INSTALLATION AND USE MANUAL

AUTOMATIC TEST GENERATION TOOL

Miguel Angel Gallardo Ruiz

Departamento de Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
Málaga, Agosto de 2022

Table of Contents

1. Installation Manual

This part shows a manual so that the user can replicate the test generation part of this project. In this section we will focus specifically on the installation.

The first thing the user should do is download Eclipse and install it, this part is trivial, the only thing the user should do is go to the browser and look for the executable that best suits their computer, in this case for us it is the latest version as seen in Figure 1.

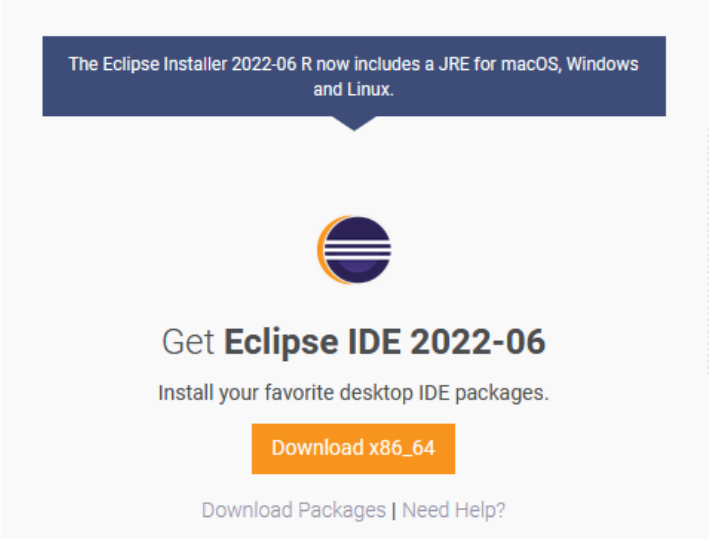

Figura 1: Eclipse version

Once it is installed, the user must install Xtext, for this he must go to the following link:

`https://www.eclipse.org/Xtext/download.html`

Once you are on this page, go as shown in Figure 2 to the 'Releases' section and with the right click it should give the option to copy link.

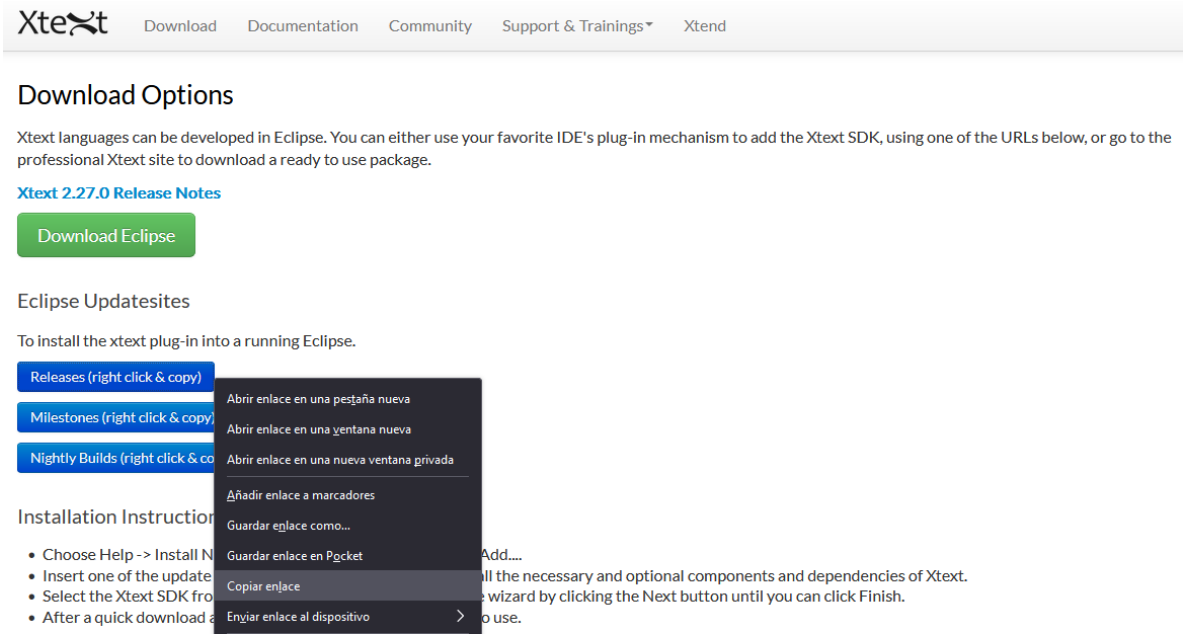

Figura 2: Xtext Release

With this address copied, you must go to the Eclipse environment and in the 'Help' section open 'Install New Software', as seen in Figure 3

Here the tab of Figure 4 is opened which allows the installation of packages, here the user must select the 'Add...' button which opens the tab of Figure 5. In this you must write the name that you want to give to the package to install, in this case Xtext, and paste the link that has previously been copied from Xtext. Once this is done, the user clicks the 'Add' button and have to wait for the options to load.

Once loaded, the user must select all of them as shown in Figure 6 and click on the 'Next' button. When all the options have been loaded, the Eclipse program must be restarted and you can start using Xtext.

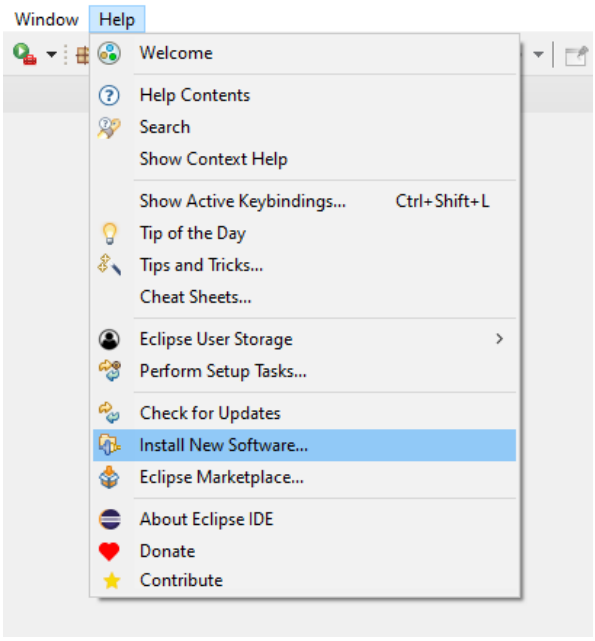

Figura 3: Opening the installation tool

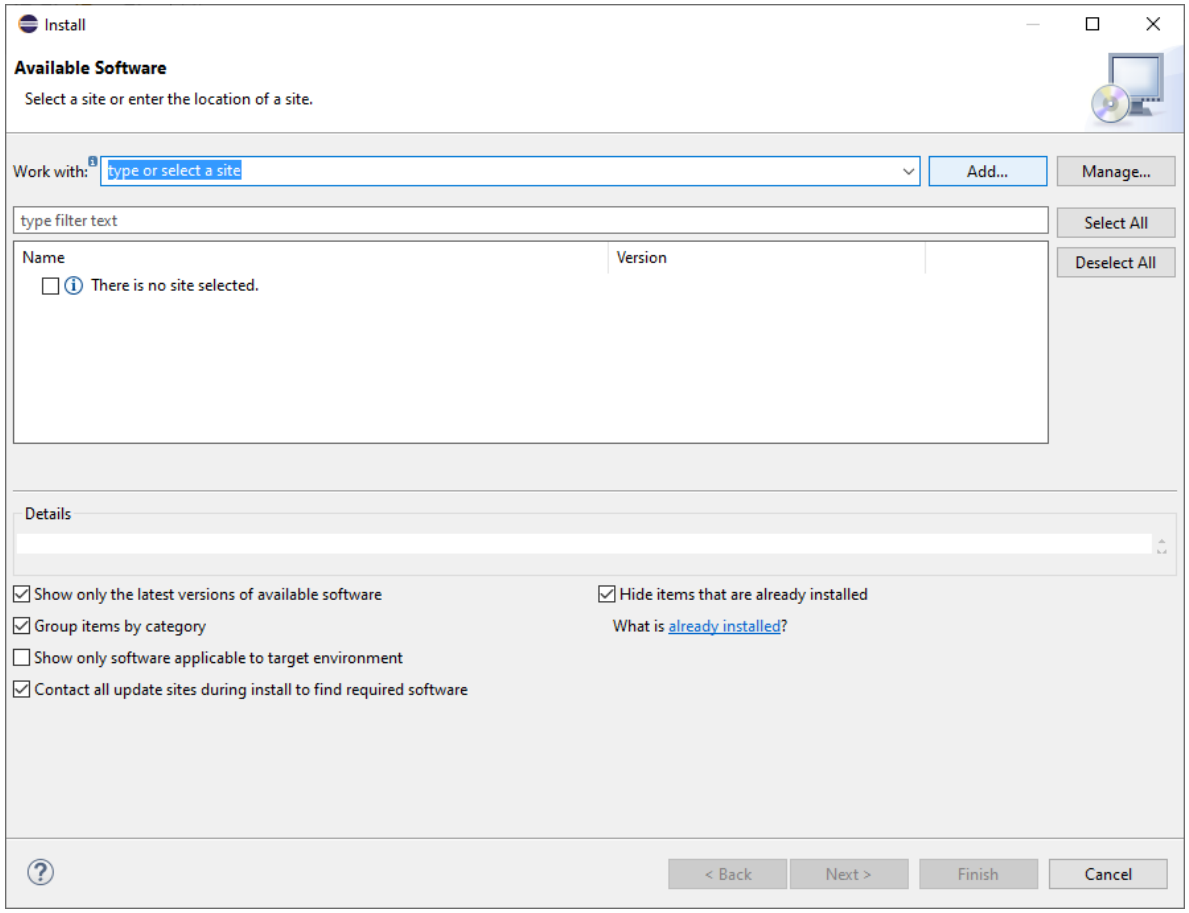

Figura 4: Installation tool

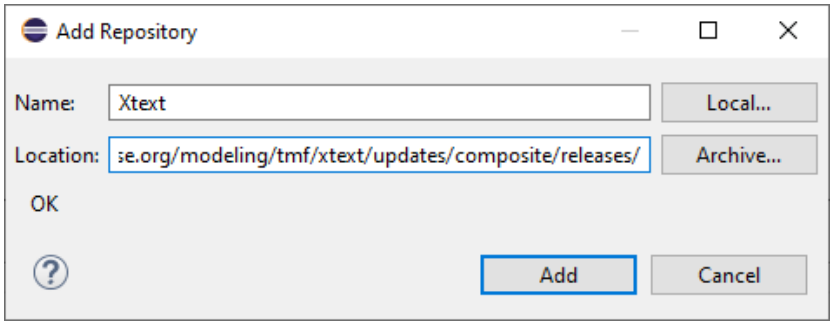

Figura 5: Xtext repository import

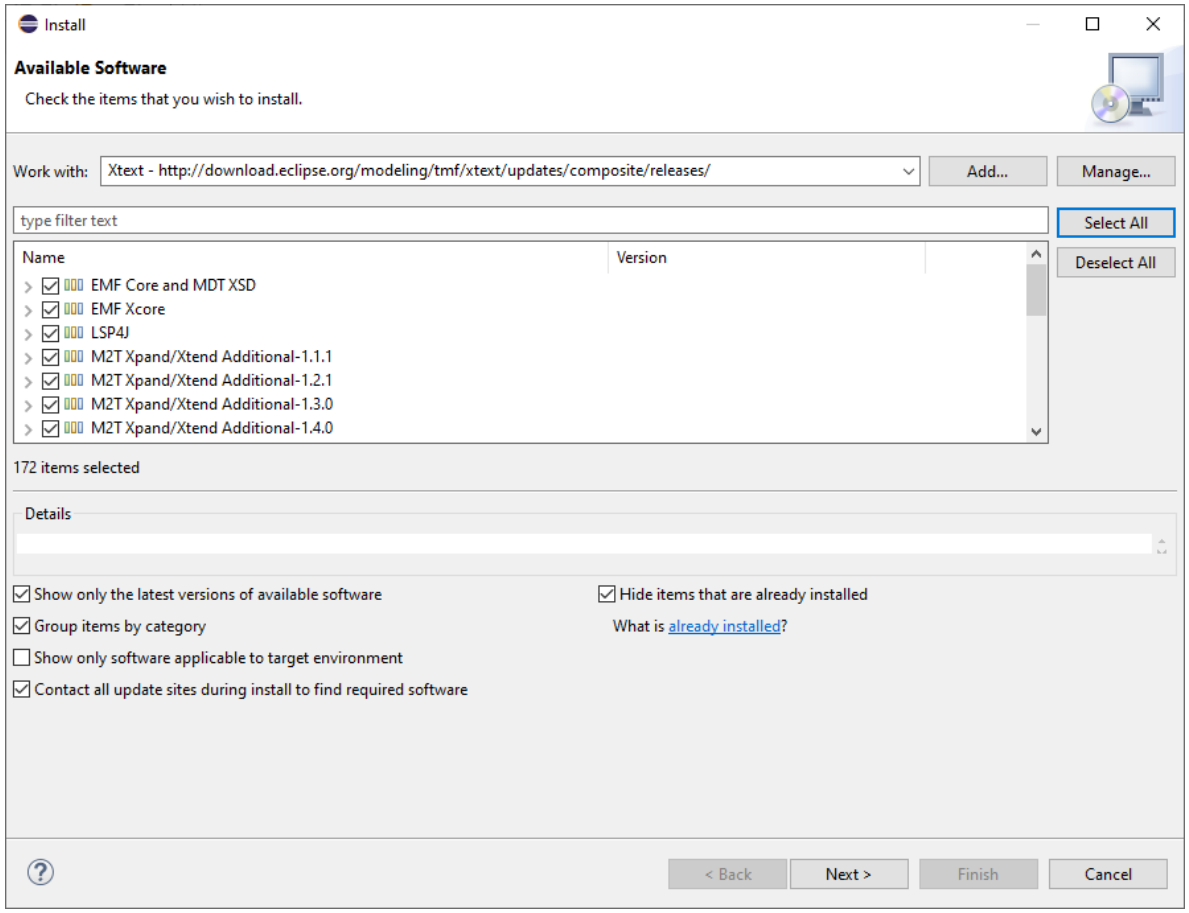

Figura 6: Installation of the Xtext repository

2. User Manual

This section shows a manual so that the user can replicate the test generation part of this project.

The first thing the user must do is open Eclipse, once opened they must create a new project, which must choose an Xtext project, as shown in Figure 7.

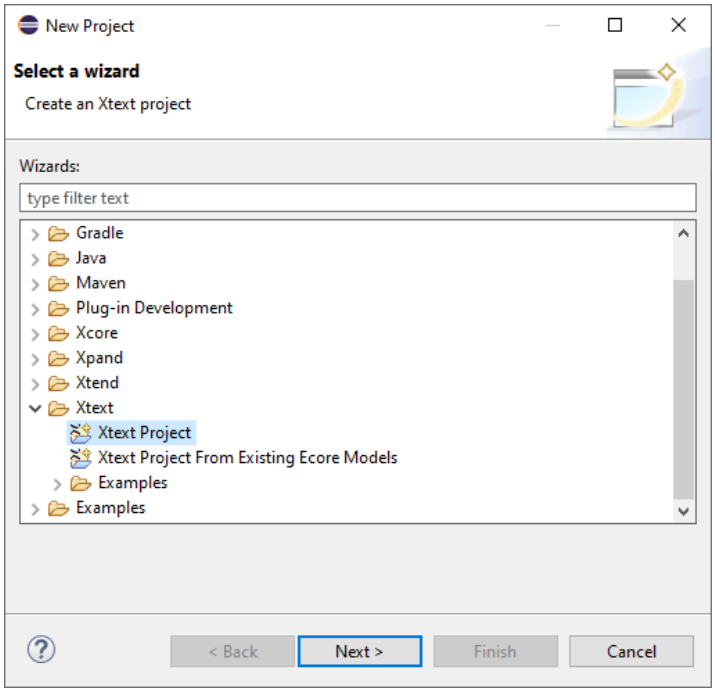

Figura 7: Selection of the new project

Once it is given to continue (Next), a new tab is opened like the one seen in Figure 8, in this the name of the project is defined, as well as the name of the language and the extension which will have this

These first two steps have to be carried out twice, since a general project and a specific one for the operations to be carried out are required.

Now that we have the two projects created, the next step is to add the operations project as dependencies of the general. The way to do this is shown in Figure 9, where the 'MANIFEST.MF' file is opened and, in the dependencies section, click on the Add button. A pop-up screen then opens where you must search for the name of the project to add as a dependency and then add it (Add).

The user can now write the code for both the general project and the operations project in Xtext.

The only thing to keep in mind is that the general project must add to the operations project its grammar and, in the file with the '.mwe2' extension, as shown in Figure 10, add the location of the model generated for the operations project.

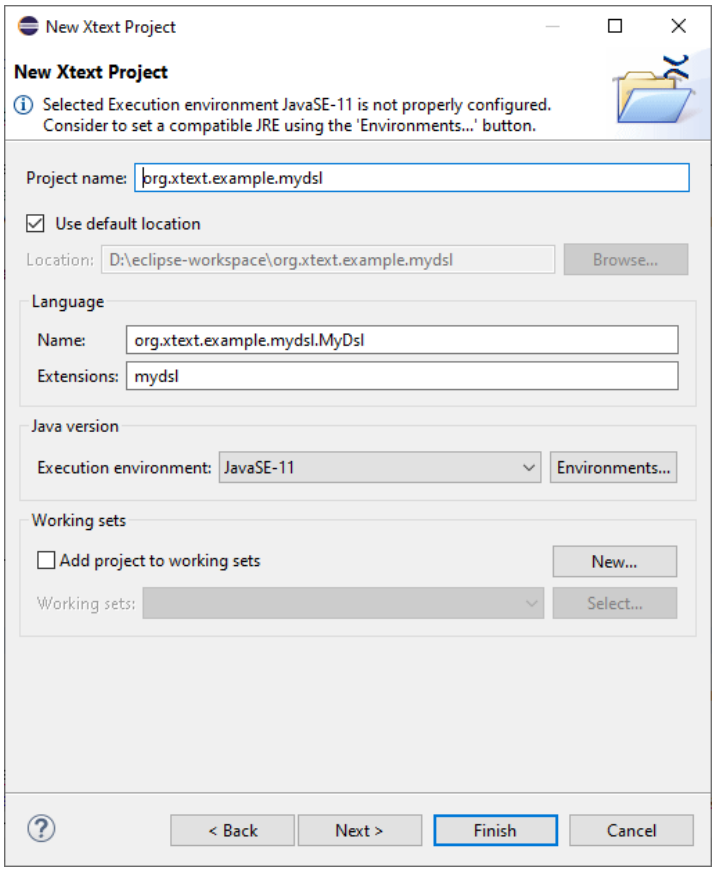

Figura 8: Defining the characteristics of the project

Once you have this, you can proceed to compile the projects. For this, the user must locate the main '.xtext' file of each project and compile them, as seen in Figure 11. The order will be; first the operations one and then the general one.

Now the user can define the code of the '.xtend' file with the text that he wants to generate as output.

Having finished this, you can proceed to launch the application as shown in Figure 12, this will open a new Eclipse environment.

In this new environment that has been opened, a new empty project is created. Within this, a folder must be created which will be called 'src', and within this folder a new file which must be given the extension of the general project with which we are launching this application and which has been defined at the beginning, as seen in Figure 13.

The user can write to this new file, which will use the specifications defined in the project. Once the user writes the code and saves it, a new 'src-gen' folder is automatically generated in which a file with a text extension is created with all the language that has been defined in the '.xtend file ' making use of what was defined by the user in the file with the specifications.

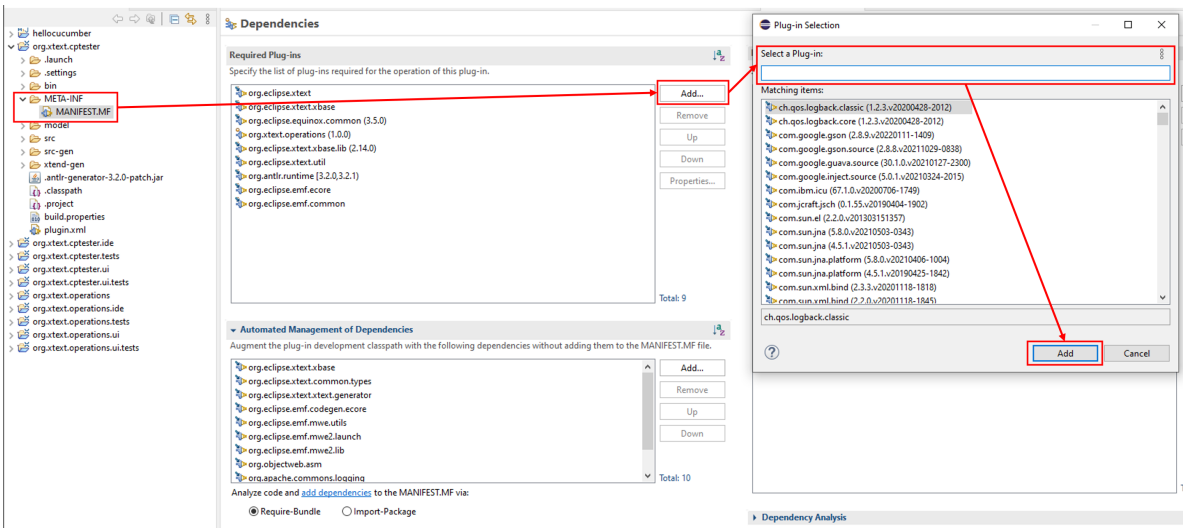

Figura 9: Addition of dependencies

```
bean = org.eclipse.emf.mwe.utils.StandaloneSetup {
    scanClassPath = true
    uriMap = {
        from = "platform:/plugin/org.eclipse.emf.ecore/model/Ecore.ecore"
        to = "platform:/resource/org.eclipse.emf.ecore/model/Ecore.ecore"
    }
    uriMap = {
        from = "platform:/plugin/org.eclipse.emf.ecore/model/Ecore.genmodel"
        to = "platform:/resource/org.eclipse.emf.ecore/model/Ecore.genmodel"
    }
    registerGenModelFile = "platform:/resource/org.xtext.operations/model/generated/Operations.genmodel"
```

Figura 10: Code of the '.mwe2' file of the general project with the locations of the model of the operations project

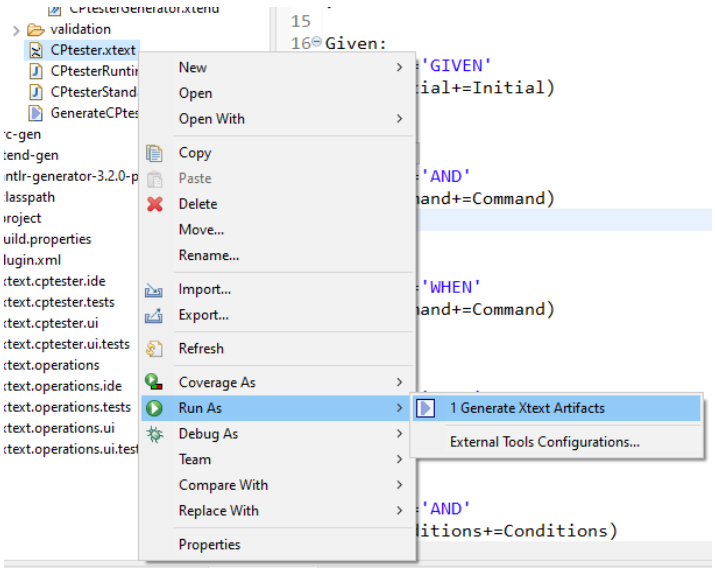

Figura 11: Compilation of '.xtext' code

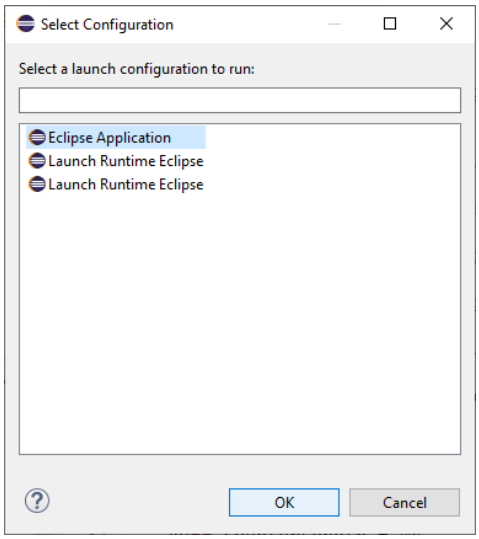

Figura 12: Project execution

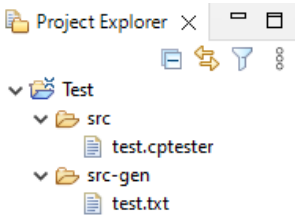

Figura 13: Generation of folders and files in the project

3. Example

In this example the user creates a series of operations to perform tests with the arm.

The operation to be carried out by the user in this case consists of the correct initialization of the arm with a time of 1000ms, the turning off and subsequent turning on of the buzzer, a subsequent movement of all servos to a position of 45° with a time of 2000ms, the rotation of servo number 3 to a position of 30° with a time of 1000ms, the rotation of servo 1 to a position of 90° with a time of 2000ms and the verification that it is in this position with a margin of error of 2° and that the operation has not taken more than 2000ms.

All this user generated code can be seen in figure 14

```
Scenario "Testings"
{
    GIVEN posInicial(1000)
    AND buzzerOff()
    AND buzzerOn(1)
    AND rotateAllServos(45, 45, 45, 45, 45, 45, 2000)
    AND rotateServo(3, 30, 1000)
    WHEN rotateServo(1, 90, 2000)
    THEN result(2000)
    AND NotLaterThan(2000) Query isAtSingle(1, 90, 2)
}
```

Figura 14: User code

Next, we are going to see the pivot code generated through the code written by the user.

As can be seen in Figure 15, on the left side there is the code in CPtester that the user specifies. On the right you can see the result of the pivot language that is obtained by Xtend.

For this particular case, we see how the user first defines the name of the scenario through *"Scenario "Testings" "*, this is translated by creating *"StateMachine: Testings "*, this will be the name of the state diagram it will have in Altova UModel once it is imported.

ANext, we have first *"GIVEN StartPos(1000)"*, this corresponds in the pivot language to *"Activity: Arm.StartPos(1000)"* as seen in Figure 14. With this, what the user wants is to perform the first possible action with the arm, in this case take it to the initial position and specifically, as defined, with a time of 1000ms. It is also observed how, apart from the state with the operation, different transitions are generated from the current state to the next one if everything goes well, as well as transitions to states of type "Warning" if there is an error.

The next instance is *"AND buzzerOff()"*, which turns off the buzzer, the pivot language operation corresponding to this is *"Activity: Arm.Buzzer.buzzerOf()"*. Like the previous operation, the corresponding state and all the transitions to the different states are generated.

Similarly, we have the following instance *"AND buzzerOn(1)"*, which will turn on the led with a period of 1000ms. The operation that is generated is *"Activity: Arm.Buzzer.buzzerOn(1)"* within the corresponding state and with the consequent possible transitions also generated automatically.

In the following Figure 16, we have the second part of the code that is generated.

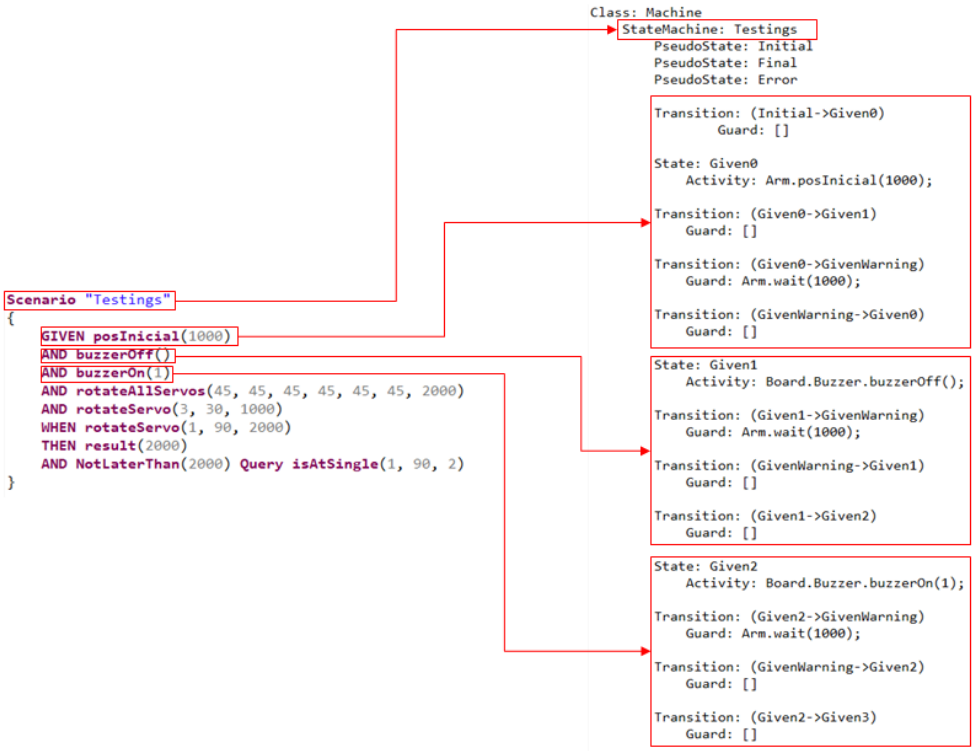

Figura 15: First part of the generated code

Following the operations discussed above, we now find *"AND rotateAllServos(45, 45, 45, 45, 45, 2000)"*, which translates in pivot language as *"Activity: Arm.rotateAllServos(45, 45, 45, 45, 45, 2000)"* in the corresponding state and with the transitions pertinent to it.

Finally, in the AND section, we find *"AND rotateServo(3, 30, 2000)"* which corresponds to the operation with its status and transitions, except for the last transition that is generated automatically when next state corresponding to the main operation of this example, which corresponds to the WHEN. In this case the transition will take the position of the last given operation to use it as a guard when moving to the next state. Also note the creation of the WARNING state that has been referenced so far in previous transitions.

Now we find *"WHEN rotateServo(1, 90, 2000)"* which corresponds to the main operation to perform, this is translated in pivot language as *"Activity: Arm.BaseServo.ServoOperations .rotateServo(1, 90, 2000)"*. Next, we have the *"THEN result(2000)"* and *"AND NotLaterThan(2000) Query isAtSingle(1, 90, 2)"* operations, which correspond to the transition to ERROR state if it is not done within the indicated time as well as the transition to the FINAL state if the condition indicated by the user regarding the position of the angle is fulfilled.

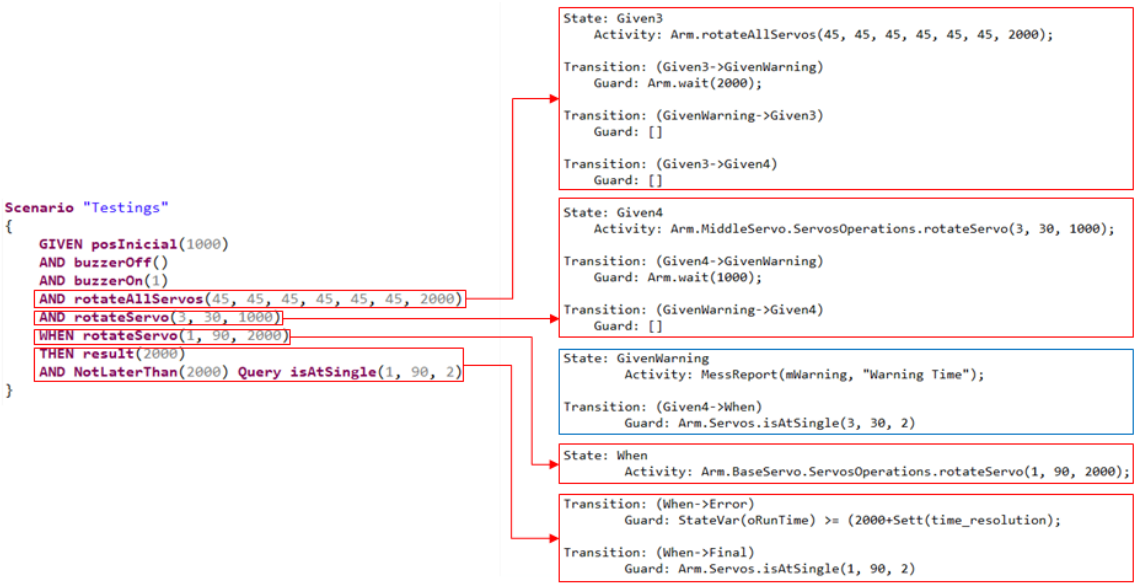

Figura 16: Second part of the generated code