained in this document needs to get the code generated from the blocks to pass it to the Java Virtual Machine. This code will be used to execute the experiments written in Ruby programming language. Because of this, it is necessary that the block programming system used allows the code to be exported.
- Language conversion. It allows to convert the scripts into several programming languages by default and it is also possible to adapt this system to get the code in any other programming language. Ruby, which is the language necessary is not implemented in Blockly but an external implementation with some modifications has been used.

- It can be easily customizable. As it is developed using JavaScript it can be used to have the system in different languages just changing the browser configuration.

2.1.2. Scratch

Scratch is a visual programming language developed in the multimedia laboratory of the Massachusetts Institute of Technology (MIT), with the aim of making easier and funnier the programming learning. It allows the creation of multimedia projects, articles, animated stories, etc. It includes a set of tools for the creation of applications to execute all those functionalities [4].

Scratch was built following some principles [5]:

- More Tinkerable. In the same way a kid starts to play as soon as possible with Lego bricks once received, the process of programming in Scratch tends to be similar to it. That is why the graphical programming blocks have been created to be snapped together to create programs.
- More Meaningful. The value of personalization is very important in Scratch based on two main design criteria:
 - Diversity. Providing support to different kind of projects.
 - Personalization. It is possible to personalize every program as it allows to upload pictures, music clips and record voices among other functionalities.
- More social. The development of the Scratch programming language is based on the concept of sharing. It depends on the support of a community where people participate, collaborate and critique one another and build on one another's work. Such collaborations open opportunities not only to learn but also to create an improved environment.

In this language, as in all the block programming languages, it is not necessary to write commands and the instructions are written connecting blocks. It is based on

some active objects called sprites. Using Scratch it is possible to include more than one sprite at the same time, or make the sprite to follow some instructions.

Scratch also allows the use of event driven programming using sprites. Blocks are concatenated between them as puzzle pieces in such a way that, when they are joined, the instructions specified can be executed in order, generating events.

One of the biggest drawbacks of this programming language is that it does not allow to export the code. It is only possible to export the scripts created into a Scratch file with the format “.sb” but it can only be opened using the web interface.

Another disadvantage is that the execution needs to be done using the interface already created in the webpage and it is not possible to execute it in local so the system cannot be implemented using Java. This situation causes that this language is not feasible for the needed purposes.

2.1.3. Waterbear

Waterbear is a tool to make the programming task easier and more accesible. It is not a programming language but it is a block syntax inspired by Scratch that can be used to represent programming languages.

Waterbear is a drag-and-drop block system developed using HTML5, CSS3 and JavaScript. The main purpose of Waterbear is not duplicate Scratch or create a programming language but to create a visual tool that can be used in a variety of languages and projects [6].

Another aims of Waterbear are to reduce the syntax errors and also to provide a way to reduce some restrictions present in Scratch.

One of the main differences between Waterbear and another block programming languages is that it does not include expressions to get an easier understanding

of the commands as it has been created thinking in a natural use of web technologies, without an intention of changing the paradigm.

As it is a language based on Scratch, the disadvantages when using it are the same as the ones that appeared with Scratch. Therefore it is not possible to export the code but it is possible to export some program created in a specific format only valid for Waterbear which is “.wb”. The execution only takes part in the web interface already created so it is not possible to execute the code generated in any other system. This feature causes that the code cannot be passed to a Java environment.

2.2.Related Work

“Programming is one of the most important competencies that require students to use computational tools to address real world problems in the 21st century” [7]. Because of this, several tools have been developed to provide novice programmers with visual environments that allow them to learn how to construct programs and understand the programming constructions.

The study realized in [1] focuses on the exploration of how novice programmers use the provided visual programming elements to learn by solving a computational problem. In order to do that a visual problem-solving environment is provided to the students to solve computational problems in a visual way.

In [8] a programming environment for visual domain specific languages is developed. It converts the information in multiple general purpose programming languages. Using this environment, the programs are developed combining visual blocks expressed in natural language. These items represent program elements as operations and variables. The main objective of this study is the application of a DSL along with Blockly visual model in order to use the system without requiring a knowledge of programming.

Another similar work is the one explained in [9]. In this document the final aim is to get a domain oriented visual programming environment as a specific instance of a generic environment so the techniques used in it can be applied also in a specific environment and the common functionalities in these environments do not need to be developed from scratch. To carry out this task, a visual programming environment is specified to split the complete development process into multiple standalone functional modules. This is done using a model view controller modified in which the view controller and the image controller are independent of one another so the final model results more flexible.

In terms of visual blocks another similar study was done in [2]. In this case a visual general purpose programming block system is explained. This new language has several features:

- Block based.
- Self-sufficient.
- Simple. As it is created in order to be used for everyone, one of the main purposes of this system is that it should be simple, removing languages features that can be obstacles for the use.

The system more similar to the one explained in this document is the one present in [10]. In this document, Agrawal et al. introduce FabCode, a visual programming environment. Using it, one can create designs that can be manufactured using digital fabrication techniques like 3D printing and laser cutting. In this paper a Blockly based environment is created to apply it in a very specific system.

The environment is formed by the block palettes categories tab, the script area and the 3D canvas. This system is very similar to the one developed in this document as both of them have the block palettes categories tab and the script area. Also both systems create an output that it is used by a different system. The main difference is that this system has been developed to create a 3D modeling

system to create pieces that can be obtained using digital systems as 3D printers and the one described in this document is to be used along with the Experiment Editor tool.

2.3.Limitations

Several environments have been created similar to this one. One of the limitations of those systems is that they have been developed with the intention of teaching children to program instead of trying to teach everyone how to program. Another limitation present is the difficult to learn the different kind of blocks present in these systems. Once one has learnt all the different blocks and where they are located it is not a big problem but until then, it can cause a delay in the software development. Also it is important to note that, even taking into account that all the block programming systems are similar, they are not compatible between them. This situation causes issues as the content developed using one programming language cannot be used in another one. That situation and the one that Blockly is the only block programming language that can be easily converted and exported to another different text based programming languages are the main reasons to select it to develop the solution explained in this document.

2.4.Domain Specific Language

System described in this document is a graphic Domain Specific Language, DSL, applied over another DSL, Experiment Editor. DSL is the opposite to general purpose language or GPL. GPLs as Java or C are used by multiple domains and provide a big number of functionalities. In contrast, domain specific languages provide functionalities for specific fields and they could not pass the Touring test. Therefore coding in a domain specific language is usually shorter, more productive, readable and reusable than coding in a general purpose language.

DSLs can be classified as internal or external. An internal DSL is represented with the syntax of the origin language and it is provided as an API or a library. On

the other hand, an external DSL can use a customized syntax and its implementation is usually a translator that generates code using the language of the origin. An external DSL converts coding in an easier and more abstract way to program compared to general purpose languages.

In this document, the language developed is an external DSL, this DSL will allow to use the Experiment Editor tool in an easier way and it will get a more accesible system.

2.5.Easy Java(Script) Simulations

EjsS is a freeware, open source tool developed in Java, its design is focused on the creation of simulations. It requires programming knowledge but not in an advanced level. The architecture of EjsS derives from the Model-View-Controller paradigm, which philosophy states that the interactive simulations have to be composed of three main parts:

1. The model. It describes the process in terms of variables, which hold the different states of the process and relationships between these variables.
2. The view, It provides a graphical representation, it can be realistic or schematic, of the process states.
3. The control. It defines certain actions that a user can perform on the simulation. In EjsS this step is not performed and it is distributed between the model and the view.

But the applications in this Experiment Editor case are created following only two of those steps:

1. Building the model to simulate using the built-in simulation mechanism of EjsS.
2. Creating the view in order to show the model state and its reaction to changes made by users interactively [11].

EjsS allows the users to define their own simulations introducing a software tool for defining and running them.

EjsS can be used to create simulations from scratch as well as to use simulations already created. It also allows to modify content, as variables, initial values or customized methods inside a high level design. Apart from all those things, it is possible not only to create simulations with experiments that could be done in a laboratory but also to create simulations with experiments that simulate a physic phenomenon, the gravity force for example, which may simplify the task to explain how they are produced or the situations that cause them.

It includes connections with digital libraries. These libraries are repositories of simulations created with EjsS that can be accessed directly from the application.

2.6. Experiment Editor

Due to the high cost of maintaining laboratories and the difficulties to get access to them, to realize tests when the schools are closed for example, a laboratory system, Experiment Editor, was developed to create experiments over laboratories created using Easy Java(script) Simulations, EjsS [12]. “The Experiment Editor for EjsS enables scripting and running experiments on VLS created using EjsS and collecting and analyzing data from them” [13].

It includes several features as the possibility to modify the functionality of the laboratories, to analyze the results obtained during the experiment performing complex or repetitive tasks easily and to improve the learning process using the open inquiry-based experimentation paradigm.

Experiment Editor is composed of three main elements:

- The Laboratory. Used to open and interact with the experiments created using EjsS.

- The Data Tool. It is a toolset formed by graphics and another tools to plot and fit data obtained from the laboratory.
- The Experiments Editor. Software to manage the different experiments. It provides an interface to create and develop experiments.

Experiment Editor uses its own language to define experiments over Virtual Laboratories. Using this tool one can interact with the system as well as define variables, create functions, define types, loops, etc. It is Touring complete as it provides loops, if-then-else commands, supports defining functions, etc.

2.7. Blockly system

One of the problems existing when using the experiments created with the Experiment Editor tool is that it is necessary a minimum knowledge of programming to even create a simple experiment. The solution applied with Blockly not only allows to create programs in an easier way but it also serves as a way to learn how to construct them. In addition, one of the biggest problems when using this tool is the time that takes to create an experiment coding and testing it interacting with the elements. In contrast, one preferable way to code the experiment would be using an intuitive, user friendly experiment language that also allows to run it automatically and avoids the possible syntax issues.

The main goal of this system is to provide a tool to enrich the Experiment Editor tool with a visual system that can also execute the experiments. In order to get this, a new Application Programming Interface (API) has been created as well as several blocks that can be converted into different programming languages so the use is not limited to this environment.

The solution created is an open source system and it is introduced as a new method to apply in every system needed.

2.8. Block Factory

The blocks have been created as substitutes to the code so the design has been done trying to get the same values in the blocks and just avoid the step to write them to construct the experiments. In order to create the system, the Block Factory provided with Blockly, Figure 2, was used to create the blocks at the beginning of the development.

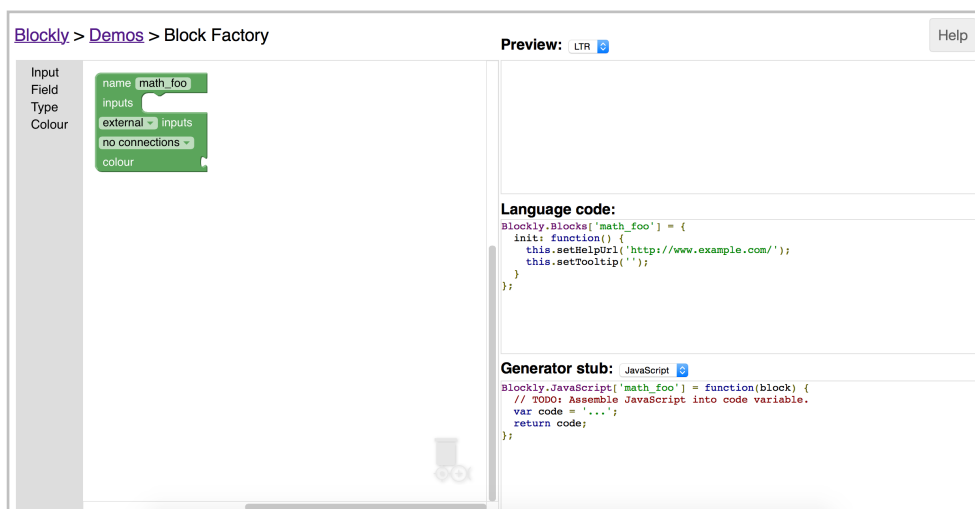


Figure 2. Block Factory Interface

This factory allows to create blocks using one block in which the different parameters of the block that it is going to be created can be specified. Different parameters can be selected as it can be seen in the Figure 3.

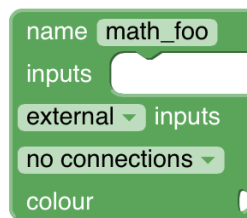


Figure 3. Block to create blocks

The first parameter that can be specified is the name of the block, this name is going to be used to differentiate the block from the others therefore, it is important to have an unique identification per block so they can be called separately. The

next parameter is the inputs. In this field one of the pieces that appear in the Figure 4 can be introduced. Depending on the piece selected the input parameter will be a value, a statement or a dummy. It also allows to select the name of the input and the type of the input value. This name will be the reference to get the value of the variable in the code. Once selected the type, it won't allow blocks from other type than the one specified to be used as an input in the block blank.



Figure 4. Input value blocks

Referring to the types of values allowed, several are included, Figure 5. It is possible to allow one specific, more than one type between some specified, select one type created for the user or allow any.

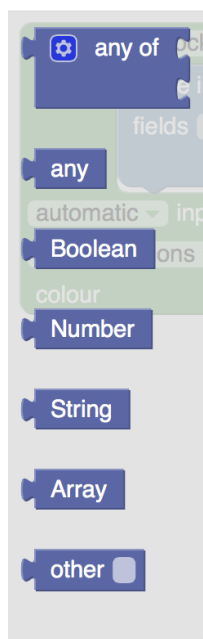


Figure 5. Types of blocks

Inside input value there is a hollow in which a piece can be introduced to select the name that will appear associated with the input blank. This name should be clear if an intuitive system is necessary. The different values that can be used in the name of the field can be seen in the Figure 6. Inside the possible options available one can find:

- Text
- Text with a default value
- Angle
- Numeric input
- Tab with several options
- A color
- A variable

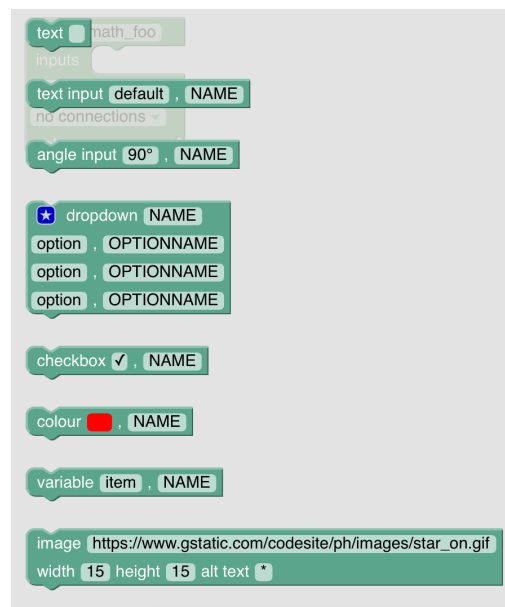


Figure 6. Field Values

The next parameter that can be defined is the situation of the inputs. There is a scroll bar in which three options are available:

- Automatic. Value by default, the input value is introduced from the right side.
- External. Similar as automatic, the input values are introduced from the right side.



Figure 7. Automatic and External Input

- Inline. The input value has to be introduced inside the block as it can be seen in Figure 8.



Figure 8. Inline input

The next option is the block connections. There are several options as shown at Figure 9.

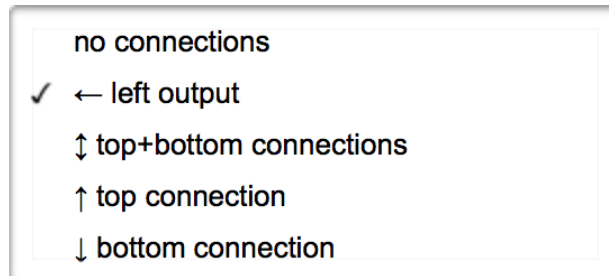


Figure 9. Connection Options

Depending on the option selected the block will have connections at the left, above, below or above and below. The connections above and below allow the user to concatenate several blocks in order to get more complex programs getting event driven programs.

The following field is to select the type of the connections. These types are the same as the ones explained for the inputs and can be selected for all existing connections.

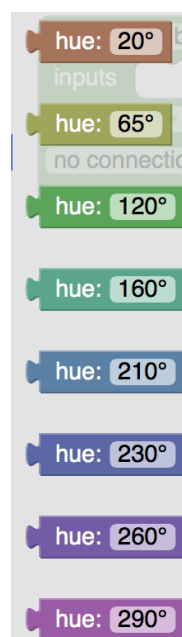


Figure 10. Color Grades

Finally the last eligible component is the color of the block. This one can be selected from a group of colors specified by their grade, Figure 10.

After creating the block using this method some items will appear on the left divisions:

- Preview. It shows a visual preview of the final block. There is also a tab to select the preferred side to receive the inputs. It is very useful when creating blocks for right to left languages.



Figure 11. Preview Screen

- Block Definition. In this tab the block is defined with all the visual elements that determine it. It also includes a tab to get the code in JavaScript, JSON or just to edit it manually.

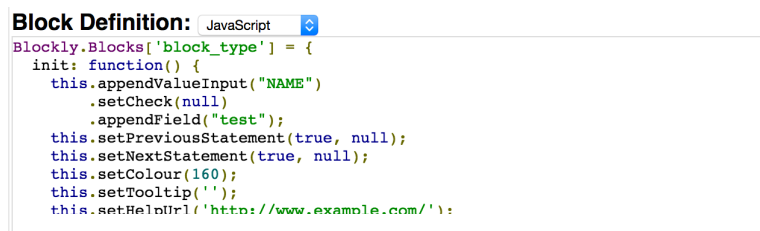


Figure 12. Block Definition

- Generator Stub. In this screen is where the code obtained from the block should be specified. As it can be implemented with several programming languages it allows to generate this part of the code in different languages.

Generator stub: JavaScript

```
Blockly.JavaScript['block_type'] = function(block) {  
  var value_name = Blockly.JavaScript.valueToCode(block, 'NAME', Blockly.JavaScr  
  // TODO: Assemble JavaScript into code variable.  
  var code = '...;\n';  
  return code;  
};
```

Figure 13. Generator Stub

3. GRAPHIC LANGUAGE

3.1. Development

During the development phase several steps were done. At the beginning all the code was created using the Block Factory but once all the rules were known some blocks were created coding them directly.

To do that it is only necessary to use the Blockly library with the values needed populated. Once done a code like the one showed in Code 1 is obtained.

In order to get the code in Ruby, Blockly2Ruby project was used. It is an open source project that implements Blockly generators for the Ruby programming language. This project contains the next folders that are specified in its description [14]:

- generators: the JavaScript scripts that generate Ruby code, these generators are based on the Python generators.
- test: tests to validate the generators and Javascript scripts that generate Blockly unit tests.
- Ruby_tester: Ruby code to run the generated tests.
- compiler: Javascript scripts that allow to compile from XML to Ruby using command line (using <http://phantomjs.org/>).
- compiler.sh; shell script to execute the JavaScript compiler from the command line.
- server.rb: a Ruby script that runs a server to run Blockly on a local box.


```
Blockly.Blocks['complex_event'] = {
  init: function() {
    this.appendValueInput("event")
      .setCheck("String")
      .appendField("Event");
    this.appendStatementInput("action")
      .setCheck("String")
      .appendField("action");
    this.appendStatementInput("check")
      .setCheck("Boolean")
      .appendField("check");
    this.appendValueInput("type")
      .setCheck(["Zero crossing", "State
        Event", "Positive Crossing"])
      .appendField("type");
    this.appendValueInput("interactions")
      .setCheck("REAL")
      .appendField("Interactions");
    this.appendValueInput("method")
      .setCheck(["BISECTION", "SECANT"])
      .appendField("method");
    this.appendValueInput("tolerance")
      .setCheck("REAL")
      .appendField("tolerance");

    this.appendValueInput("end_step_at_event")
      .setCheck("Boolean")
      .appendField("end step at event");
    this.setPreviousStatement(true, "String");
    this.setNextStatement(true, null);
    this.setColour(160);
    this.setTooltip('');

  }
};
```

Code 1. Complex Event Block

But not all those components are important for the original purposes of this study. The only component needed and used was the “generators”. It includes several files to convert the different common Blockly blocks into Ruby code. These files are similar, as they have the same file names and blocks definition, to the ones existing for another programming languages. The files necessary to get the same general purpose language code as any other programming language with Blockly are the following:

- “logic.js” it includes all the logic functions defined in any general purpose language, if, else or logic operators i.e.
- “loops.js” in this file all the different loops are defined as for or while loops.
- “math.js” it includes mathematical operators as addition, subtraction, multiplication or division.
- “text.js” in this file different operations with strings are defined as a string printer or an empty string check.
- “lists.js” these are different operations that can be done with a list as list creation or calculation of the length of the list.
- “colour.js” functions to determine colors using RGB scale or selecting the color from a palette.
- “variables.js” includes functions to create a variable and to assign a value to it.
- “procedures.js” it contains empty functions to add operations inside them. It also includes a return to get the return value.

```
Blockly.Ruby['experiment_with_events']  
= function(block) {  
  var value_experiment =  
  Blockly.Ruby.valueToCode(block,  
  'experiment',  
  Blockly.Ruby.ORDER_ATOMIC);  
  var value_reacts_to =  
  Blockly.Ruby.valueToCode(block,  
  'reacts_to',  
  Blockly.Ruby.ORDER_ATOMIC);  
  var statements_name =  
  Blockly.Ruby.statementToCode(block,  
  'statements') || '\n';  
  var code = 'experiment ' +  
  value_experiment + '{ \n' +  
    'reacts_to ' + value_reacts_to +  
    '\n' +  
    'action{\n' +  
    statements_name  
    + ';\n}\n';  
  return code;  
};
```

Code 2. Block Ruby Code

All those JavaScript files have to be included in the head of the index file. Apart from those files another one should be added, “Ruby.js” file. This file includes several functions and it is the file in which the blocks created should be added. Following the previous example, the code implemented for the experiment with events can be seen in Code 2. In this specific case it can be seen that the different inputs are defined as variables that can be used in the code. The code structure of this block can be seen in the Figure 14. This table has the different input elements highlighted specified by the expected type of data of each input value.

Name	Experiment with Events
Structure	<pre>experiment STRING { reacts_to [STRING LIST] action { FUNCTIONS } }</pre>

Figure 14. Experiments with Events Structure

The code generated with the block is the same as the one specified in the previous figure. The block creates the common part of the code to the function, the other values are taken from the different input variables and, with them, it returns the complete function in text.

Once the block is situated in the Script Area the code present in Code 3 will appear in the Code Generation Area.

```
experiment {  
  reacts_to  
  action{  
  
  ;}  
}
```

Code 3. Experiment Text Examples

This code is incomplete as it does not have any input parameter so if the code is executed this way it is not going to work properly. But it generates this code because it is the base code to create the function. Therefore once the inputs are added, a correct function will appear.

An example of the experiment with events with some input values added can be seen in Figure 15.

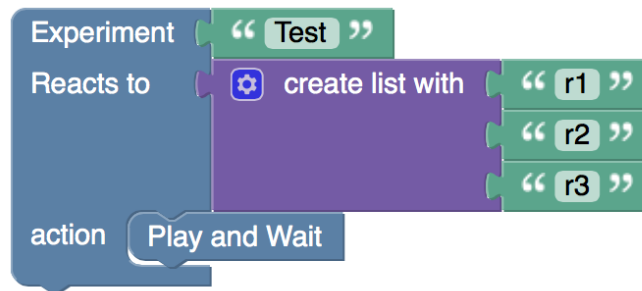


Figure 15. Example of Experiment with Events

In this example input values have been introduced in all the fields and the code obtained with Blockly can be seen in Figure 16. In this case we can see the code complete as it would appear in an usual use of the function.

```
experiment 'Test'{
  reacts_to ['r1', 'r2', 'r3']
  action{
    Simulation.playAndWait()
  };
}
```

Figure 16. Complete Experiment with Events Function

In this example all the values introduced as input appear in the code generated.

3.2. Interface creation

The interface has been developed using HTML as well as JavaScript with two main objectives. The first one is to try to get one interface as much easier to use as possible because one of the main objectives desired is to get a system that can be used for every person even without knowledge of programming. The second one is to offer an interface either functional or attractive.

To get this objectives, a simple interface has been designed trying to get a clear, easy to understand environment.

At first sight, Figure 17, it can be seen that the design is based on Blockly original design. It is because it offers a very clear and simple design. In this interface it is

easy to understand where all the elements are located. It is also very clear to see the blocks once situated in the workspace.

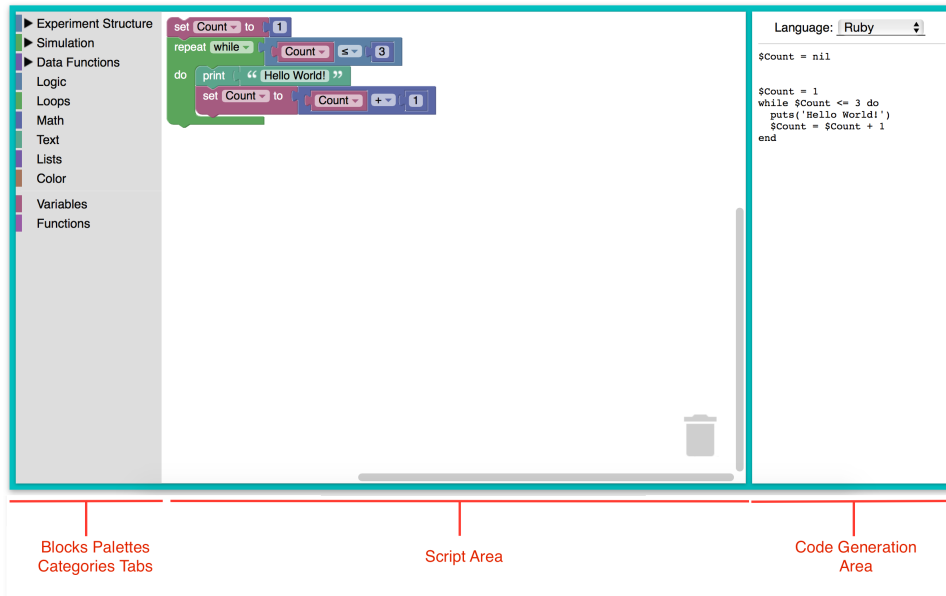


Figure 17. Graphic Interface

There are three different areas situated in the visual interface. This interface is divided by:

- Blocks Palettes Categories Tabs. It is situated at the left side of the interface. It includes a tab that can be opened to see the different eligible blocks separated in different categories. In this tab the blocks can be selected in order to drag-and-drop them into the Script Area.
- Script area. In this place is where the blocks should be situated to create the scripts. It also includes a trash to remove the blocks that are not necessary.
- Code Generation Area. In this area the code generated from the blocks situated in the Script Area appears. The programming language in which the code appears is selected using the tab situated above. The code appears here for two main purposes:
 - Check that the blocks created generate the correct code once located in the workspace.

- Allow the users to check what they are doing so it can serve as a way to learn how the code should be written for the experiments.

It is important to note that the code workflow is implemented not only for Ruby but also for JavaScript. This is done because this system developed has not been created to be used only with the Experiment Editor tool but also in any other similar system.

3.3.Blocks Palettes Categories Tabs

In the left part of the blocks workspace it is possible to see the different classifications used to differentiate the blocks in the interface, Figure 18.

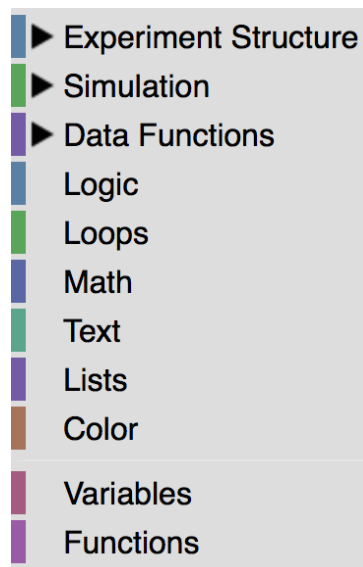


Figure 18. Blocks Classification

- Experiment Structure. Inside this menu it is possible to find different blocks that represent different types of experiments. The experiment blocks are prepared to include one set of input parameters depending on the configuration required.
- Simple Experiment. It represents a simple experiment. In this kind of experiment some parameters are specified as the experiment name or the different functions that are going to be executed in it. The kind of block associated to this experiment can be found in the Figure 19. This kind of

block allows two inputs. The upper one is to introduce the name of the experiment and it allows only strings and the second one allows to introduce functions as inputs.



Figure 19. Simple Experiment Block

```
Blockly.Ruby['simple_experiment'] = function(block) {  
  var value_name = Blockly.Ruby.valueToCode(block, 'name'  
    , Blockly.Ruby.ORDER_ATOMIC);  
  var statements_name = Blockly.Ruby.statementToCode(block,  
    'statement') || '\n';  
  var code = 'experiment ' + value_name + '{' +  
    '\naction{\n' +  
    statements_name + '};\n}\n';  
  return code;  
};
```

Code 4. Simple Experiment Code

- Experiment with Events. This kind of experiment is more complex than the previous one as it includes a list of parameters to which it reacts. In this case the block created is the one that appears in Figure 20. In this picture it is possible to see the new input. This input has to be a list of strings.

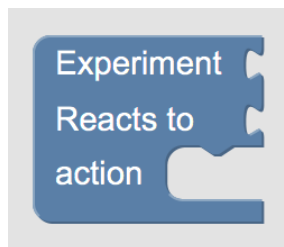


Figure 20. Experiment with Events Block

- Event. This is the part of the menu in which the blocks related to events are selected. Inside this category two events have been defined:

- Simple Event. This event is the simplest that can be defined in which only the name of the event, the functions that are going to be executed until the condition specified in “check” is reached, the condition, the iterations and tolerance parameters are specified.

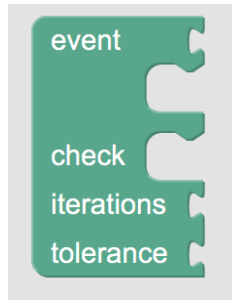


Figure 21. Simple Event Block

- Complex Event. This kind of event is similar to the previous one. The main difference between them is that this one allows to specify additionally the type of event, the method employed and the final condition of the event, Figure 22.



Figure 22. Complex Event Block

- Condition. Blocks used to execute one action once the condition is achieved.
- Simple Condition. This kind of block just specify the condition that needs to be achieved to execute the functions that are defined inside it.

- Simulation. In this tab, the simulation variables and the simulation functions are defined:
 - Variables. It is where the simulation variables are defined. These variables are created from the input parameters received in the Java program. The functionality to create these variables is explained in detail in the 3.10 section.
 - Functions. Functions used to determine the status of the experiment execution. Between these variables it is possible to find [13]:
 - Play. Run the VL.
 - Pause. Pauses the VL.
 - Restore. Resets completely the VL.
 - Play and Wait. Delays the execution until the condition is reached.
 - Stop. Finishes the experiment.

3.4. Java Development

The experiments have been developed to be executed in an implementation of Ruby of the Java Virtual Machine so the code returned by the interface needs to be returned in a Java program. In order to reach this objective a main class has been created using the JavaFX library to open the JavaScript system using Java as a browser and getting the code to be executed in the JVM.

“JavaFX is a Java library that consists of classes and interfaces that are written in Java code” [15]. In particular, the JavaFX WebView has been used in this example as the purpose was to get the webpage executed from Java. The WebView is a web component to embed web pages within a JavaFX application. JavaScript running in WebView can call Java APIs, and Java APIs can call JavaScript running in WebView. The final result is an interface developed using

Java that can open the “index.html” file as if it was a web browser and get the results of the execution in the Java code.

```
String path = "/index.html";  
webEngine.load(this.getClass()  
                .getResource(path).toExternalForm());
```

Code 5. Load method call

As it is possible to see in Code 5 the browser opens the index file loading it using the method “load()”. This method executes the content of any HTML file as if it was a browser. As the index file is included as a Java resource, the call to get the path to open it is done using the “getResource()” method.

This browser has three main purposes:

1. Open the JavaScript system using Java.
2. Create the simulation variable blocks using the input parameters received in Java.
3. Get the code obtained in Java using the Blockly system.
4. Be an easy to use system.

To reach those objectives the design of Java part was done to accomplish all of them. In order to get that, four buttons have been defined, Figure 23, one to get the code in Java, “Code”, the next one, “Export Code”, to export the code generated into a Ruby file, the “Export Blocks” button allows to export an XML file saving the blocks situated in the Script Area, the following one, “Import Blocks” is to get the blocks saved previously in an XML file and the last one, “Exit”, serves to exit the system.

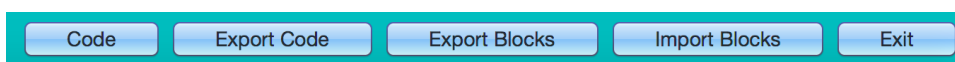


Figure 23. Java Interface Buttons

3.5.Code Button

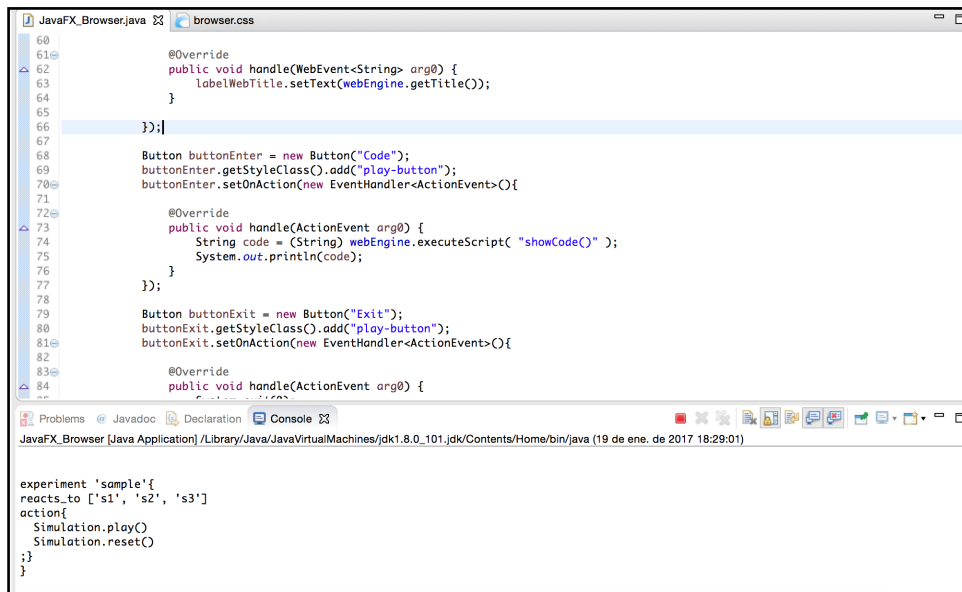
The button to get the code just calls a JavaScript function called “showCode()”. This function, Code 6, returns the code that has been generated using the blocks. In order to do that it calls the method “workspaceToCode()”. This other method gets the blocks situated in the Script Area in that moment and generates the code associated to them.

To get the code generated in Java the method “executeScript()” is called passing the name of the function as input parameter and it returns the code generated in a String value. This String text returned is saved in a variable in Java and this variable is passed as an input to the Java Virtual Machine of Ruby.

```
function showCode() {  
    // Generate JavaScript code and display it.  
    Blockly.Ruby.INFINITE_LOOP_TRAP = null;  
    var code = Blockly.Ruby.workspaceToCode();  
    return code;  
}
```

Code 6. showCode function

It is important to note that this function has only been defined for Ruby but another function can be created to get the code in any other language, JavaScript for example. Once this button is pressed the code appears in Java automatically. An example of a code obtained as an input and showed in screen can be seen in Figure 24.



```
60
61 @Override
62 public void handle(WebEvent<String> arg0) {
63     labelWebTitle.setText(webEngine.getTitle());
64 }
65
66 }
67
68 Button buttonEnter = new Button("Code");
69 buttonEnter.getStyleClass().add("play-button");
70 buttonEnter.setOnAction(new EventHandler<ActionEvent>(){
71
72     @Override
73     public void handle(ActionEvent arg0) {
74         String code = (String) webEngine.executeScript( "showCode()" );
75         System.out.println(code);
76     }
77 });
78
79 Button buttonExit = new Button("Exit");
80 buttonExit.getStyleClass().add("play-button");
81 buttonExit.setOnAction(new EventHandler<ActionEvent>(){
82
83     @Override
84     public void handle(ActionEvent arg0) {
85
86     }
87 });
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

Figure 24. Demonstration of code obtained in Java.

3.6.Ruby interpreter

In order to get the results, a Ruby interpreter for Java is necessary to execute the Ruby code generated. To do that the interpreter selected has been JRuby. As it is defined in [16] by Justing Edelson and Henry Liu “JRuby is an open source implementation of the Ruby programming language for the Java Virtual Machine (JVM). JRuby allows Ruby applications to be run within a Java Virtual Machine and interface with libraries written in either Java or Ruby. Although the JRuby project was initiated in 2001, interest in JRuby has grown significantly over the last few years, reflecting an overall growth in interest in Ruby sparked by the success of the Ruby on Rails framework”.

JRuby provides a complete set of core “builtin” classes and syntax for the Ruby language, as well as most of the Ruby Standard Libraries. The standard libraries are mostly Ruby’s own complement of “.rb” files, but a few that depend on C language-based extensions have been reimplemented. Not all of them have been implemented as some are still missing [17].

```
public void run(String code) {  
  
    Ruby = new ScriptingContainer();  
    // Method to execute the Ruby Code  
    Ruby.runScriptlet(code);  
  
}
```

Code 7. JRuby Code to Execute Ruby Code

As it can be seen in Code 7, the method created to execute the code just use another method defined in JRuby library, “runScriptlet()”. This method evaluates a script and returns a result only if the script has a value to return. Right after the parsing, the script is evaluated once.

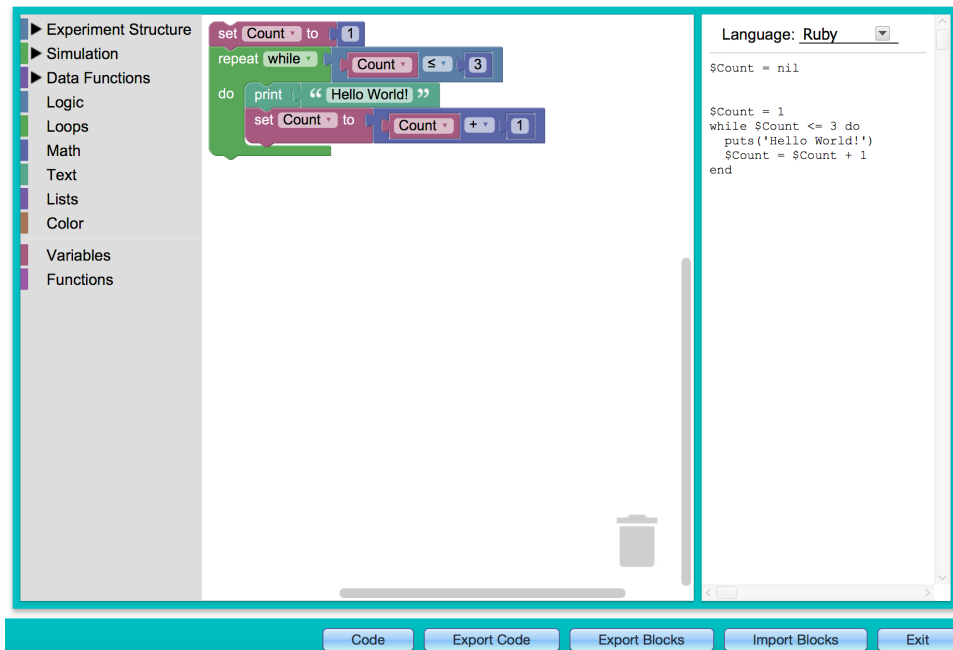


Figure 25. Ruby Code Execution

After using this method the values obtained in the output are the results of the Ruby code execution so the Java Virtual Machine just executes the code and get the results. This library only executes the common part of Ruby code so to run all the code including the functions defined in the Experiments Editor explained in

[11] is necessary to implement the system completely with the Experiment Editor.

One example of Ruby code execution can be seen in Code 8.

```
$Count = 1
while $Count <= 3 do
  puts('Hello World!')
  $Count = $Count + 1
end

Hello World!
Hello World!
Hello World!
```

Code 8. Result of an Execution

3.7.Export Code Function

This system allows not only to generate code during the execution using blocks but also to export it in a separate Ruby file that contains fully executable Ruby code.

A button, Export Code button, has been added in the interface to provide the functionality to get the code in a Ruby file.

The code developed to get the button can be seen in Code 9. This button gets the code from the JavaScript system and creates a new file inserting the code in it. As this system has been created to create Ruby code, the output is always created with the extension of a Ruby file “.rb” so the files can be read easily from a system as if they had been created writing, as usual. To obtain this button JavaFX has been used using the class “FileChooser”.

```
Button buttonExport = new Button("Export Code");  
  
buttonExport.getStyleClass().add("play-button");  
buttonExport.setOnAction(new EventHandler<ActionEvent>()  
{  
    @Override  
    public void handle(ActionEvent arg0) {  
        String code = (String)  
        webEngine.executeScript( "showCode()" );  
        FileChooser fileChooser = new FileChooser();  
        fileChooser.setTitle("Save Ruby file");  
        File file = fileChooser.showSaveDialog(stage);  
        FileWriter outputFile;  
        try {  
            outputFile = new FileWriter(file.getAbsolutePath()  
            + extension);  
            outputFile.write(code);  
            outputFile.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
});
```

Code 9. Export Code Button

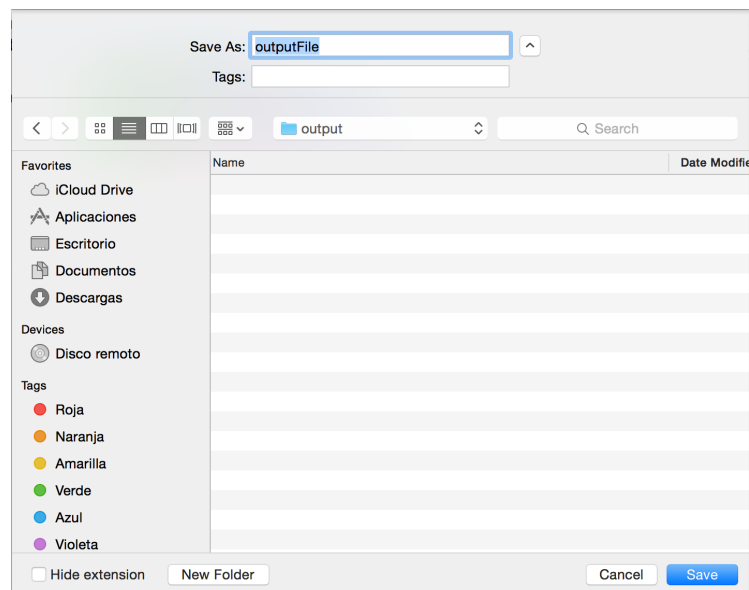


Figure 26. Select Output File Directory Screen

Once the button is pressed, a new pop up window appears asking for the name of the output file.

This file explorer window allows the user to select the folder in which the file will be saved. Automatically a Ruby file will appear in the folder with the code generated.

The files generated can be used to execute the laboratory system or to save an experiment created to use it later.

3.8.Export Blocks

Using this functionality it is possible to export the blocks situated in the Script Area into an XML file. This is a very important functionality as it allows to export the scripts generated using blocks so it can be considered as a way to save the script generated to be loaded later.

The functionality is similar to the one for exporting the code. Once the button is pressed a file explorer window appears in order to select the folder in which the XML file will be saved. This file will always be an XML file containing the information of the blocks present in the script in the moment they are saved because it is the format in which Blockly methods store the Script Area.

```
function exportBlocks(){  
    var xml = Blockly.Xml.workspaceToDom(workspace);  
    var xml_text = Blockly.Xml.domToText(xml);  
    return xml_text;  
}
```

Code 10. exportBlocks Function

As it can be seen in Code 10, a function in JavaScript has been created to export the code into an XML file. What this function does is to convert the blocks situated in the workspace in an XML text using the methods “workspaceToDom”

and “domToText”. This text is returned as the output of the function. This function is called from Java using the method “executeScript()” and the returned text is passed to a “FileWriter” object in order to write the text in an XML file selecting the folder using the file explorer. As the blocks only can be stored in XML format, it is the only one possible.

```
String code = (String)
webEngine.executeScript( "exportBlocks()" );
FileChooser fileChooser = new FileChooser();
fileChooser.setTitle("Save Block file");
File file = fileChooser.showSaveDialog(stage);
try {
    FileWriter outputFile = new
FileWriter(file.getAbsolutePath().concat(extensionXML));
    outputFile.write(code);
    outputFile.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

Code 11. XML File Creation

3.9.Import Blocks

Exporting blocks would not have sense if they cannot be imported so a function for it has also been created. The function can be seen in Code 12. This function gets the text passed as an input parameter and converts it into blocks using the methods “textToDom” and “domToWorkspace”.

```
function importBlocks(xml_text){
    var xml = Blockly.Xml.textToDom(xml_text);
    Blockly.Xml.domToWorkspace(xml, workspace);
}
```

Code 12. importBlocks Function

To get the blocks into the system an additional button has been added. Once pressed, similarly to the “Export Code” button and to the “Export Blocks” button, a file explorer window is opened but in this case it is not opened to create a new file but to open an existing one. As the blocks are always exported in XML format, this explorer will only allow to open XML files.

```
FileChooser fileChooser = new FileChooser();
FileChooser.ExtensionFilter extFilter = new
FileChooser.ExtensionFilter("XML Files", "*.xml");

fileChooser.getExtensionFilters().add(extFilter);
fileChooser.setTitle("Load Block file");
File file = fileChooser.showOpenDialog(stage);
FileReader inputFile = null;
try {
    inputFile = new FileReader (file.getAbsolutePath());
    BufferedReader b = new BufferedReader(inputFile);
    StringBuilder xmlCode = new StringBuilder();
    String line = null;
    while ((line = b.readLine()) != null){
        xmlCode.append(line);
    }
    b.close();

    webEngine.executeScript( "importBlocks(\"" +
        xmlCode.toString() + "\")");
} catch (FileNotFoundException e1) {
    e1.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Code 13. XML File Creation

The code used to import the file can be seen in Code 13. In this code a “FileChooser” object is defined, then the extensions allowed are specified, in this case only XML files as it is the only format in which the blocks can be exported. The next steps are done to get the content of the file in a String and, at the end, the function created in JavaScript is called as explained before using the method

“executeScript()”. The code read is passed as an input argument of the function called. Therefore the JavaScript function is called with the code and it converts the XML file data into blocks.

3.10. Input Variables

```
function addElement(name, value, block_type){
    var xml =
        '<block type="variables_set">' +
        '<field name="VAR">' + name + '</field>' +
        '<value name="VALUE">' +
        '<block type="' + block_type + '">' +
            '<field name="NUM">' + value + '</field>' +
        '</block>' +
        '</value>' +
        '</block>' +
        '<block type="variables_get">' +
        '<field name="VAR">' + name + '</field>' +
        '<value name="VALUE">' +
        '<block type="' + block_type + '">' +
            '<field name="NUM">' + value + '</field>' +
        '</block>' +
        '</value>' +
        '</block>';
    $(' .Variables').append(xml);

    Blockly.updateToolbox(document.getElementById('toolbox'));
}
```

Code 14. addElement method

The expected functionality of the system is the following:

- Experiment Editor is opened. It includes a set of simulation variables initialized.

- The visual programming system explained in this document is launched. It receives the simulation variables as arguments and creates the blocks related to them.
- Experiment development. Using the graphic language explained in this document.
- Result passed to the Experiment Editor to get the execution results.

Therefore the block system has to include a method to use the input values to automatically generate new variable blocks with them. In order to do that, a method, “addElement()”, has been created in JavaScript file using jQuery. This method, Code 14, has three input parameters: name of the variable, value of the variable and variable type. With those values it creates the text of two variable blocks, one setter and one getter. Once created, it adds the code into the “Variables” category in such a way that the blocks will appear in the corresponding tab of the Block Palettes Categories Tab. In order to situate them in the correct tab inside the Block Palettes Categories Tab the method “append” is used specifying the category in which the blocks should be inserted. Finally, the toolbox is updated using the Blockly method “updateToolbox()” to get the new values in the tab added automatically.

The code introduced using the function “addElement()” uses two different methods already created in Blockly. These methods are used to create the simulation variables in the same way that Blockly uses to create them. Set method, Code 15, is a variable setter to create variables during runtime. Get method, Code 16, just creates a block to get the value of the variable defined in the block to use it as input parameter. The variables defined using these methods can be created before the execution but they can also be created during runtime using the block “create variable”, Figure 27.



Create variable...

Figure 27. Create variable block

Create variable block, Figure 27, can be used to create new blocks during runtime. Once clicked, a prompt appears allowing the user to introduce the name of the new variable. After introducing the name a new set of blocks is created.

```
Blockly.Ruby['variables_set'] = function(block) {  
  // Variable setter.  
  var argument0 = Blockly.Ruby.valueToCode(block, 'VALUE',  
    Blockly.Ruby.ORDER_NONE) || '0';  
  var varName = Blockly.Ruby.variableDB_  
    .getRubyName(block.getFieldValue('VAR'),  
    Blockly.Variables.NAME_TYPE);  
  return varName + ' = ' + argument0 + '\n';  
};
```

Code 15. variables_set method

```
Blockly.Ruby['variables_get'] = function(block) {  
  // Variable getter.  
  var code = Blockly.Ruby.variableDB_.getRubyName  
(block.getFieldValue('VAR'),Blockly.Variables.NAME_TYPE);  
  return [code, Blockly.Ruby.ORDER_ATOMIC];  
};
```

Code 16. variables_get block

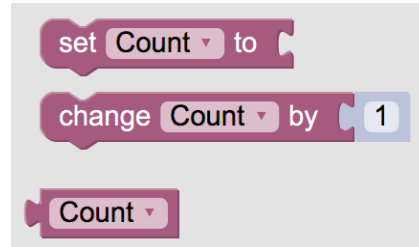


Figure 28. Variable block

The set created, Figure 28, includes a block to select the variable with a blank to assign a value to it, another one to change the value of the variable selected and the last one to use the variable as an input. Only the first block and the third one will appear when the simulation variables are defined using the method explained in the 3.10 section to get a simpler interface as the other block is not indispensable.

To create the simulation variables, they have to be received as input values in the Java program. The program has a listener, Code 17, to execute the “addListener()” method. This listener has been defined to execute the method when the browser is launched automatically with no need to do anything to run it. This name can be a simple name or it can be a name with the value together with the format “name=value”. If the value is received in this last format, the system takes the name and the value separately and check what type of data is the value, checking between String, Double and Boolean. Once checked, it calls the “addElement()” method passing the name, the value and the type of data of the value as input parameters. This method creates the corresponding blocks with that information. If the input parameter received is only the name, the system creates a variable with value equal to zero which is the default value.

```
webEngine.getLoadWorker().stateProperty().addListener(  
    new ChangeListener<State>() {  
        public void  
changed(@SuppressWarnings("rawtypes") ObservableValue ov,  
State oldState, State newState) {  
    if (newState == State.SUCCEEDED) {  
        for (String variable : inputValues){  
            String type = variable;  
            String value = "0";  
            String block_type = "math_number";  
            if (variable.contains("=")){  
                type = variable.split("=")[0];  
                value = variable.split("=")[1];  
                try {  
                    Double.parseDouble(value);  
                    block_type = "math_number";  
                } catch(NumberFormatException ex){  
                    if (value.equals(StringTrue) ||  
                        value.equals(StringFalse)){  
                        Boolean.parseBoolean(value);  
                        block_type = "logic_boolean";  
                    } else{  
                        block_type = "text";  
                    }  
                }  
            }  
            type = "Simulation.".concat(type);  
            webEngine.executeScript( "addElement('" + type + "','" +  
value + "','" + block_type + "')" );  
        }  
    }  
});
```

Code 17. addElement() listener

As the simulation variables are not Ruby variables several changes have been applied in the code already existing for the simulation variables:

- Blockly2Ruby system contains a check to ensure that the variables created do not have special characters. To fix this problem once it finds any special

character the system converts it into an underscore. After the change applied only the variable names that do not start their name with "Simulation." will change the special characters, the other ones will remain the dot in their names. It had to be done because the dot was considered a special character and the system changed it to underscore.

- Variables declaration was done before the change using the dollar symbol before the variable names every time they are used in the workspace but simulation variables, as they are not usual variables, they should not work that way. Therefore they do not have to have dollar symbol when initialized or when used in the program. To fix it the same check that was applied before has been used but when writing the variables in the program avoiding the dollar symbol to appear before the simulation variable names.

4. CONCLUSIONS

This work appears in the simulations scope with the intention of creating a graphic language to be used inside the Experiment Editor tool developed by the Department of Computer Science and Automatic Control, Universidad Nacional de Educación a Distancia and the Department of Software Engineering and Computer systems, Universidad Nacional de Educación a Distancia.

The creation of this system was done to cover some needs inside this scope and to provide alternatives to get a more accesible system. Different solutions and options have been investigated from this point as well as the reutilization of some systems already developed in order to modify and adapt them to this particular scenario.

All of this has resulted in a theoretical solution in which the graphic system is explained seeing all the benefits and advantages that it can provide. Furthermore, the different elements and functionalities that it should contain to be used together with the Experiment Editor have been also described.

Additionally a Blockly based implementation has been proposed. This proposal can be used as a substitute of the step to code the experiments and provide an alternative for all the people that want to use the Experiment Editor but do not know how to code the experiments. The graphic language developed is included in a graphic interface which converts it in a very intuitive system.

The graphical language explained in this document uses the principles of automatic generation of code learned during the Software Engineering Master studies as it generates the code automatically from the blocks. Another principles learned during the Master studies have been applied as secure software principles applied to manage errors or to remove parts of the code and libraries not used to avoid security issues.

Thanks to this document not only an easy to use system has been obtained but also an exportable system that can be used with other different projects or environments.

It is important to note that the system described in this document is not a substitutive but it is a system to enrich the Experiment Editor already created to facilitate its use.

4.1. Results

Some of the achievements reached using this system have been:

1. A complete functional interface has been created with several options to interact with. The complete interface can be seen in Figure 24, it contains the JavaScript interface with the Block Palettes Categories Tab, the Script Area and the Code Generation Area getting a very intuitive and visual system. Furthermore, an additional interface has been created allowing the user to perform different functions as executing the code, exporting the Ruby code, exporting the blocks in the Script Area into a file and importing them.
2. Block based system to develop Ruby code without the need of writing code.
3. Importation of the variables defined in the experiment creation and creation of the corresponding blocks.
4. Code generated is returned to Java so it is not necessary to copy it manually from the graphic language to the Experiment Editor.
5. Ruby code execution in Java using a JVM.

5. FUTURE DIRECTIONS

The work explained in this document is not complete and it can be improved adding several functionalities. This chapter summarizes some ideas and functionalities that can be added to expand the work carried out in this Master's Thesis.

5.1.Import Code

A function that allows import code in the system would be an interesting improvement in order to get the code in the Script Area that was previously exported. Several methods and functions have to be defined in JavaScript to get this functionality. These functions have to convert the code into different blocks analyzing the syntax and the constructors and choosing the blocks that correspond to the input code.

This functionality would get also a system to develop scripts and to modify them when necessary as it is done usually modifying the code but with blocks.

5.2.Adapt the system

As it has been seen previously this system has been created not to use it only with Ruby and the Experiments Editor but also with any environment necessary. This is an open source project so it can be used for any environment or situation needed.

5.3.JVM Complete Implementation

In order to get one of the main functionalities developed it is necessary to implement a Java Virtual Machine of Ruby along with the code generated for the Experiments Editor. Another option to do this is to implement the system in the experiments one and create the experiments using this tool.

5.4.DataTool implementation

This tool is an open source project to plot data, to compute basic statistical information, to compute linear, quadratic or cubic fits, showing the slope and the area under a curve, etc. It is available at OSP (<http://www.compadre.org/osp/items/detail.cfm?id=7331>) and it is already implemented in the Experiment Editor tool and the corresponding block categories have been added in the Blocks Palettes Categories Tab. Future improvements should include the implementation of this functionality to have a fully executable system.

6. REFERENCES

- [1] Po-Yao Chao, Exploring students' computational practice, design and performance of problem-solving through a visual programming environment, *Computers & Education*, Volume 95, April 2016, Pages 202-215, ISSN 0360-1315
- [2] Y. Ohshima, J. Mönig and J. Maloney, "A module system for a general-purpose blocks language," *Blocks and Beyond Workshop (Blocks and Beyond)*, 2015 *IEEE*, Atlanta, GA, 2015, pp. 39-44.
- [3] Blockly Wiki, Accedido 3 de Junio de 2016. URL: <https://github.com/google/blockly/wiki>
- [4] Blocks, Scratch Wiki, Accedido el 31 de Mayo de 2016. URL: <https://wiki.scratch.mit.edu/wiki/Blocks>
- [5] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all Commun. *ACM*, 52 (11) (2009), pp. 60–67
- [6] Waterbear Wiki, Accessed January 2017. URL: <https://github.com/waterbearlang/waterbear/wiki>
- [7] Einhorn, S. (2011). *Microworlds, computational thinking, and 21st century learning*. Logo Computer System Inc, White Paper. Retrieved from <http://www.microworlds.com/>
- [8] Azusa Kurihara, Akira Sasaki, Ken Wakita, Hiroshi Hosobe, A Programming Environment for Visual Block-Based Domain-Specific Languages, *Procedia Computer Science*, Volume 62, 2015, Pages 287-296
- [9] Da-Qian Zhang and Kang Zhang, "On the design of a generic visual programming environment," *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*, Halifax, NS, 1998, pp. 88-89.
- [10] Harshit Agrawal, Rishika Jain, Prabhat Kumar, and Pradeep Yammiyavar. 2014. FabCode: visual programming environment for digital fabrication. In *Proceedings of the 2014 conference on Interaction design and children (IDC '14)*. ACM, New York, NY, USA, 353-356.
- [11] Ruben Heradio, Luis de la Torre, Daniel Galan, Francisco Javier Cabrerizo, Enrique Herrera-Viedma, and Sebastian Dormido. 2016. Virtual and remote labs in education. *Comput. Educ.* 98, C (July 2016), 14-38.
- [12] Daniel Galan, Ruben Heradio, Luis de la Torre, Sebastian Dormido, Francisco Esquembre, Performing Automated Experiments with EJS Laboratories, *IFAC-PapersOnLine*, Volume 48, Issue 29, 2015, Pages 134-139, ISSN 2405-8963
- [13] Daniel Galan, Ruben Heradio, Luis de la Torre, Sebastian Dormido, Francisco Esquembre, *The Experiment Editor: Supporting Inquiry-Based Learning with Virtual Labs*.

- [14] Jean Lazarou, Blockly2Ruby project, GitHub repository, <https://github.com/jeanlazarou/blockly2ruby>
- [15] JavaFX Overview (Release 8) *Docs.oracle.com*. Retrieved 2017-08-01.
- [16] Justin Edelson and Henry Liu. *JRuby Cookbook*. "O'Reilly Media, Inc.", 2008
- [17] «Rails Support». JRuby Team. Accessed January 2017