Thymio

Lishun Su - fc56375, Ayla Stehling - fc63327

May 28, 2024

1 Introduction

In this document, the program will be explained, detailing how a DSL was created for the Aseba Thymio programming language to make it easier for students with no programming experience to use.

2 Meta-Model

The language design was started with creating a meta-model:

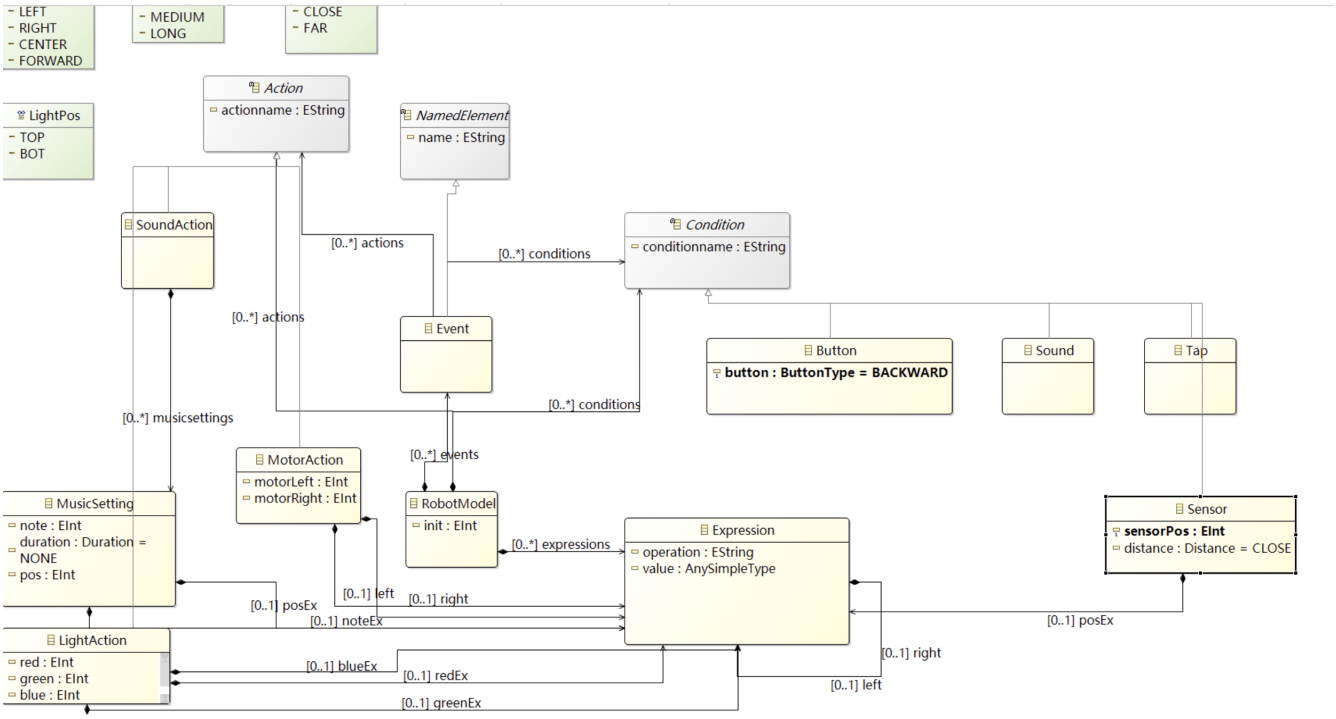

Figure 1: Meta-Model

The four main instances in the meta model are RobotModel, Event, Action, and Condition. A RobotModel consists of events, conditions, and actions. An Event has associations with actions and conditions, so an Event is made up of Condition and Action. A Condition can be of the type Button, Sound, Tap, or Sensor. An Action is either a MusicSetting, SoundAction (which is made up of a MusicSetting), or MotorAction. MotorAction, LightAction, MusicSetting, Sensor, and RobotModel can all have an Expression. Later in the grammar, the expression will be implemented to contain arithmetic expressions (+, -, *, /), as well as int. Also defined in the meta model are the options for LightPos, Distance, Duration, and ButtonType.

3 Grammar

The rules of the grammar are defined in the MyDsl.xtxt file.

```
RobotModel returns RobotModel:
    {RobotModel}
    ((events+=Event) |
    ('Action:' '-' actions+=Action ('-' actions+=Action)*) |
    ('Condition:' '-' conditions+=Condition ('-' conditions+=Condition)*) |
    (expressions+=Expression))* ;

Expression returns Expression:
    PlusOrMinus ;

PlusOrMinus returns Expression:
    MultOrDiv ({Expression.left=current} operation=('+' | '-') right=MultOrDiv)* ;

MultOrDiv returns Expression:
    Primary ({Expression.left=current} operation=('*' | '/') right=Primary)* ;

Primary returns Expression:
    '(' Expression ')' | Atomic ;

Atomic returns Expression:
    {Expression} (value=EInt) ;

Event returns Event:
    {Event} 'Event' name=EString
    'conditions:' conditions+=Condition (and='and' conditions+=Condition)*
    'actions:' actions+=Action (',' actions+=Action)* ;

Condition returns Condition:
    Button | Sensor | Sound | Tap ;

Button returns Condition:
    {Button} 'Button' name=EString 'button' button=[ButtonType] ;

Sensor returns Condition:
    {Sensor} 'Sensor' name=EString 'sensorPos' sensorPosition=ESensorPos 'distance'
    distance=EDistance ;

Sound returns Condition:
    {Sound} 'Sound' name=EString ;

Tap returns Condition:
    {Tap} 'Tap' name=EString ;

Action returns Action:
    MotorAction | SoundAction | LightAction ;

MotorAction returns Action:
    {MotorAction} 'MotorAction' name=EString 'motorLeft' motorLeft=EInt 'motorRight' motorRight=EInt

SoundAction returns Action:
    {SoundAction} 'SoundAction' name=EString 'set:' '(' note=ENote ',' duration=EDuration ','
    pos=EPos ')' ;
```

```
LightAction returns Action:
    {LightAction} 'LightAction' name=EString 'pos:' pos=EPos 'red:' red=EInt 'green:' green=EInt
    'blue:' blue=EInt ;
```

What is not yet implemented in the meta model are syntax-specific elements in the language and arithmetic equations.

Syntax-specific elements include constructing lists of conditions and actions starting with 'Condition:' and 'Action:' and listing elements with '-'. All tags are also defined in the grammar, e.g., 'MotorAction', 'motorLeft', and 'motorRight' for MotorAction.

Arithmetic functions +, -, *, and / are applicable to all numbers. For example, you can add sensor positions 1 and 4 to get sensor position 5. For arithmetic operations +, -, , and /, there is a hierarchy of execution. In this hierarchy, multiplication () and division (/) take precedence over addition (+) and subtraction (-).

The code in the end will look like this:

```
Condition:
    -Sensor LeftSensor sensorPosition 1 detectObstacle
    OBSTACLE

Action:
    -MotorAction TurnRight motorLeft 200 motorRight -200

Event ObjectOnLeft
    conditions: LeftSensor actions: TurnRight
```

Start by declaring the conditions and actions, which will be used later to build the event. The name of the event here is "ObjectOnLeft". More details of the code will be described in the following section.

Conditions:

Button:

```
Button "ButtonBackward" BACKWARD
```

For the button, the name is followed by the ButtonType. ButtonType can be: BACKWARD, LEFT, RIGHT, CENTER, or FORWARD.

Sound:

```
Sound "SoundName"
```

The sound condition only has a name of the condition.

Tap:

```
Tap "TapName"
```

Tap also only has a name the name of the condition.

Sensor:

```
Sensor "LeftSensor" sensorPosition 1 detectObstacle OBSTACLE
```

In the sensor condition, the name is followed by 'sensorPosition' and then a number between 1 and 7 for the corresponding sensor. There are two more sensors on the bottom of the robot, sensors 8 (bottom right) and 9 (bottom left). Then the 'detectObstacle' is defined to decide wether a object has triggered the sensor or no object has been detected, either: OBSTACLE or NOOBSTACLE.

The corresponding sensors are:

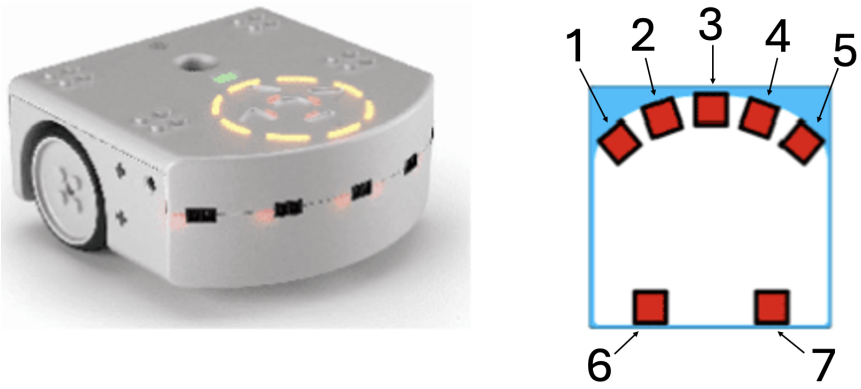

Figure 2: Sensors 1 to 5 in the front, sensors 6 and 7 in the back [Jiw14]
(not shown: sensor 8 and 9 on the bottom)

Actions:
An Action is one of three types: MotorAction, SoundAction or LightAction.

MotorAction:

```
MotorAction "TurnRight" motorLeft 200 motorRight -200
```

Starting with 'MotorAction' and then followed by a name, here TurnRight. Then it is defined how the robot should turn, drive, or stop. The keywords are 'motorLeft' and 'motorRight', followed by a number between 500 and -500.

SoundAction:

```
SoundAction "Play" set: (1, MEDIUM, 1)
```

After the name of the action, a 'set:' is defined in parentheses. This begins with a note, which is a number between 1 and 5, followed by a duration of NONE, MEDIUM, or LONG, and a pos between 1 and 6.

LightAction:

```
LightAction "TurnOnLight" pos: TOP red: 32 green: 0 blue: 0
```

After the name, the position of the light is defined. After 'pos:' either TOP or BOT is chosen. Then, the color of the light is designated. The colors are based on RGB values, which means 'red:', 'blue:', and 'green:' are all values between 0 and 32. Whichever color is set to 32 is the color of the light. In this example, the color is red.

Actions and conditions are listed with '-' and follow this pattern:

```
Action:
    -MotorAction "TurnRight" motorLeft 200 motorRight -200

Condition:
    -Sensor "LeftSensor" sensorPos 0 distance CLOSE
```

Actions and conditions are listed with '-' and follow the same pattern as defined above.

Lastly, with actions and conditions, events will be built.
All events start with a name. Afterwards, an event can have conditions and actions. If an event has more than one condition, they are separated by 'and'. If it has more than one action, they are separated by a ','.

```
Event ObjectOnLeft
    conditions: LeftSensor actions: TurnRight
```

4 Validation

In the file MyDslValidator.java validators have been implemented.
Errors implemented are:

- **checkActions**
  - Can't add two of the same actions.
- **checkNames**
  - Can't declare instances with the same name.
- **checkMotorSpeed**
  - The speed cannot exceed the limit of -500 to 500.
- **checkSensor**
  - The sensor position has to be between 1 and 9.
- **checkLightColor**
  - The color setting has to be between 0 and 32.
- **checkSound**
  - The sound note can only be between 1 and 5.
  - The sound position can only be between 1 and 6.
  - The sound position is already defined.

Warnings implemented are:

- **processEventsCondition**
  - The button type has already been defined.
- **checkTapDeclare**
  - Tap function has already been defined.
- **checkSoundDeclare**
  - Sound function has already been defined.

These have been implemented as warnings because while defining these conditions, they can be defined twice. However, when used in an event, each can only be used one time.

Three examples of the validations in the file are:

Sensors:
The Thymio robot has 9 sensors. In the DSL, these sensors are referenced with numbers between 1 and 9. Any number outside this range is not allowed, and an error is thrown.

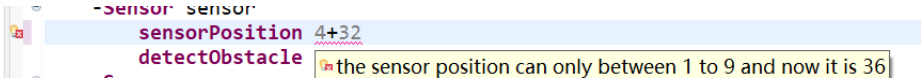

Figure 3: Sensors 1 to 9 exists

An example of an implementation in MyDslValidator.java of a validator is:

```
public static final String INVALID_SENSOR_POS = "the sensor position is invalid";

@Check
public void checkSensor(Sensor s){
    int max = 9, min = 1;
    if(s.getSensorPos() < min || s.getSensorPos() > max){
        error("the sensor position can only between " + min +" to " + max, s,
        Project2Package.eINSTANCE.getSensor_SensorPos(), INVALID_SENSOR_POS);
    }
}
```

Motor Speed:
A Thymio robot has a speed limit of -500 to 500. Therefore, each motor can only accept inputs within these values. If values outside this range are provided, an error will be thrown.

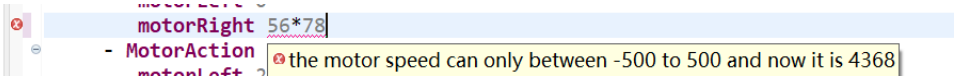

Figure 4: Speed of a robot has to be between -500 and 500

Unique Name:
Since we're working with names of conditions and actions to create an event, these names must be unique. No name can be used twice, as the program won't know which action or condition is intended. Additionally, an event cannot be referenced in another event, action, or condition.

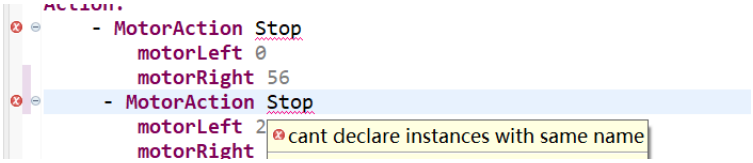

Figure 5: Every name has to be unique

5 TypeCheck

Type checking is implemented in the MyDslValidator.java:

```
public static final String INVALID_CONDITION = "the contidion cant ";
@Check
public void checkCondition(Event e) {
    if(e.getConditions().size()>1) {
        if( (e.getConditions().get(0) instanceof Tap || e.getConditions().get(0) instanceof Sound)){
            error("this condition can only be singular", e,
            Project2Package.eINSTANCE.getEvent_Conditions(), INVALID_CONDITION);
        } else {
            Condition c = e.getConditions().get(0);
            for(int i = 1; i < e.getConditions().size(); i++) {
```

```
                if(c instanceof Button && !(e.getConditions().get(i) instanceof Button)){
                    error("this condition can only assing with same type of condition", e,
                    Project2Package.eINSTANCE.getEvent_Conditions(), INVALID_CONDITION);
                } else if(c instanceof Sensor &&!(e.getConditions().get(i) instanceofSensor)){
                    error("this condition can only assing with same type of condition", e,
                    Project2Package.eINSTANCE.getEvent_Conditions(), INVALID_CONDITION);
                }
            }
        }
    }
}
```

The implemented type checking verifies the types of conditions.
In an event, conditions can be joined with "and," but these conditions must be of the same type. For instance, two sensors or two buttons are allowed, but not a sensor and a button. Additionally, tap and sound cannot be combined with any other type.

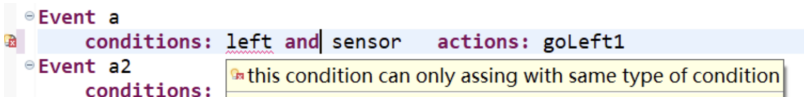

Figure 6: Conditions have to be Same Type

6 Auto Complete

Implemented in MyDslProposalProvider.java, with the path:
org.xtext.project2.robotdsl.ui/src/org/xtext/example/mydsl1/ui/contentassist/MyDslProposalProvider.java
The autocomplete feature implemented serves two functions. Firstly, when you have an event, it provides a list of possible conditions or actions.

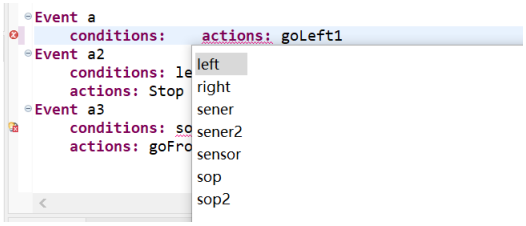

Figure 7: Auto Complete List of Conditions

The other autocomplete feature suggests options for buttons, light positions, sensor distances, and sound durations, suggestions which are defined in the meta model (see: 1). Additionally, autocomplete is implemented for light positions, sensor positions, sound notes, and sound positions, where auto complete lists the possible numbers.

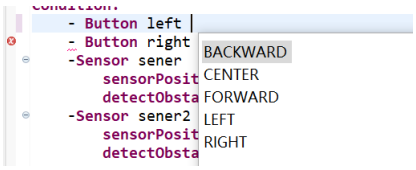

Figure 8: Auto Complete ButtonType

An example code:

```
@Override
public void completeButton_Button(EObject model, Assignment assignment,
ContentAssistContext context, ICompletionProposalAcceptor acceptor) {
    acceptor.accept(createCompletionProposal("FORWARD","FORWARD", null, context));
    acceptor.accept(createCompletionProposal("BACKWARD","BACKWARD", null, context));
    acceptor.accept(createCompletionProposal("LEFT","LEFT", null, context));
    acceptor.accept(createCompletionProposal("RIGHT","RIGHT", null, context));
    acceptor.accept(createCompletionProposal("CENTER","CENTER", null, context));
}
```

7 Quick Fix

Implemented in MyDslQuickfixProvider.java, with the path:
org.xtext.project2.robotdsl.ui/src/org/xtext/example/mydsl1/ui/quickfix/MyDslQuickfixProvider.java
When using the DSL, if you encounter an error message from one of the implemented validators, a quick fix is suggested. The implemented quick fixes include fixing the sensor position and fixing invalid conditions.

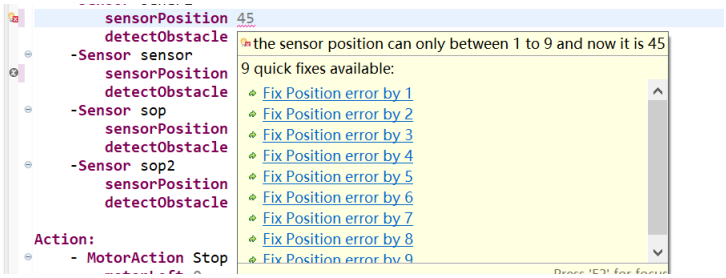

Figure 9: Quick Fixes for Sensors

An example code (the code only shows 2 sensors; in the implementation, all sensors are included):

```
@Fix(MyDslValidator.INVALID_SENSOR_POS)
public void fixSensorPos(final Issue issue, IssueResolutionAcceptor accpetor) {
    accpetor.accept(issue, "Fix Position error by 1", "Fix Position error by 1", null,
    new ISemanticModification() {
        @Override
        public void apply(EObject element, IModificationContext context) throws Exception {
            Sensor sensor = (Sensor) element;
            sensor.getPosEx().setValue(1);
        }
    });
    accpetor.accept(issue, "Fix Position error by 2", "Fix Position error by 2", null,
    new ISemanticModification() {
        @Override
        public void apply(EObject element, IModificationContext context) throws Exception {
            Sensor sensor = (Sensor) element;
            sensor.getPosEx().setValue(2);
        }
    });
}
```

8 Demos

Following are four test cases. Each test case will get more complicated, starting with showing simple functionalities of the robot and ending with a complicated program.
Each program demonstrates slightly different ways of writing the code. The differences are in how many lines the definition of conditions, actions, and events are divided into. This can then be chosen by the user according to their preferences.

8.1 simpleTests

This demo is to check if the different conditions and actions of the robot in the DSL work. Arithmetic equations are also tested. It checks the button, sound, and tap conditions, as well as the actions of the light, motors, and sound.

```
Condition:
    - Tap tap
    - Button center CENTER
    - Button left LEFT
    - Sound sound

Action:
    - MotorAction goFront motorLeft 300 motorRight 200+100
    - MotorAction spinLeft motorLeft 300 motorRight 0
    - MotorAction spinRight motorLeft 0 motorRight 300
    - MotorAction stop
    - LightAction lightred position:TOP red:32
    - LightAction lightblue position:TOP blue:32
    - LightAction lightgreen position:BOT green:32
    - LightAction light position:TOP red:12 blue:32
    - SoundAction playmusic set:(2, MEDIUM, 2), (5, MEDIUM, 4), (2, LONG, 5)

Event event
    conditions: center actions: goFront, lightred, playmusic

Event event2
    conditions: left actions: stop, lightblue

Event spinEvent
    conditions: sound actions: spinLeft, light

Event spinRightEvent
    conditions: tap actions: spinRight, light
```

8.2 followBlackLine

A simple first demo involves following a black line. This demonstrates that short programs can be written and then followed by the Thymio robot.

```
Condition:
    -Tap y
    -Sensor obstaculel
        sensorPosition 8
        detectObstacle NOOBSTACLE
    -Sensor obstaculer
        sensorPosition 9
        detectObstacle NOOBSTACLE
```

```
    -Sensor noobstacul
        sensorPosition 8
        detectObstacle OBSTACLE
    -Sensor noobstacur
        sensorPosition 9
        detectObstacle OBSTACLE

Action:
    -MotorAction gofront motorLeft 200 motorRight 100+100
    -MotorAction turnLeft motorLeft 200+100 motorRight 0
    -MotorAction turnRight motorLeft 0 motorRight 300

Event event1 conditions: obstaculer and obstaculel actions: gofront
Event event2 conditions: obstaculer and noobstacul actions: turnLeft
Event event3 conditions: obstaculel and noobstacur actions: turnRight
```

8.3 followWhiteLine

Another simple demo is to follow a white line. In this test case, it is shown that small programs can have more than one functionality. The program follows a line and displays light.

```
Condition:
    -Sensor bottomLeftNo
        sensorPosition 8
        detectObstacle NOOBSTACLE
    -Sensor bottomRightNo
        sensorPosition 9
        detectObstacle NOOBSTACLE
    -Sensor bottomLeft
        sensorPosition 8
        detectObstacle OBSTACLE
    -Sensor bottomRight
        sensorPosition 9
        detectObstacle OBSTACLE
    -Button center CENTER

Action:
    -MotorAction stop
    -MotorAction gofront motorLeft 300 motorRight 200+100
    -MotorAction turnLeft motorLeft 200+100 motorRight -150
    -MotorAction turnRight motorLeft -150 motorRight 300
    - LightAction lightred position:TOP red:32
    - LightAction lightblue position:TOP blue:32
    - LightAction lightgreen position:BOT green:32

Event goforward conditions: bottomLeft and bottomRight actions: gofront, lightblue
Event event2 conditions: bottomLeft and bottomRightNo actions: turnRight, lightgreen
Event event3 conditions: bottomLeftNo and bottomRight actions: turnLeft, lightred
Event event4 conditions: center actions: stop
```

8.4 avoidObstacle

The complex demo involves avoiding obstacles. Depending on the direction the robot drives, it will display a different color.

```
Condition:
    -Sensor sensorMostLeft
        sensorPosition 1
```

```
        detectObstacle NOOBSTACLE
    -Sensor sensorLeft
        sensorPosition 2
        detectObstacle NOOBSTACLE
    -Sensor sensorMostRight
        sensorPosition 5
        detectObstacle NOOBSTACLE
    -Sensor sensorRight
        sensorPosition 4
        detectObstacle NOOBSTACLE
    -Sensor sensorMidlle
        sensorPosition 3
        detectObstacle NOOBSTACLE
    -Sensor sensorMostLeftOBSTACLE
        sensorPosition 1
        detectObstacle OBSTACLE
    -Sensor sensorLeftOBSTACLE
        sensorPosition 2
        detectObstacle OBSTACLE
    -Sensor sensorMostRightOBSTACLE
        sensorPosition 5
        detectObstacle OBSTACLE
    -Sensor sensorRightOBSTACLE
        sensorPosition 4
        detectObstacle OBSTACLE
    -Sensor sensorMidlleOBSTACLE
        sensorPosition 3
        detectObstacle OBSTACLE

Action:
    - MotorAction goLeft
        motorLeft 500
        motorRight 0
    - MotorAction goLeft2
        motorLeft 500
        motorRight -250
    - MotorAction goRight
        motorLeft 0
        motorRight 500
    - MotorAction goRight2
        motorLeft -250
        motorRight 50
    - MotorAction forward
        motorLeft 500
        motorRight 500
    - MotorAction frontdetect
        motorLeft -500
        motorRight 500
    - LightAction lightred position:TOP red:32
    - LightAction lightblue position:TOP blue:32
    - LightAction lightgreen position:TOP green:32
    - LightAction lightyellow position:TOP red:32 green:32


Event a
    conditions: sensorLeftOBSTACLE actions: goLeft, lightred
```

```
Event a2
    conditions: sensorMostLeftOBSTACLE actions: goLeft2, lightred
Event a3
    conditions: sensorMidlleOBSTACLE actions: frontdetect, lightyellow
Event a4
    conditions: sensorMostRightOBSTACLE actions: goRight2, lightblue
Event a5
    conditions: sensorRightOBSTACLE actions: goRight, lightblue
Event a6
    conditions: sensorLeft and sensorMidlle and sensorMostLeft and sensorMostRight
    and sensorRight actions: forward, lightgreen
```

9 Tests

In the project we have test classes for the code generator, parsing and validator.

9.1 Code Generator Test

File: RobotDslCodeGenTest.xtend
During the code generator tests we test the generation of aseba code. Tested is:

- testAsebaFile()
  - Tests if the aseba code is generated correctly. Testing each possible condition (button, sensor, tap and sound) and action (motor, light and sound) creation once. And then if the onevent is generated correctly by defining events in our dsl code.
- testAsebaFileWithAritmetics()
  - Tests if the arithmetic operations are correctly translated into aseba. Tested are addition and multiplication.
- testAsebaFileWithNoEvent()
  - Tests the code generation if no event is defined in the dsl code.

Each test is designed by havong the dsl code in the beginning and then checking if when generated to aseba code it equals to the aseba code in .assertCompilesTo().

9.2 Parsing Test

File: RobotDslParsingTest.xtend
For testing parsing each action and condition is tested that an error is thrown for an invalid declaration. This means for example if a button is defined with a number instead of a buttontype, or an event isn't defined with an action.

- loadValidModel()
  - Checks if a correct model is a correctly parsed.
- loadInvalidModel()
  - Each condition and action should have a name.
- loadNoAction()
  - Event missing action.
- loadEventNoCondition()

  - Event missing condition.

- loadInvalidLightAction()
  - Light action missing position. Action needs the position of the light.
- loadInvalidSoundAction()
  - A sound action missing set:.
- loadInvalidMotorAction()
  - Motor action missing name.
- loadInvalidSensorCondition()
  - Sensor missing name.
- loadInvalidTapAndSoundCondition()
  - Both tap and sound are missing name.
- loadInvalidButtonCondition()
  - Buttontype defined with a number. Buttontypes are BACKWARD, LEFT, RIGHT, CENTER or FORWARD

9.3 Validator Test

File: RobotDslValidatorTest.xtend
To test the different implemented validators. In section 4 the validators are explained. Each validator is tested in one of the following methods:

- testInvalidActionType()
- testInvalidNameDefinition()
- testDefineTwoSameButtonConditions()
- testInvalidMotorSpeed()
- testInvalidSensorPosition()
- testInvalidSingularConditions()
- testInvalidMultipleConditions()
- testInvalidSoundNote()
- testInvalidSoundPosition()
- testSameSoundPosition(
- testDefineTwoTapCondition()
- testDefineTwoSoundCondition()

References

[Jiw14] Stéphane Magnenat Jiwon Shin R. Siegwart. *Visual Programming Language for Thymio II Robot*. sensors ith numbers. 2014. URL: https://www.semanticscholar.org/paper/Visual-Programming-Language-for-Thymio-II-Robot-Shin-Siegwart/754041308dcac5657dd703bb89d55a8citing-papers (visited on 05/22/2023).