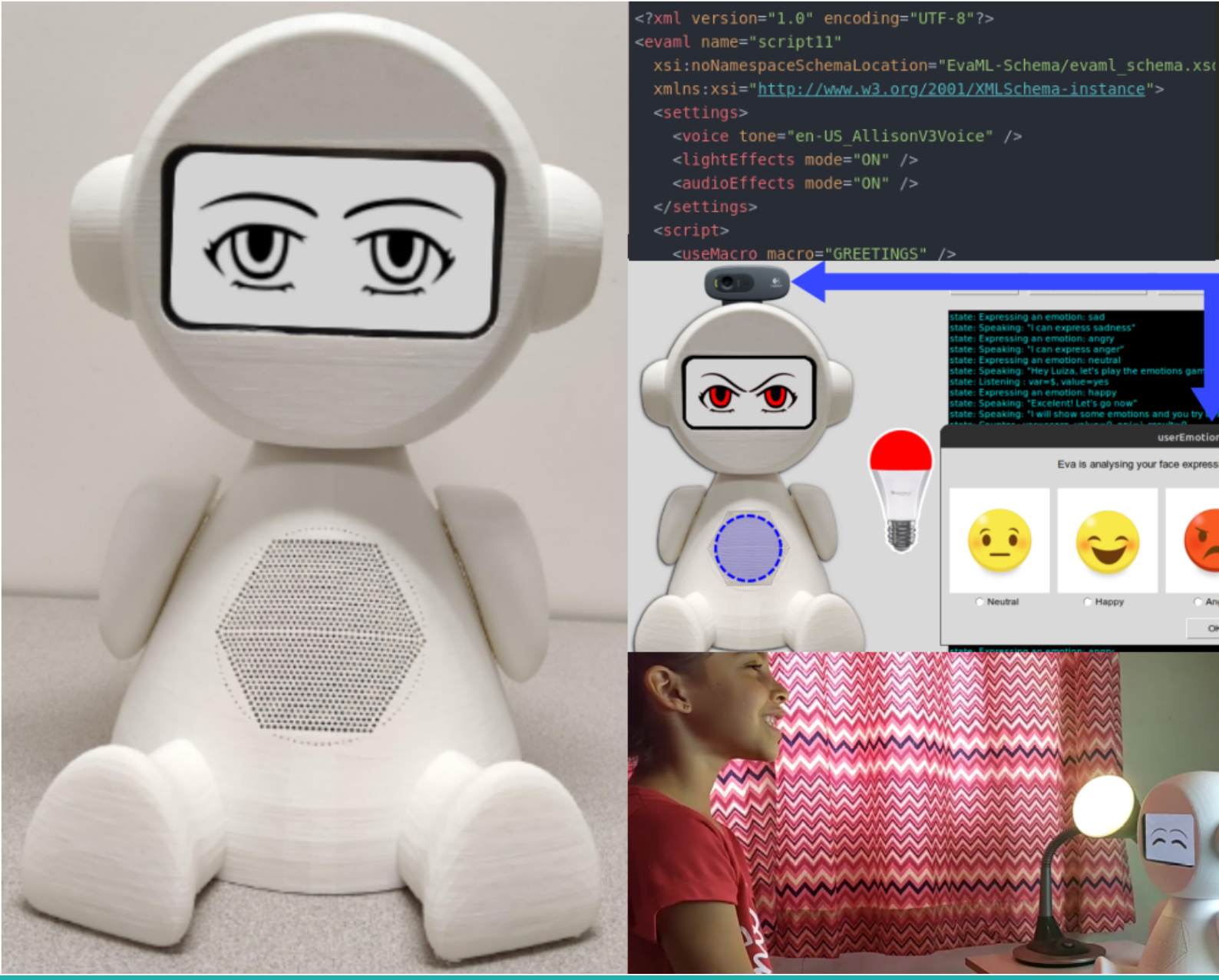

EvaML and EvaSIM

An XML-Based Language for Specification of Interactive Sessions and a Simulator for Robot EVA

Author: Marcelo Marques da Rocha

Email: marcelo_rocha@midiacom.uff.br

Institute: Fluminense Federal University - MídiaCom Lab - UFF

Version: 1.0

Contents

Chapter 1 Introduction

Assistive robots are expected to become ubiquitous by transforming everyday life and to be widely used in healthcare therapies [11]. Socially Assistive Robots (SARs) are a class of robots that are at an intersection between the class of assistive robots and that of interactive social robots [2, 10]. The purpose of SARs is not just to provide some kind of assistance, but to provide stimuli through interaction with the robot. SARs are being explored to assist in the diagnosis and treatment of children with ASD (Autism Spectrum Disorder) [8]. The SAR that we will present is called EVA. EVA was developed by CICESE (Centro de Investigación Científica y de Educación Superior de Ensenada), Baja California, in Mexico, and it has been used in therapies with patients with dementia and Alzheimer's disease [1]. Several studies [4, 9] show that the use of SAR technologies can:

- enrich healthcare interventions;
- facilitate communication between therapists and patients;
- support data collection and assessment in the diagnosis of patients.

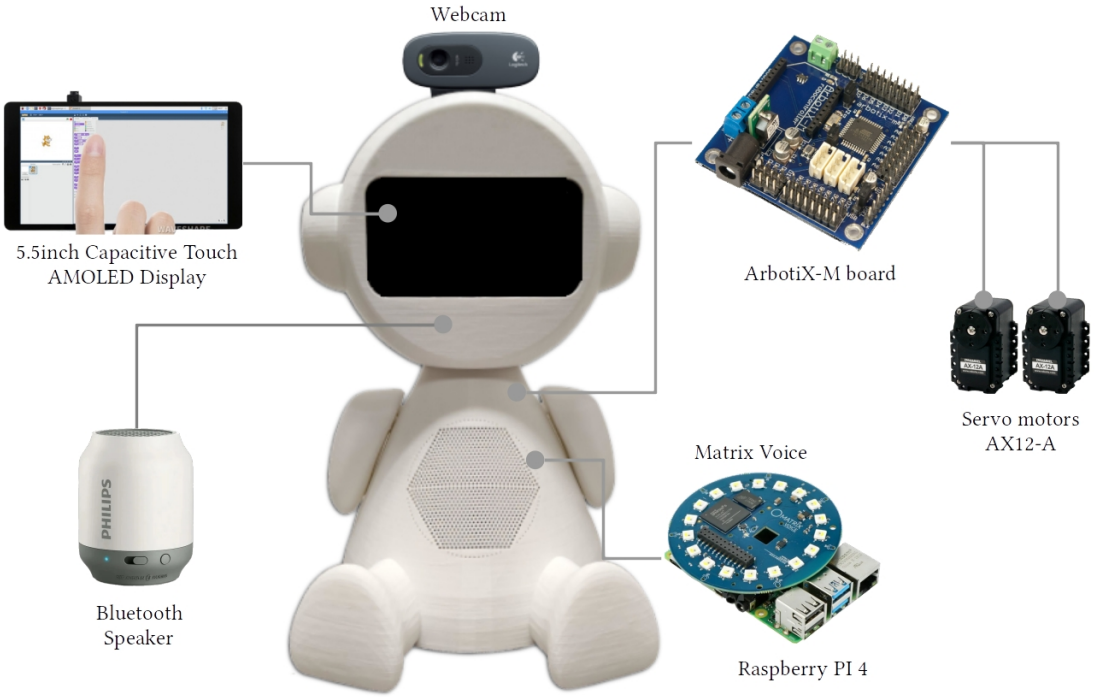

Figure 1.1: Eva - Hardware Components

Figure 1.1 illustrates the hardware components of Robot EVA. First, we have a 5.5" touch display. Through that screen, the robot is able to express itself through the eyes. We also have a bluetooth speaker. On the right, we can see the controller board with two servo motors, this set is responsible for moving the robot head, and lastly, we have the Matrix Voice[1] board connected to a Raspberry PI 4[2]. The Matrix Voice has 7 microphones

[1] https://store.matrix.one/collections/frontpage/products/matrix-voice-esp32-wifi-bt-microcontroller

[2] https://www.raspberrypi.org

and eighteen RGB LEDs. EVA, in its basic version, has verbal and non-verbal communication skills. It is capable of:

- recognizing the human voice (using Google API) and is able to speak (using IBM Watson service);
- expressing emotions through the eyes;
- moving its head;
- running animations with the LEDs on its chest.

Originally, robot EVA has a Visual Programming Language (VPL), which aims to facilitate the development of interactions by people without experience in computer programming. The VPL was built using a powerful graphical tool called GoJS[3]. During the process of creating a script using VPL, the user is responsible for inserting robot commands, which are nodes in the execution flow. However, the elements are automatically positioned in the graphical area of the interface by the GoJS application. Script graph nodes are automatically repositioned after removing or inserting an graph node or edge. When a script becomes large, this automatic repositioning often makes it difficult to build the script. As the number of graphic elements in the interaction script increases, the graph visualization becomes very limited. To get an overview of the graph, the interface allows you to use the zoom in and zoom out features, however, with the reduced size of the elements, it is very difficult to visually identify each one of them. You can select a group of nodes and edges and duplicate them using copy and paste, the problem is that the script starts to have different elements (which can have different properties) with the same visual identifier (its name). This makes it very difficult when you want to edit the properties of an element, because when we select an element for editing, we are not sure if we are selecting the correct element, as we have more than one element with the same name. So, it was observed that it would be important to develop a language that could be suitable for people with some programming knowledge but still maintain a reasonable level of readability.

When you have to choose between a Domain Specific Language (DSL) or a General Purpose Language (GPL), you have to choose between expressiveness and generality [5]. Within an application domain, DSLs have gains in expressiveness and ease of use compared to GPLs. DSLs also show gains in productivity, maintenance cost and requires less domain and user knowledge. The idea of creating a textual DSL for the EVA robot, arouse during an experimentation process, that investigated the acceptance of the use of the robot as an assistive tool applied in therapies with patients with Autistic Spectrum Disorder (ASD) [7].

With the objective of facilitating the development of interactive sessions by people with technical knowledge in programming, but also aiming to maintain the readability of the script codes, EvaML was created. It is an XML-based language for developing interactive sessions and makes it possible to create interaction scripts for the EVA robot using just a simple text editor. All the robot's multimodal interaction controls are present in EvaML, among them, the *Light* component control (which controls the smart bulb), the speech and voice recognition commands and the *userEmotion* command that enables the recognition of the user's facial expression through the webcam. The language also has elements for creating and manipulating variables, generating random numbers, conditional controls using *switch* and *case* elements and so on. The EvaML parser automatically generates a corresponding script to be run by the robot's hardware.

In order to enable testing interactive sessions built with the EvaML language, an EVA robot simulator software was developed. EvaSIM is the software that simulates the robot's multimodal interaction elements and behavior. It has a display that shows the robot's eye expressions, simulates a smart light bulb, uses IBM Watson

[3]https://gojs.net/latest/index.html

to transform text into speech, it simulates voice recognition capability (using a text-based interface), simulates facial expression recognition, can play audio files and so on. In addition to the robot components, the simulator also has a terminal emulator that displays the robot states and the actions being performed. It also displays the contents of the variables being used by the script being run. EvaSIM receives as input and executes the XML code generated by the EvaML parser.

This manual presents the EvaML language, its structure as an XML document its elements and attributes. For each element of the language, XML code examples are presented. The remaing of this text has the following structure: in Chapter 2, the motivation for using the XML language as the basis of the EvaML language, the structure of an EvaML document and its sections are presented. Still in Chapter 2, the manual brings a complete reference to all language commands, accompanied by EvaML code examples. At the end of that chapter, the EvaML parser is introduced. In Chapter 3, the EVA robot simulator software, EvaSIM, is presented. There is a description of the elements of its graphical interface and the simulation processes of some multimodal interactions.

Chapter 2 EvaML

2.1 An XML-Based Language for an Open-Source SAR

Although the use of graphical tools makes an average user be productive, a user with advanced skills in the application domain can be reduced in efficiency [6]. The XML-based DSL development goals are: enabling greater control in the entry and editing of language commands and its respective parameters; adding abstractions of programming elements that aim to facilitate the construction of scripts and enabling the development of scripts independent of the robot's control interface, that is, using any text editor. The XML language has many advantages for developing DSLs:

1. The XML language is more readable to non-programmers than the GPLs.
2. In an XML-based DSL, the grammar can be described using DTD (Document Type Definition) or a document of type XML-Schema.
3. It is simple to parse the XML structure using the DOM (Document Object Model).

Therefore, we used the XML language and the tools related to it to propose and implement EvaML, the markup language to specify interactive sessions for robot EVA.

2.2 EvaML - Document Sections

We can see in Figure 2.1 an EvaML script that shows the document root element <evaml>, with its name attribute that defines the script name, and which contains the following three elements: the <settings>, <script> and <macros>.

Figure 2.1: EvaML - Document Sections

Table 2.1 shows the root element of the EvaML document (<evaml>) and the elements <settings>, <script>

and <macros> that represent the sections of the document. You can also observe the attributes of each element and its contents. In the attribute column, an underlined attribute indicates that it should be used. In the column of contents, occurrence indicators are used to indicate the order and number of times an element can occur. The "**,**" (comma) symbol indicates that all child elements listed must be used in the sequence shown. The "**|**" (pipe bar) indicates that either element can occur within the parent element. The "**+**" (plus sign) symbol, on the other hand, indicates that the child element must appear one or more times. The "*****" (asterisk) symbol indicates that the element may be used zero or more times within the parent element. The "**?**" (question mark) indicates that the element is optional, the element may not exist or there is only one occurrence of it.

Table 2.1: EvaML - Document Elements (Root and Main Elements)

Element	Attributes	Content
evaml	name	(settings, script, macros?)
settings		(voice | lightEffects? | audioEffects?)
script		(random* | wait* | talk* | stop* | light* | goto* | motion* | loop* | userEmotion* | evaEmotion* | useMacro* | listen* | audio* | led* | counter* | switch*)
macros		(macro+)

2.2.1 <settings>

In this first section, some global characteristics of the script are defined in the <settings> element. It is possible to define how the voice tone and the language in which the robot will communicate. It is also possible to define whether the generated code will perform light effects, sound effects or even play music. By configuring these parameters, it is possible to globally modify the operation of the script without having to directly change the definitions of its individual elements. Here is an example from the <settings> element.

```
1 <settings>
2   <voice tone="en-US_AllisonV3Voice" />
3   <lightEffects mode="ON" />
4   <audioEffects mode="ON" />
5 </settings>
```

2.2.2 <script>

The <script> element contains the sequence of commands that the robot must execute. We can see some of them on following code snippet. We can see in line 2 of the script, the <light> command that lights the smart bulb setting its color to blue. Next we have the <talk> command, which makes the robot say something, for example, introducing itself. The <wait> command on line 4 causes the script to pause for 2000ms (2s). In the next line, the <audio> command plays an audio file named "*mario-start*". Then the robot speaks "bye" and turns off the smart bulb. A detailed explanation of each of these commands will be presented in Section 2.3.

```
1 <script>
2   <light state="ON" color="BLUE" />
3   <talk>Hi, I am robot EVA</talk>
4   <wait duration="2000" />
5   <audio source="mario-start" block="TRUE" />
```

```
  <talk>Bye</talk>
  <light state="OFF" />
</script>
```

2.2.3 <macros>

The <macros> element is one of the abstractions created in the EvaML language. As you can see in the next code snippet, it is possible to create **macros** that can be referenced within the <script> element. A macro has the id attribute that serves to identify it. These macros can be used within the <script> section using the <useMacro> command. The macro attribute of the command <useMacro> references the <macro> element defined in the <macros> section. During the *parsing* process of the EvaML document, macros are expanded with their code in the <script> section. There is no limit to the number of macros created, nor to the number of references to these macros within the script. As can be seen in Table 2.1 the **macros** section is not mandatory.

```
<script>
  <useMacro macro="START" />
</script>
<macros>
  <macro id="START">
    <talk>Hello, I'm robot Eva. I'll pick a number from one to four</talk>
    <talk>For each one, I'll turn the bulb on in a different color</talk>
    <talk>I will only terminate when the number drawn is three</talk>
  </macro>
</macros>
```

2.3 EvaML - Command Reference

Before presenting each EvaML command with its description and usage examples, Table 2.2 shows EvaML elements that represent the language commands. Its attributes and what each element can contain will also be listed. For such representation, the same notation used in Table 2.1 will be used. The description of each symbol used can be seen in Section 2.2.

2.3.1 <voice>

The <voice> command does not produce any action, it actually sets a parameter that defines, at the same time, the voice tone (its gender) to be used by the robot, and the language that will be taken into account during the Text-To-Speech (TTS) process from the IBM Watson service. It defines only one attribute, tone and as we presented previously, in the attribute column, an underlined attribute indicating that it should be used. As it is a configuration command, it is placed inside the <settings> section. In Table 2.3, we can see a small list with some voice options for the robot. In the following code, we select the "*en-US_AllisonV3Voice*" option, which is female.

Table 2.2: EvaML - Document Elements (Commands)

Element	Attributes	Content
voice	tone	empty
lightEffects	mode	empty
audioEffects	mode	empty
random	id, min, max,	empty
wait	id, duration	empty
talk	id	text
stop		empty
light	id, state, color	empty
goto	target	empty
motion	id, type	empty
userEmotion	id	empty
evaEmotion	id, emotion	empty
useMacro	macro	empty
listen	id	empty
audio	id, source, block	empty
led	id, animation	empty
counter	id, var, op, value	empty
switch	id, var	(case+, default?)
macro	id	(random* | wait* | talk* | stop* | light* | goto* | motion* | loop* | userEmotion* | evaEmotion* | listen* | audio* | led* | counter* | switch*)
case	op, value	(random* | wait* | talk* | stop* | light* | goto* | motion* | loop* | userEmotion* | evaEmotion* | useMacro* | listen* | audio* | led* | counter* | switch*)
default		(random* | wait* | talk* | stop* | light* | goto* | motion* | loop* | userEmotion* | evaEmotion* | useMacro* | listen* | audio* | led* | counter* | switch*)
loop	id, var, times	(random* | wait* | talk* | stop* | light* | goto* | motion* | loop* | userEmotion* | evaEmotion* | useMacro* | listen* | audio* | led* | counter* | switch*)

Table 2.3: Voice options from IBM Watson (TTS)

Code	Gender	Dialect
pt-BR_IsabelaV3Voice	Female	Brazilian
en-US_AllisonV3Voice	Female	American
en-US_EmilyV3Voice	Female	American
en-US_HenryV3Voice	Male	American
es-LA_SofiaV3Voice	Female	Spanish (Latin American)
es-ES_EnriqueV3Voice	Female	Spanish

```
<settings>
  <voice tone="en-US_AllisonV3Voice" />
</settings>
```

2.3.2 <random>

The <random> command generates a random integer in the closed range [min, max]. The command has two attributes, min, representing the lower limit, and max, determining the upper limit of the random number to be generated. The value generated by the random function is stored in a special region of the robot's memory, which works with a array. The **$** character accesses the element at the end of the array. This detail will be explained more clearly later in the text. In the following example, a random integer in the range [1, 3] will be generated and the robot will speak a text, including the generated value, using the **$** character.

```
<script>
  <talk>I will randomly choose a number from 1 to 3</talk>
  <random min="1" max="3" />
  <talk>The number drawn was $</talk>
</script>
```

2.3.3 <wait>

The <wait> command pauses the script execution. The duration attribute is expressed in milliseconds. In this example, the command causes the script to pause for one second (1000ms).

```
<script>
  <talk>I will wait for one second</talk>
  <wait duration="1000" />
  <talk>Ready!</talk>
</script>
```

2.3.4 <talk>

The <talk> command makes the robot speak the specified text. The character **$**, by convention, in the original EVA language, accesses the value at the end of the array in memory that stores the answers returned

by commands that interact with the user, such as the commands <listen> and <userEmotion>. As mentioned in Section 2.3.2, it also stores the value generated by the <random> command. Inside a text, you can use the character **$** to include the last answer (a command <listen> return, for example), **$1** to include the first answer, or **$-1** to include the second-to-last answer. Separating the sentences with the character **/** causes the robot to randomly speak one of the sentences. Let us look at an example that applies all these ideas.

```
1 <script>
2   <talk>What is your name?</talk>
3   <listen />
4   <talk>Hello $ / Hi $ / Hi $, how is it going</talk>
5   <random min="1" max="3" />
6   <talk>Hello $1, I sorted number $</talk>
7 </script>
```

In this code snippet, the first <talk> command, in line 2, makes the robot ask for the username. Then the <listen> command causes the robot to capture the audio and send it to the Google API service which returns the audio converted to a string, which will contain the username. This value (in string format) is stored in the robot memory. In line 4, the command <talk> contains 3 sentences separated by the character **/**, which will make the robot, randomly, speak one of the sentences. These sentences contain the **$** character which accesses the value at the end of the array in memory. This value is the answer obtained via the <listen> command. The **$** will be replaced with the username and then the resulting string will be sent to the IBM Watson TTS process. In the next line, the <random> command generates a random number between 1 and 3 and stores it in the memory, at the end of array. In line 6, the robot speaks the name of the person, that can be accessed by referring to **$1** (*first answer*) and can reference the value generated by random using **$** (*last answer*). Figure 2.2 shows this special region of EVA memory that functions like a array and can have its elements referenced by the **$** character. The **$1** represents the oldest element in the array and **$** represents the newest element, while the **$-1** represents the predecessor of **$**.

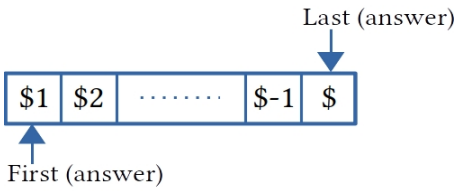

Figure 2.2: The EVA Robot Memory Array View

2.3.5 <stop>

The <stop> command stops the script from running. In the following example, the command that lights the bulb green will not be executed, as the script is stopped by the command <stop>. The example is not very good, as it does not make sense to put any command after a <stop> command, but the purpose is just to show that the script will be terminated before the <light> command in line 5. The use of the <stop> command is best justified when used inside a <case> command in a <switch> block. Its use allows interrupting an execution flow created by the <case> command.

```
<script>
  <light state="ON" color="RED" />
  <wait duration="1500" />
  <stop />
  <light state="ON" color="GREEN" />
</script>
```

2.3.6 <light>

The <light> command controls the smart bulb and has two attributes. The state, which can assume the values "ON" and "OFF" and the color attribute, which defines the bulb color. This color can be indicated using the RGB hexadecimal representation "#00ff00" or some of the elements from the predefined list: "WHITE", "BLACK", "RED", "PINK", "GREEN", "YELLOW", "BLUE".

```
<script>
  <light state="ON" color="RED" />
  <wait duration="1000" />
  <light state="ON" color="#00ff00" />
  <wait duration="1000" />
  <light state="ON" color="BLUE" />
  <wait duration="1000" />
  <light state="OFF" />
</script>
```

Note *If the state is "OFF", the color attribute value will not be used. In this specific case, the color attribute can be ommited.*

2.3.7 <goto>

The <goto> command switches the execution flow to the command with id referenced in its target attribute. The id attribute defines the label that will be used as a value in the target attribute of the <goto> command. In this example, the script is looped.

```
<script>
  <light id="BEGIN" state="ON" color="RED" />
  <wait duration="1000" />
  <light state="OFF" />
  <wait duration="1000" />
  <goto target="BEGIN" />
</script>
```

As seen in Table 2.2, some commands do not have the id attribute and therefore cannot be referenced as "**targets**" of <goto> commands. These are the commands: <voice>, <stop>, <goto>, <macro>, <useMacro>, <case> and <default>.

2.3.8 <motion>

As mentioned in Section 1 the robot can move its head. The combination of robot expressions with head movement can increase the robot's expressiveness. The command <motion> is responsible for controlling the movement of the robot's head. It has only one attribute called type and it can assume the following values: "YES", "NO", "CENTER", "LEFT", "RIGHT", "UP", "DOWN", "ANGRY", "2UP" , "2DOWN", "2LEFT", "2RIGHT". Setting type to "YES" makes the robot perform a *yes* movement with its head. Using the value "NO" causes the robot to signal a *no* with its head. The options "LEFT", "RIGHT", "UP", "DOWN" are responsible for moving the robot's head in the corresponding directions. The value "ANGRY" causes the robot to tilt its head downwards. The "CENTER" option makes the robot head return to its original, centered position. The values "2UP", "2DOWN", "2LEFT", "2RIGHT" execute the movements in the corresponding directions, however, with twice the amplitude.

```
<script>
  <talk>I will move my head to the right</talk>
  <motion type="RIGHT" />
  <talk>I am very angry</talk>
  <light state="ON" color="RED" />
  <evaEmotion emotion="ANGRY" />
  <motion type="ANGRY" />
</script>
```

2.3.9 <userEmotion>

The robot is able to recognize facial expressions through a webcam. It does this using an external module written in Python that runs as a service within the same device that runs the robot software, a Rasperry PI 4.

The process of capturing the user's facial expression is as follows: EVA sends a facial recognition request to the Python module, the module receives the request, activates the webcam and returns the following expressions as a string: "NEUTRAL", "ANGRY", "DISGUST", "FEAR", "SURPRISE", "HAPPY" and "SAD". This answer can then be used in the rest of the script. Let us see an example.

```
1 <script>
2   <light state="ON" color="GREEN" />
3   <talk>Hi! How are you feeling?</talk>
4   <userEmotion />
5   <audio source="mario-fundo" block="TRUE" />
6   <talk>I see that you feel $</talk>
7 </script>
```

The <userEmotion> command (line 4), starts the process of capturing the user's facial expression (via the webcam) and returns it as a string that can be referenced via the $ character, as seen in Figure 2.2. This is an asynchronous (non-blocking) function and needs 3s to 5s to return the detected expression. In the code, you can see on line 5, that an audio file is played. This audio has 7s and its execution allows the Python facial recognition module (running in parallel) to have the necessary time for its correct functioning. In line 6, the robot speaks the text replacing the $ with the expression (string) returned by the <userEmotion> command.

Note *As we can see in Figure 3.3, the EVA robot simulator works with only 5 types of expressions, while the physical robot's facial recognition module can return up to 7 types of facial expressions.*

Note *Line 5 of the example code, the one that plays the audio file, would not be necessary in the robot simulator since the facial recognition function, in this case, is synchronous and blocks the code execution flow.*

2.3.10 <evaEmotion>

The <evaEmotion> command controls the robot's gaze expressions. The emotion attribute can take the following values: "NEUTRAL", "HAPPY", "SAD", "ANGRY". Figure 2.3 shows the EVA gaze expression set.

Here is a short example.

```
<script>
  <evaEmotion emotion="HAPPY" />
  <talk>I am feeling happy</talk>
  <wait duration="1000" />
  <evaEmotion emotion="SAD" />
  <talk>I am feeling sad</talk>
  <wait duration="1000" />
</script>
```

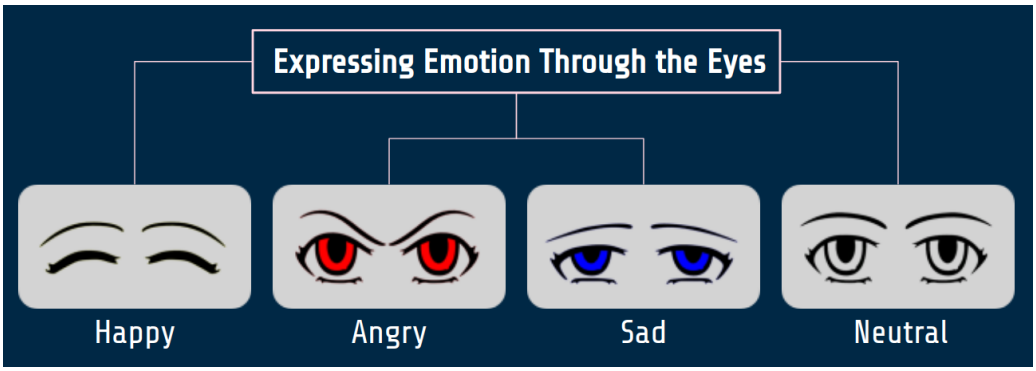

Figure 2.3: EVA Gaze Expressions

2.3.11 <macro>

As we saw in Section 2.2.3, the <macro> command defines a sequence of commands that can be referenced anywhere in the script using the <useMacro> command. It has the id attribute which is used to identify it. The following example shows a declaration of two macros. It is important to note that, as shown in Table 2.2, a <macro> element cannot contain the definition of another *macro* and cannot, through the <useMacro> command, refer to another *macro*.

```
<macros>
  <macro id="START">
    <talk>Hello, I'm robot Eva. I'll pick a number from one to four</talk>
    <talk>For each one, I'll turn the bulb on in a different color</talk>
    <talk>I will only terminate when the number drawn is three</talk>
  </macro>
  <macro id="END">
```

```
    <talk>I'm leaving now / It's getting late</talk>
  </macro>
</macros>
```

2.3.12 <useMacro>

We have seen how macros can be defined using the <macro> command, now we will see how they can be used within the script. The <useMacro> command causes the macro code, referenced in its macro attribute, to be expanded by the code where the <useMacro> command is declared. This macro expansion process occurs in the first step of the EvaML language parsing process and will be explained in Section 2.4.1. The command <useMacro> has only the attribute macro that refers to the macro that will be expanded. In the following example, the macros created in the <macro> command example, in Section 2.3.11, are used.

```
<script>
  <useMacro macro="START" />
  <useMacro macro="END" />
</script>
<macros>
  <macro id="START">
    <talk>Hello, I'm robot Eva</talk>
    <talk>I'm a socially assistive robot</talk>
    <talk>I can speak</talk>
    <talk>I can recognize facial expressions</talk>
  </macro>
  <macro id="END">
    <talk>I'm leaving now / It's getting late</talk>
  </macro>
</macros>
```

2.3.13 <listen>

As mentioned in Charpter 1, the robot can recognize the human voice and for that it uses the Speech-To-Text (STT) service in the Google cloud API. Audio capture is done using the 7 microphones on the Matrix Voice board. An important point to note is that there is latency in this process, as the captured audio needs to be sent to the cloud, be processed there and only then can the text resulting from the STT process be returned to be used by the robot software. This latency will depend on the internet connection speed. The answer (in string format) can be accessed via the **$** character.

```
<script>
  <talk>Say what you are thinking</talk>
  <listen />
  <talk>Wow! You are thinking of $</talk>
</script>
```

2.3.14 <audio>

The <audio> command plays an audio file contained in the "*sonidos*" folder[1]. It has two atributes, source and block. The source attribute must indicate the file name only, without the folder path and without the extension. EVA only plays files in wav format. The block attribute can have the folowing values, "TRUE" or "FALSE" and it defines whether the file's play should block the script's execution, that is, the command that comes after the <audio> command will only be executed after the audio play is finished. In the folowing example, the bulb will only turn off after the "mario-start" audio file has finished playing.

```
1 <script>
2   <light state="ON" color="RED" />
3   <audio source="mario-start" block="TRUE" />
4   <light state="OFF" />
5 </script>
```

2.3.15 <led>

This command starts the animation with the LEDs on the EVA robot's chest. The animation attribute can take the following values: "HAPPY" (green), "SAD" (blue), "ANGRY" (red), "STOP" (no color/off), "SPEAK" (blue), "LISTEN" (green) and "SURPRISE" (yellow). The folowing example uses some animations.

```
1  <script>
2    <led animation="HAPPY" />
3    <wait duration="1000" />
4    <led animation="SAD" />
5    <wait duration="1000" />
6    <led animation="ANGRY" />
7    <wait duration="1000" />
8    <led animation="SPEAK" />
9    <wait duration="1000" />
10   <led animation="LISTEN" />
11 </script>
```

Note *Some LED animations are associated with other robot commands and are automatically activated with them.*

2.3.16 <counter>

The <counter> command creates or modifies a variable in EVA memory. The var attribute defines the variable name. The op attribute defines the type of operation and can take the following values: "**=**" (assignment), "**+**" (addition), "*" (multiplication), "**/**" (division) and "**%**" (module). In the following example, variable **x** is declared with value 10 (line 2), then it is added 10, then its value is multiplied by 10, then it is divided by 10 and then **x** becomes equal to **x** mod 3. Then, in line 7, the value of **x** is referenced in the <talk> command content by referencing the variable **x**, using **#x**.

[1]The "sonidos" folder is located in the root directory of the robot application (on the Raspberry PI) or in the root directory of the EVA simulator application.

```
<script>
  <counter var="x" op="=" value="10" />
  <counter var="x" op="+" value="10" />
  <counter var="x" op="*" value="10" />
  <counter var="x" op="/" value="10" />
  <counter var="x" op="%" value="3" />
  <talk>The value of x is #x</talk>
</script>
```

Note *Unlike usnig the $ character, to reference a value contained in a variable, we need to use the # character before the variable name.*

2.3.17 <switch>

The <switch> command creates a sequence of conditional branches from <case> commands. The var attribute of the <switch> command determines with which variable comparisons will be made in the <case> commands and can have the values "$" or any other variable name. Let us see its use in the following example.

```
<script>
  <light state="ON" color="WHITE" />
  <random min="1" max="2" />
  <switch var="$">
    <case op="eq" value="1">
      <light state="ON" color="BLUE" />
      <talk>I choose the blue color</talk>
    </case>
    <case op="eq" value="2">
      <light state="ON" color="RED" />
      <talk>I choose the red color</talk>
    </case>
  </switch>
</script>
```

We saw in the example that the var attribute of the <switch> command selects the "$" character that is referring to the value generated by the <random> command. Therefore, the <case> commands will make the comparisons with the value referenced by "$". More examples will be seen when we cover the <case> command options. As we can see, the code in the example conforms to the grammar of the language. According to Table 2.2, a <switch> element must have one or more <case> commands and may or may not have a <default> command (optional).

2.3.18 <case>

The <case> command specifies a sequence of commands that will be executed if the condition defined in its attributes is true. The <case> command has the op attribute, which defines the type of comparison or logical operator that will be processed, and the value attribute, which contains the value to be compared with the variable defined in the var attribute of <switch>. The value content can be: a constant (a number or a string),

the "**$**" character, or another variable used in the script. It is important to remind that to refer to the value of a previously declared variable, you must use the **#** character before the variable name.

We can define three different comparison types: "**exact**", "**contain**", and "**math**". They will be explained and exemplified in the following subsections.

2.3.18.1 "exact" type comparison

The "**exact**" type is used to compare the content of the value attribute of the <case> command with the value referenced by "**$**". The value specified in value attribute will be compared exactly with the value referenced by "**$**", character by character (*this comparison is not case sensitive*). In the following example, the robot asks if it is raining now. Since the comparison is of "**exact**" type, the first condition handled by the first <case> command will be true, if the answer is exactly "yes". And the second condition, handled by the second <case>, is true if the answer is exactly "no". The next type of comparison that will be presented will make this explanation clearer.

```
<script>
  <light state="ON" color="WHITE" />
  <talk>It is rainning now?</talk>
  <listen />
  <switch var="$">
    <case op="exact" value="yes">
      <light state="ON" color="RED" />
    </case>
    <case op="exact" value="no">
      <light state="ON" color="BLUE" />
    </case>
  </switch>
  <talk>Bye</talk>
</script>
```

2.3.18.2 "contain" type comparison

Comparison of type "**contain**" is used to compare the content of the value attribute of the <case> command with the value referenced by "**$**". But in this case, it is true if the value contained in the value attribute of the <case> command is a substring of the value in "**$**" (*this comparison is not case sensitive*). Suppose the same situation as the previous example, where, the robot asks if it is raining now. With this type of comparison, the following answers would be valid for the first <case> command: *"yes"*, *"I think that yes"*, *"of course yes"*, just contain *"yes"* and the condition will be true. The same happens in the second <case>, when the user's answer contains *"no"* that condition will be true. This is an example of an EvaML script with the "**contain**" type.

```
<script>
  <light state="ON" color="WHITE" />
  <talk>Is it rainning now?</talk>
  <listen />
  <switch var="$">
    <case op="contain" value="yes">
      <light state="ON" color="RED" />
```

```
8     </case>
9     <case op="contain" value="no">
10      <light state="ON" color="BLUE" />
11    </case>
12  </switch>
13  <talk>Bye</talk>
14 </script>
```

Note *"**Exact**" and "**contain**" type comparisons should only be used when the var attribute of the <switch> command is "$", otherwise an error message will be thrown by the parser. These comparisons are based on strings and not numbers.*

2.3.18.3 "math" type comparison

To perform a mathematical comparison in the script, you shall use the symbols of equality, inequality, greater than, less than and etc. In EvaML, due to the conflict generated by the use of the character "<" with the parser, the comparison operators of the "**math**" type are as follows: "**eq**", "**lt**", "**gt**", "**lte**", "**gte**" and "**ne**", which represent, respectively, *equal*, *lower than*, *greater than*, *lower than or equal*, *greater than or equal*, and *not equal*. This approach and some other concepts adopted in the design of EvaML were based on the NCL language [3]. In the following example, we can see the declaration of two variables, **x** and **y**. The <switch> command defines that the variable "**x**" will be compared with the values contained in the value attribute of each <case> command. Line 10 defines that the value contained in variable **x** will be compared with **y**. This was done, as explained before, using the **#** character before the variable name, in this case we use "**#y**".

```
1  <script>
2    <counter var="x" op="=" value="0" />
3    <counter var="y" op="=" value="3" />
4    <counter id="INCREMENT_X" var="x" op="+" value="1" />
5    <switch var="x">
6      <case op="lte" value="3">
7        <talk>The value of x is #x</talk>
8        <goto target="INCREMENT_X" />
9      </case>
10     <case op="gt" value="#y">
11       <talk>The value of x is #x and it is greater than the value of y, that is #y</talk>
12     </case>
13   </switch>
14   <talk>I will terminate</talk>
15 </script>
```

Note *The use of "**math**" comparison assumes that the content of the value attribute of the <case> command is a number.*

2.3.19 <default>

As shown in Table 2.2, the <default> command composes the group of elements (commands) that are children of the <switch> element and its use is not mandatory. If used, it must come as the last element

of a <switch> block, after all <case> commands. The block of commands contained in <default> will be executed if no condition evaluated in the <case> commands is true. If all the possibilities of a comparison in a <switch> block are handled with the <case> commands, there is no need to use the <default> command. The <default> command can and should be used when the <switch> is comparing answers from the <listen> and <userEmotion> commands, as we can have any type of answer. In the specific case of the <userEmotion> command, we may receive an "*undefined*" answer, as a result of a failure in the process of capturing a facial expression. In this case, the <default> command can be seen as an "*exception handler*" which can ensure the script works if any unexpected answer is provided.

In the following example, the robot generates a random number between 1 and 5, inclusive. The <case> commands of the <switch> block provide answers if the drawn value is 1 (first <case> command), if the number is 2 (second <case> command) and case is neither 1 nor 2 (<default> command).

```
<script>
  <random min="1" max="5" />
  <switch var="$">
    <case op="eq" value="1">
      <talk>The number drawn was 1</talk>
    </case>
    <case op="eq" value="2">
      <talk>The number drawn was 2</talk>
    </case>
    <default>
      <talk>The number drawn was neither 1 nor 2</talk>
    </default>
  </switch>
</script>
```

This approach can be used in other situations, as mentioned above. We can use the <default> command to handle unexpected answers, for example, we could have a <switch> block that handles the answers of a <listen> command, having a <case> command to handle a "yes", a <case> command to handle a "no" and finally a <default> command to handle "*I don't know*" or "*maybe*" answers.

2.3.20 <loop>

Loops are very useful elements in any programming language. The construction of loops, in the EvaML language, can be done using the <switch> and <case> elements, however, in addition to this construction being a little tedious, it can leave the code a little confusing. In order to facilitate the construction of these repetition loops by the programmer, making the code clearer, EvaML has the <loop> command. The <loop> command executes "n" times the code block defined inside it and it has the following three attributes: id, var and times. The id and var attributes are not required attributes while the times attribute is required. The id attribute follows the same definition presented in the other language commands, it serves to define a label that will be used as the target of a <goto> command. The var attribute allows the programmer to define the iteration variable that will be used by the <loop> command. This allows the developer to reference it within the loop structure. If the programmer does not define such a variable, the parser will automatically create one. As this variable is created automatically and its name defined randomly, its use inside the loop, by the programmer, becomes impossible. The times attribute defines the number of repetitions of the loop. Allowed values for the times attribute are

positive integer values or other variables defined by the programmer beforehand. Next, in the code snippet, you can see an example of a <loop> that will execute 3 times the code block in its entirety and that uses the variable "L1" as iteration variable. Notice how the iteration variable "L1" is used inside the <talk> command.

```
<script>
  <loop var="L1" times="3">
  <talk>Counting #L1</talk>
  <evaEmotion emotion="HAPPY" />
  <led animation="HAPPY" />
</script>
```

The parser, when processing a <loop> command, as in the previous example, transforms the structure of the loop using the basic EvaML commands. See, in the following listing, the code generated by the parser for the code snippet of the previous example.

```
<script>
  <counter var="L1" op="=" value="1" />
  <switch id="LOOP_ID1_L1" var="L1">
    <case op="lte" value="3">
      <talk>Counting #L1</talk>
      <evaEmotion emotion="HAPPY" />
      <led animation="HAPPY" />
      <counter var="L1" op="+" value="1" />
      <goto target="LOOP_ID1_L1" />
    </case>
    <default />
  </switch>
</script>
```

The parser automatically generates the commands for creating the variables (<counter>), the <switch>, the <case>, the <goto> command and the label needed to create the loop. The <loop> command facilitates the writing of EvaML code, contributing to the clarity and better understanding of the created script.

2.4 EvaML Parser

Parsing an EvaML document takes place in four steps, which are performed by four modules and they can be seen in Figure 2.4. These steps will be explained in detail as follows.

2.4.1 The Parsing Process

The first step consists in the process of expanding the macros used in the <script> section. In this step, the contained commands within the macros are expanded throughout the script. This expansion is done in the places in the script where the macros are referenced, through the <useMacro> command.

The script that robot EVA runs is represented as a graph with nodes and links coded in JSON. Therefore, the second step performs the process of generating the keys of the graph nodes. They will represent, together with the links that connect them, the flow of script execution. Each robot language command is equivalent to

a node in the graph that represents the script. These nodes are connected through links that use these keys to reference them.

The third step is responsible for analyzing the script execution flow. It is, at this stage, that the parser identifies the script execution flow, with its possible deviations, jumps and repetitions, generating the links that form the script execution graph. As an output of this step, there is the XML file that is intended for EvaSIM.

In the fourth step, the parser associates the commands and links in the EvaML language to the JSON templates that represent the robot's commands. The parser identifies the XML element with its attributes, processing each command in a specific way. As a result of this last step, a JSON file is generated that can be sent to the robot's script database and then executed. The EvaML parser program was developed in Python.

Table 2.4: EVA Parsing Flags

Flag	Definition
-c	"Compiles" the EvaML script, outputting an XML file to run in the Eva simulator
-s	Saves/Inserts the JSON file generated by the parser into the robot's database
-r	Makes the EVA robot run the script immediately

2.4.2 The Parser Flags

The parser can be run using three flags that are presented in Table 2.4. Using just the "**-c**" flag, the parser outputs two files, one in EvaSIM format and the other in JSON format, which is the code for the robot. Using the "**-s**" flag, the generated JSON file, is inserted directly into EVA's database. And finally, using the "**-r**" flag, it makes the robot immediately start the script. Figure 2.4 shows an example of the parsing command line with its flags.

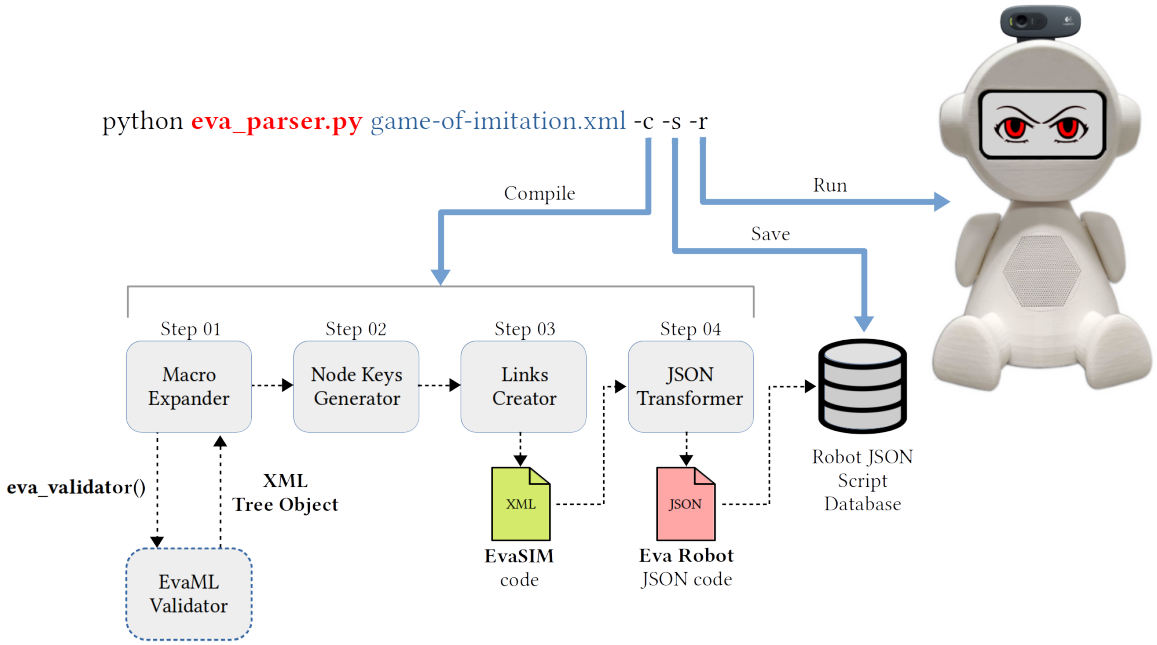

Figure 2.4: Parsing Process and Command Line Example

Note *The EVA script execution control is done through an **http request** to the EVA web service, passing the **id** of the script to be executed. The configuration of this command, as well as the service's **IP** address and **port**,*

*are found inside the **eva_parser.py** file.*

2.4.3 Validating an EvaML Document

The EvaML language was specified using an XML-Schema file. The purpose of a "schema" is to define and describe a class of XML documents using these constructs to constrain and document the meaning, usage and relationships of its constituent parts: d*ata types*, *elements and their contents*, *attributes and their values*, *entities and its contents and notations*. The EvaML script validation process takes place in parallel and is initiated by the *Macro Expander* module that uses the *EvaML Validator* module, as shown in Figure 2.4

2.4.4 Parsing a Script

Now we will show a step by step process of *parsing* an EvaML script and also present the files that are generated during the process. Any text editor can be used to write an EvaML script, but it is recommended that one that supports writing XML code and that validates the script against XML Schema on-the-fly, indicating possible inconsistencies in the editor window itself. There are several code editors available for free. We have tested and recommended two IDEs. The first is Eclipse. This IDE, with the reference to EvaML's XML Schema embedded in the script, provides auto-completion and on-the-fly validation capabilities. Another IDE that was tested was Visual Studio Code. To properly support XML and XML Schema validation on the fly, you need to install the XML Language Support by Red Hat extension. Installation is very simple. You can browse and install extensions from within VS Code. Bring up the Extensions view by clicking on the Extensions icon in the Activity Bar on the side of VS Code or the View: Extensions command (*Ctrl+Shift+X*). This will show you a list of the most popular VS Code extensions on the VS Code Marketplace. In the search bar, type XML Language Support by Red Hat. To install an extension, select the Install button. Once the installation is complete, the Install button will change to the Manage gear button.

```
<?xml version="1.0" encoding="UTF-8"?>
<evaml name="script01" >
  <settings>
    <voice tone="en-US_AllisonV3Voice" />
    <lightEffects mode="ON" />
    <audioEffects mode="ON" />
  </settings>
  <script>
    <light state="ON" color="WHITE" />
    <talk>Hi, I'm the robot Eva and I'm a socially assistive robot</talk>
    <talk>What is your name?</talk>
    <listen />
    <light state="ON" color="GREEN" />
    <talk>Hi $, how are you? Let's start the script</talk>
    <counter var="x" op="=" value="0" />
    <counter var="x" op="+" value="1" />
    <talk>The value of x is #x</talk>
    <talk>Hey $, I will terminate. Bye</talk>
    <light state="OFF" />
  </script>
</evaml>
```

The filename of the script that will be used in this example is *my_script_file.xml*. For the execution of the commands that will be presented here, we assume that both the scripts files, *eva_parser.py* and *eva_simulator.py* are in the same folder.

```
python3 eva_parser.py my_script_file.xml -c
```

Note *To generate only code for EvaSIM use just run the parser using the "**-c**" flag.*

If all goes well, you will be able to see an output on the terminal, as Figure 2.5 shows. You can see that all 4 steps of the parsing process were successful.

```
marcelo@note-mint:~/Dropbox/Estudo XML$ python3 eva_parser.py my_script_file.xml -c
Step 01 - Processing Macros...
Step 02 - Generating Elements keys...
step 03 - Creating the Elements <link>...
step 04 - Mapping XML nodes and links to a JSON file...
marcelo@note-mint:~/Dropbox/Estudo XML$
```

Figure 2.5: Parsing Terminal Output

After the execution of *eva_parser.py* we have as output a code for the EVA simulator and a code to run on the robot (in JSON format). During the *parsing* process some intermediate files are created, as seen in Figure 2.6. It is important to note that the name of the output files of EvaML and JSON type is based on the name attribute of the <evaml> root element of the *my_script_file.xml* script.

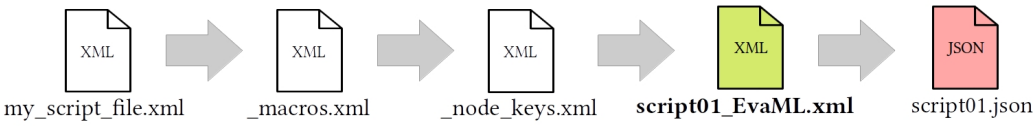

Figure 2.6: Parsing Process - Intermediate and Output Files

In the next chapter, we will present the EVA Robot Simulator Software (EvaSIM). In addition to talking about the simulator and its interface, in Section 3.5, the file generated in the parsing process (*script01_EvaML.xml*) will be imported and executed in the simulator.

Chapter 3 EvaSIM

3.1 The User Interface

In order to provide a test environment for EvaML scripts, we developed a software that simulates the EVA robot, which is called EvaSIM. The simulator was developed in Python and uses some extra modules. Appendix A shows how to install and configure Python and the extra modules needed to run the EVA Parser and the EVA simulator.

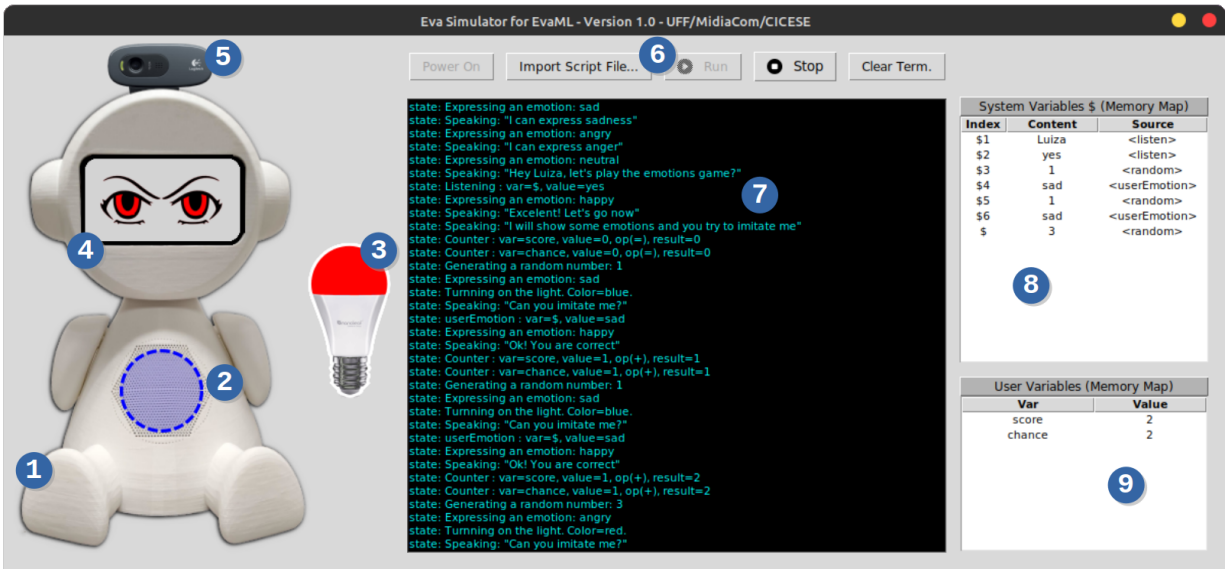

Figure 3.1: EvaSIM - User Interface

As we can see in Figure 3.1, the simulator for EvaML script language, EvaSIM, tries to imitate, in a simplified way, the robot system. All the elements in its user interface will be described as follows: the EVA robot figure (**1**), the representation of the Matrix Voice board RGB LEDs (**2**), the smart bulb (**3**), the Amoled 5.5 touch display (**4**), the webcam (**5**), buttons to import, turn on/off, and control the execution of EvaML script (**6**), a terminal emulator where some important information is presented, such as: the actions being performed by the robot, details of operations with variables, colors and states of the smart bulb, the text that the robot is speaking and some alert messages and possible script error messages (**7**), the itens (**8**) and (**9**) indicate the memory map tables. These two tables are intended to dynamically show the system and user variable values during the execution of the script. The upper table shows the system variable values that stores the responses obtained from user interaction processes, such as voice capture and facial expression recognition. This variable set also holds the values generated by the VPL random number generation command. Those system variables are indexed using the "$" character. Since the robot memory is full of values from different sources, the upper table has, in addition to the index and content columns, an extra column that shows the source of this variable value. The second table presents the variables created by the script developer, with their names and their respective values.

3.2 Listening Simulation

EVA can communicate with the user through voice interaction, capturing the audio of the answers and transforming it into text (using Google cloud API). In order to facilitate this process, within the simulator, this type of multimodal interaction was represented by a text box, where the user can answer using written text instead of speech. We can see, in Figure 3.2, the simulation of the listening process of the robot.

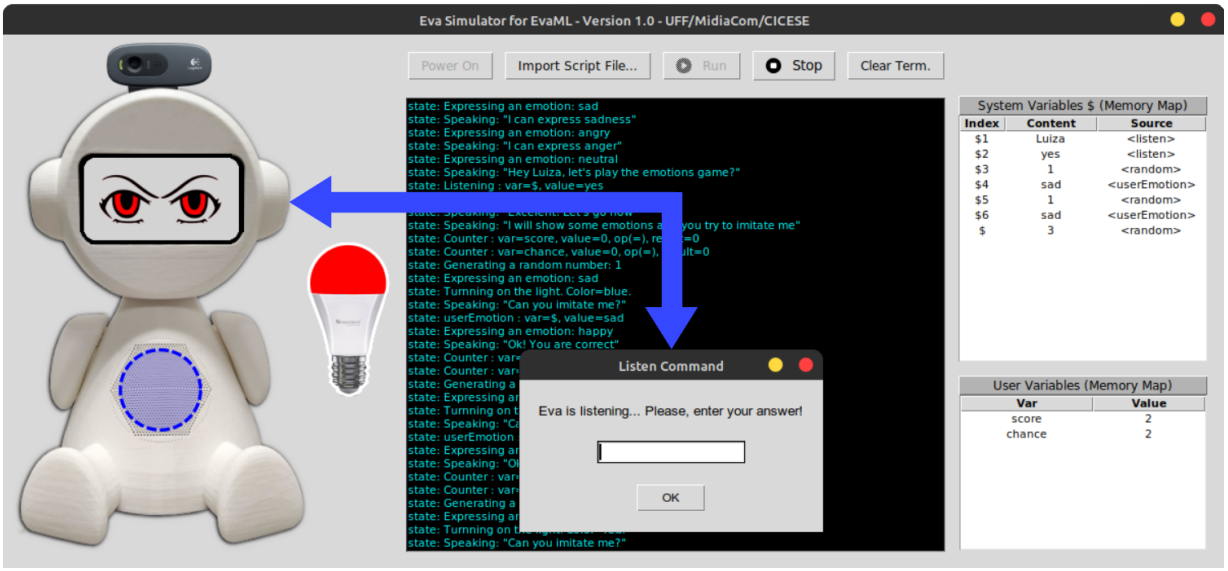

Figure 3.2: EvaSIM - Listening Command Simulation

3.3 Facial Expression Recognition Simulation

In Figure 3.3, EvaSIM simulates the facial recognition process. This function was implemented in the EVA simulator using a window with facial expressions represented by Emojis. In the simulator, the answers that can be provided through the window representing the user's facial expressions are: "NEUTRAL", "HAPPY", "ANGRY", "SAD", "SURPRISED". The execution of the <userEmotion> command, which is responsible for capturing the user's expression through the webcam, opens a window with a set of facial expressions. The user, through the mouse, can indicate his/her facial expression. This answer is processed by the simulator in the same way as the physical robot. We have a short video[1] showing the simulation of the *Imitation Game* [7] in EvaSIM.

Note *The EVA robot simulator works with only 5 types of expressions, while the physical robot's facial recognition module can return up to 7 types of facial expressions.*

3.4 Head Movement Simulation

EvaSIM was designed to be a lightweight tool that required low computational power to run. Therefore, no sophisticated animations were implemented in this 2D version of the simulator. The physical robot, as presented in Section 2.3.8, can move its head. This movement used in conjunction with other elements can increase the robot's expressiveness. When executing a script for the robot and finding a <motion> element, EvaSIM uses

[1] https://youtu.be/OfGelKZIA9c

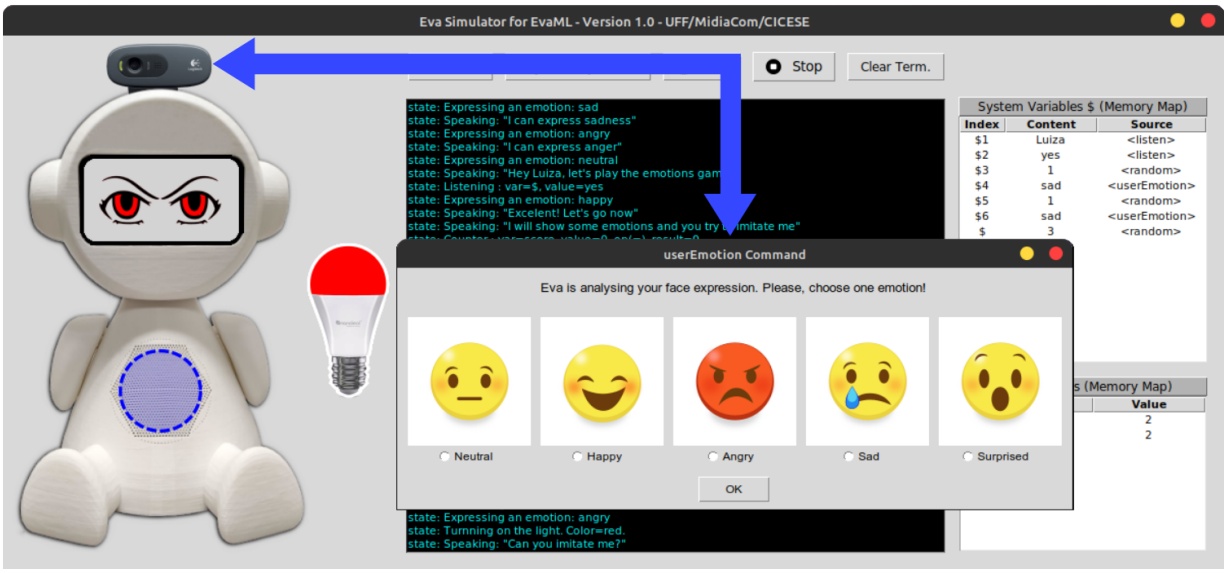

Figure 3.3: EvaSIM - userEmotion Command Simulation

the terminal to indicate that a movement is being executed, also indicating the type of movement performed. Figure 3.4 shows an example of the message indicating the execution of the <motion> element and the value of the type attribute.

```
state: Matrix Leds. Animation=STOP
state: Moving the head! Movement type: CENTER
state: Pausing. Duration=1000 ms
state: Matrix Leds. Animation=SURPRISE
state: Moving the head! Movement type: 2RIGHT
state: End of script.
```

Figure 3.4: Head Movement Messages in Terminal

3.5 Importing and Executing an EvaML Script

Let us continue the process that was started in Section 2.4.4, when we ran the parsing process on the *my_script_file.xml* file and the *script01_EvaML.xml* file was generated. The goal now is to run EvaML code in EvaSIM. Figure 3.5 shows the image of the simulator interface buttons.

To start using EvaSIM you need to click on button "Power On" (**1**). EvaSIM will speak a greeting text and wait for a script to be loaded into its memory. A script can be loaded by pressing button "Import Script File..." (**2**), which will open the file open dialog. After importing the file, the script can be run by clicking button "Run" (**3**). Button "Stop" (**4**), if clicked, stops the script execution and button "Clear Term." (**5**) clears the EvaSIM terminal emulator.

Figure 3.5: Operating the EvaSIM

Figure 3.6 shows the terminal emulator after executing script "*script01_EvaML.xml*". It presents the

emulation of a terminal where some important information is presented, such as: the actions being performed by the robot, details of operations with variables, colors and states of the smart bulb, the text that the robot is speaking and some alert messages and possible script error messages. We can see the voice selection, the smart bulb state and color being set, the texts being spoken, the capture of the username via the <listen> command, and the manipulation of the **x** variable.

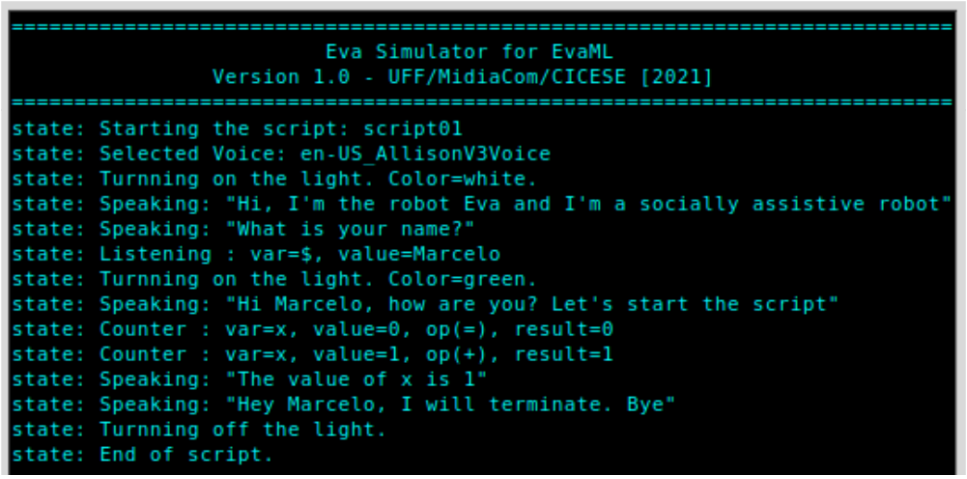

Figure 3.6: EvaSIM - Terminal Emulator

Appendix A EvaSIM - Installing and Configuring

A.1 Installing Python and Dependencies

Both the EvaML language parser and the EvaSIM simulator were developed using the Python language. The parser uses the standard Python library along with *xmlschema* module, while the simulator uses some extra libraries that must be installed. For the EvaSIM, the entire graphical user interface was built using the *tkinter* package which is a thin object-oriented layer on top of *Tcl/Tk*. The Text-To-Speech process uses the *IBM-Watson* library.

A.1.1 Installation on Windows 10

First, go to https://www.python.org/downloads/ and download the latest version of Python, in this case, we used version 3.10.1(64-bit). Start the installation process and select the options indicated in the images as you can see in Figure A.1. During the installation, make sure to choose the option "customize installation" and also choose the option "Add python 3.10 to PATH".

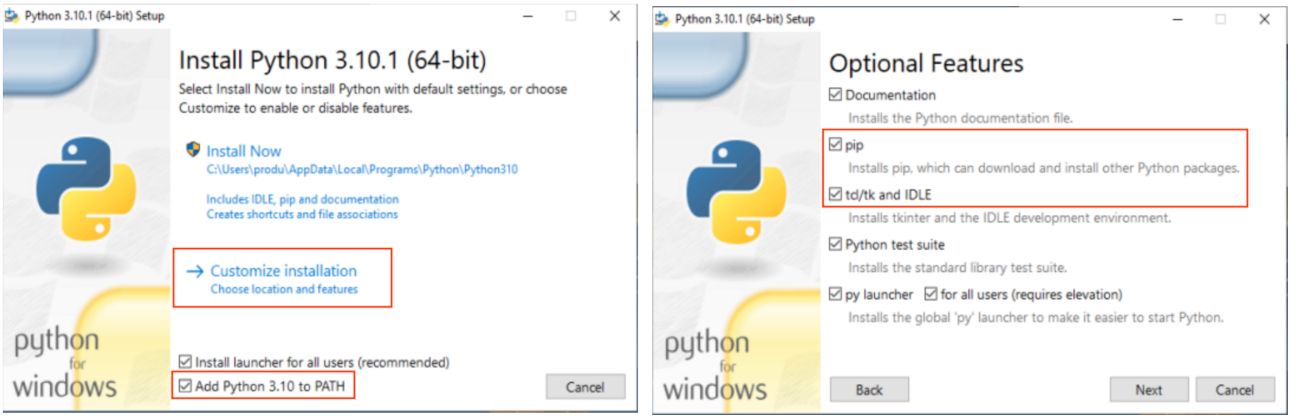

Figure A.1: Windows 10 - Python installation

Choosing the customization option allows you to select the automatic installation of *pip* (the standard package manager for Python) and also the *tk* library. Figure A.1 shows the options that should be selected.

Then we continue with the installation of dependencies. First let us install the *xmlschema* module. It is the component responsible for the EvaML script validation process. This validation is done based on the XML Schema file defined for EvaML. An EvaML script must meet the rules defined in the XML Schema file. XML Schema describes the structure of EvaML, and it is a basic consistency control mechanism. The process of validating the EvaML document against the XML Schema file is performed in the first step of the parser.

```
pip install xmlschema
```

Now let us install the *IBM-Watson* library. For that, we will use Python's package manager, which makes everything really easy. The following command must be executed in the Windows terminal.

```
pip install ibm-watson
```

Next we will install the *playsound* library. The version installed must be the version specified in the *pip* command, as the latest version had issues on Windows 10.

```
pip install sounddevice
pip install soundfile
pip install numpy
```

If all steps worked correctly, we already have everything needed for the EvaML *parsing* process and for running the EvaSIM simulator.

A.1.2 Installation on Fedora 35 (RPM)

By default, the Fedora 35 distribution already comes with Python 3 installed, so we will proceed with installing the extra libraries. First, let us install *pip* (Python's package manager).

```
sudo dnf install python3-pip
```

We continue with the installation of the *tkinter* graphic library.

```
sudo dnf install python3-tkinter
```

Following are the commands for installing the *xmlschema*, *IBM-Watson* and *playsound* libraries.

```
pip install xmlschema
pip install ibm-watson
pip install playsound
```

We already have everything we need to run the *parser* and the simulator.

A.1.3 Installation on Ubuntu 20.04.3 (Deb)

As with Fedora, Python 3 is already installed by default on Ubuntu 20.04.3. We then proceed with the installation of the package manager for Python (*pip*).

```
sudo apt install python3-pip
```

Now we install the *tkinter* graphics library.

```
sudo apt install python3-tk
```

Following are the commands for installing the *xmlschema*, *IBM-Watson* and *playsound* libraries.

```
pip install xmlschema
pip install ibm-watson
pip install playsound
```

Bibliography

[1] Dagoberto Cruz-Sandoval and Jesus Favela. "A Conversational Robot to Conduct Therapeutic Interventions for Dementia". In: *IEEE Pervasive Computing* 18.2 (2019), pp. 10–19. ISSN: 15582590. DOI: 10.1109/MPRV.2019.2907020.

[2] D. Feil-Seifer and M.J. Mataric. "Defining socially assistive robotics". In: *9th International Conference on Rehabilitation Robotics, 2005. ICORR 2005.* 2005, pp. 465–468. DOI: 10.1109/ICORR.2005.1501143.

[3] Luiz Gomes-Soares and Simone Junqueira-Barbosa. *Programando em NCL 3.0*. 2012.

[4] Julie A Kientz et al. "Interactive technologies for autism". In: *Synthesis lectures on assistive, rehabilitative, and health-preserving technologies* 2.2 (2013), pp. 1–177.

[5] Marjan Mernik, Jan Heering, and Anthony M Sloane. "When and how to develop domain-specific languages". In: *ACM computing surveys (CSUR)* 37.4 (2005), pp. 316–344.

[6] Marek Novák. "Easy implementation of domain specific language using xml". In: *Proceedings of the 10th Scientific Conference of Young Researchers (SCYR 2010), Koice, Slovakia*. Vol. 19. 2010.

[7] M. Rocha et al. "Towards Enhancing the Multimodal Interaction of a Social Robot to Assist Children with Autism in Emotion Regulation". In: *Proceedings of the 15th EAI International Conference on Pervasive Computing Technologies for Healthcare*. 2021.

[8] Settapon Santatiwongchai et al. "BLISS: Using Robot in Learning Intervention to Promote Social Skills for Autism Therapy". In: *Proceedings of the International Convention on Rehabilitation Engineering & Assistive Technology*. i-CREATe 2016. Midview City, SGP: Singapore Therapeutic, Assistive & Rehabilitative Technologies (START) Centre, 2016.

[9] Laura Santos et al. "Design of a Robotic Coach for Motor, Social and Cognitive Skills Training Toward Applications With ASD Children". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 29 (2021), pp. 1223–1232.

[10] Takanori Shibata. "An overview of human interactive robots for psychological enrichment". In: *Proceedings of the IEEE* 92.11 (2004), pp. 1749–1758. ISSN: 00189219. DOI: 10.1109/JPROC.2004.835383.

[11] Takanori Shibata. "Therapeutic seal robot as biofeedback medical device: Qualitative and quantitative evaluations of robot therapy in dementia care". In: *Proceedings of the IEEE* 100.8 (2012), pp. 2527–2538. ISSN: 00189219. DOI: 10.1109/JPROC.2012.2200559.