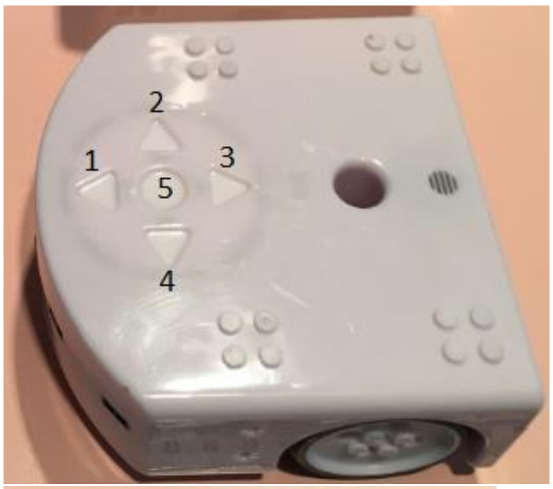
Bruno Liu
fc56297
Tomás
Carreira
fc50760

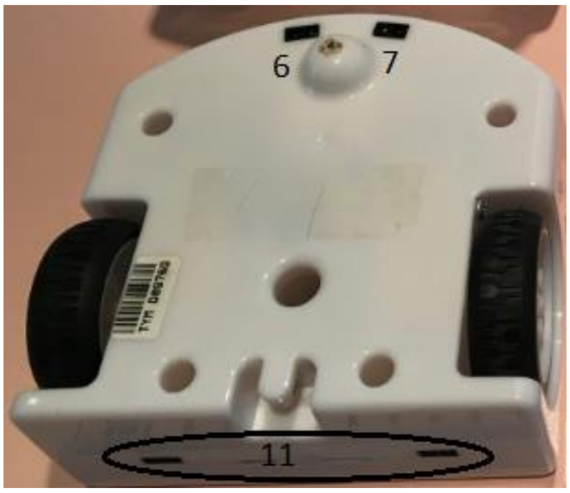
DOCUMENTATION FOR THE SIMPLEMIODSL

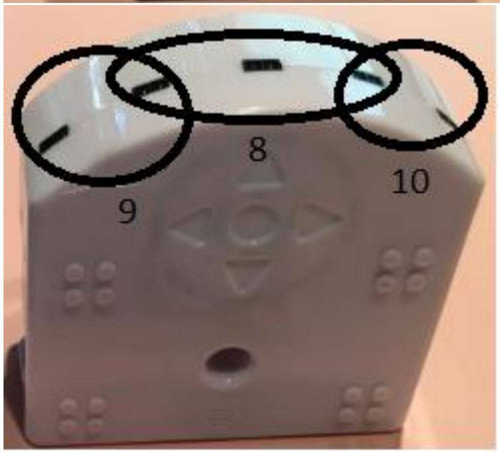
Domain-Specific Languages

SimpleMio is a domain-specific language (DSL) designed to simplify the programming of robotic behaviors. This documentation provides an in-depth look at the components of SimpleMio, including the meta model, grammar, type checking, code generation, quick fixes, and auto-completion features. The aim is to provide to high-school students an intuitive syntax so they can set some instruction to the thymio robot without needing to know how to program.

Specification of the sensors when mentioned:



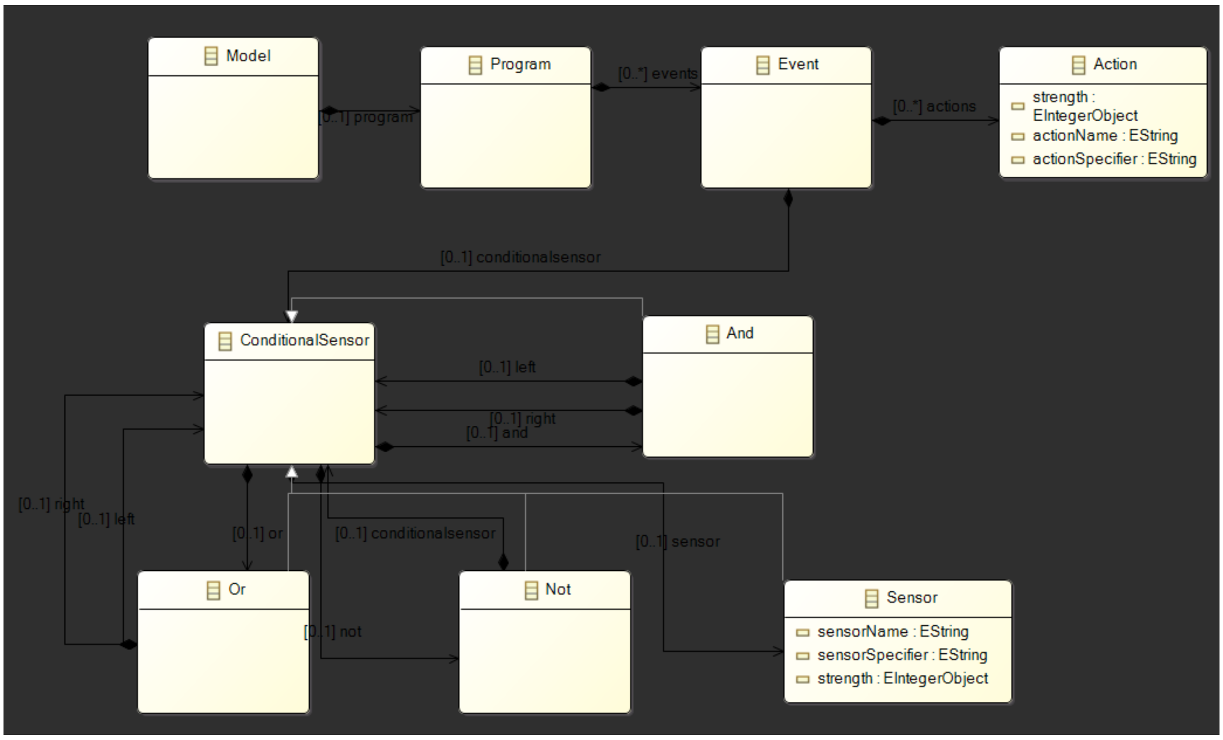




1. Used for button up
2. Used for button right
3. Used for button down
4. Used for button left
5. Used for button center
6. Used for line right
7. Used for line left
8. Used for obstacle front
9. Used for obstacle left
10. Used for obstacle right
11. Used for obstacle back

Metamodel

This is the meta model that defines the structure of the SimpleMio language.



It includes the following key entities:

- Model: The root entity that contains the entire program;
- Program: Represents a collection of events;
- Event: Defines a trigger and associated actions;
- Action: Represents an action to be performed;
  - Contains:
    - actionName: EString
    - actionSpecifier: EString
    - strength: EInteger
- ConditionalSensor: Represents a sensor condition;
- Sensor: Represents a sensor;
  - Contains:
    - sensorName: EString
    - sensorSpecifier: EString
    - strength: EInteger
- Or: used to set precedence when building a condition;
- Not: used to set precedence when building a condition;
- And: used to set precedence when building a condition.

As you can see by the image, these are the entity relationships:

- A “Model” contains a “Program”;
- A “Program” contains multiple “Event”;
- An “Event” has a “ConditionalSensor” and multiple “Action”;
- A “ConditionalSensor” contains an “Or”, an “And, a “Not” and a “Sensor”;
- An “Or” contains 2 “ConditionalSensor”, the left part and the right part of the “Or”;
- An “And” contains 2 “ConditionalSensor”, the left part and the right part of the “And”;

- A “Not” contains a “ConditionalSensor”;
- The “Sensor”, the “Or”, the “And” and the “Not” are all super types of “ConditionalSensor”.

Grammar

The grammar that defines the syntax for writing SimpleMio programs. Below are the key grammar rules:

```
Model returns Model:
	{Model}
	program=Program;
Program returns Program:
	(events+=Event)+
	;

Event returns Event:
	{Event}
	conditionalsensor=Or "->" actions+=Action ("," actions+=Action)*;

Action returns Action:
	{Action}
	actionName=("move" | "led" | "turn" | "stop")
	(actionSpecifier=("left" | "right" | "forward" | "backward" | "red" |
"green" | "blue" | "off"))?
	("@" strength=EInt)?
;

Or returns ConditionalSensor:
	And ({Or.left=current} "or" right=And)*
;

And returns ConditionalSensor:
	Not ({And.left=current} "and" right=Not)*
;

Not returns ConditionalSensor:
	{Not}("not" conditionalsensor=Paren)
		| {Not}("not" conditionalsensor=Sensor)
		| Sensor
		| Paren
;

Paren returns ConditionalSensor:
	"(" Or ")"
;

Sensor returns Sensor:
	{Sensor}
	sensorName=("obstacle" | "sound" | "line" | "button" | "motor")
	(sensorSpecifier=("front" | "back" | "left" | "right" | "up" | "down"
| "center"))?
	('@' strength=EInt)?
;

@Override
terminal SL_COMMENT: '#' !('\n'|'\r')* ('\r'? '\n')?;

EString returns ecore::EString:
	STRING | ID;

EInt returns ecore::EInt:
	'-'? INT;
```

Key Components:

- **Model Structure:** The grammar comprises a hierarchical structure with a root **Model** containing a **single Program**, which, in turn, consists **of multiple Events**.
- **Event Definition:** Each **Event** combines a **conditionalSensor** expression with a sequence of **actions**, allowing for complex behavior triggers.
- **Actions and Sensors: Actions** such as **move, led, turn and stop** are defined alongside **sensors** like **obstacle, sound, line, button and motor**, enabling diverse event triggers based on environmental stimuli.

An **event** is composed by a **conditionalSensor** followed by a "**->**" and at least 1 **action**, each action is separated by a comma ",", i.e., **conditionalSensor -> action (","action)***

A **conditional sensor** is a boolean expression of **sensors** that can have the operations **not**, **and**, **or** or **parenthesis**.

A **sensor** is composed by a **sensorName** (obstacle, sound, line, button or motor), a **sensorSpecifier** (front, back, left, right, up, down, or center) and optionally the user can set the strength of the sensor by adding a value after the "@" symbol.

An **action** is composed by an **actionName** (move, led, turn or stop), an **actionSpecifier** (left, right, forward, backward, red, green or blue) and optionally the user can set the strength of the action by adding a value after the "@" symbol.

The user of the DSL can also add **comments** to the program by adding a "**#**" before the comment.

SimpleMio's grammar provides a robust foundation for defining robotic behaviors, offering users a flexible and intuitive language for programming diverse actions.

Validator

The validator ensures that the SimpleMio code adheres to defined constraints. Key checks include:

- Action Specifiers: Validates that actions have correct specifiers.
- Sensor Specifiers: Ensures sensors have valid specifiers.
- Intensity Values: Checks that intensity values are within the allowed range (0-10).

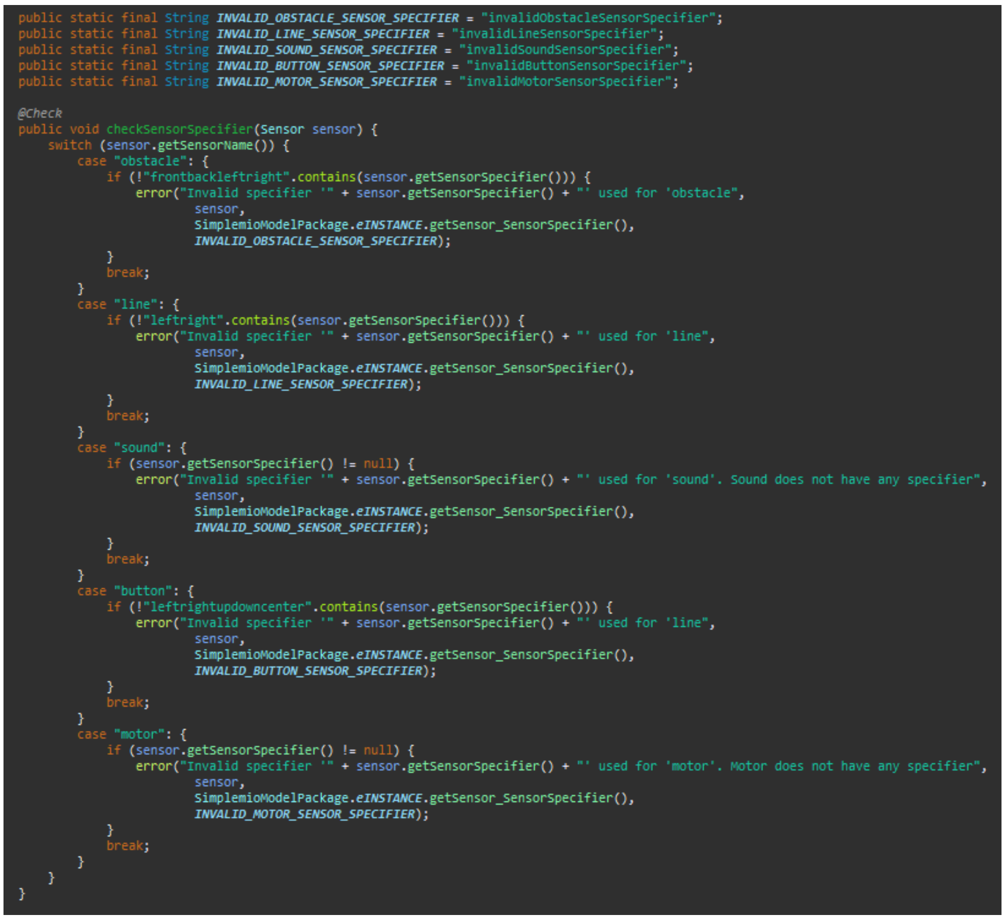
Validator Implementation

The `SimpleMioValidator` class contains these custom validation rules:

Invalid actions specifiers:

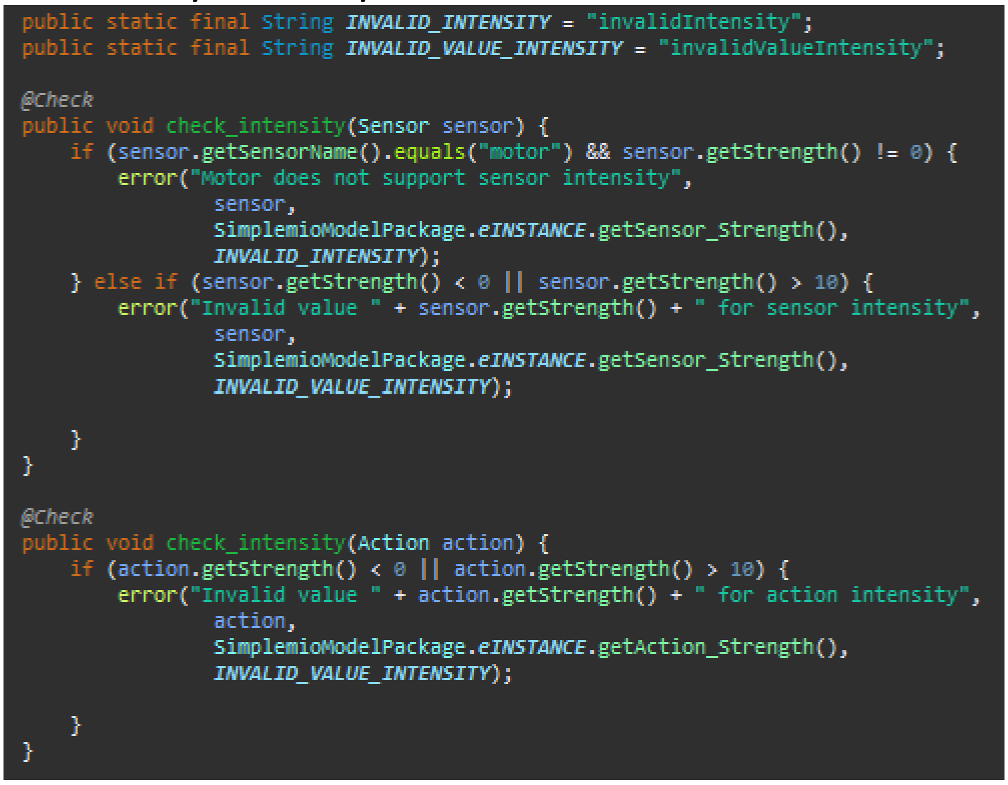
```
public static final String INVALID_MOVE_ACTION_SPECIFIER = "invalidMoveActionSpecifier";
public static final String INVALID_LED_ACTION_SPECIFIER = "invalidLedActionSpecifier";
public static final String INVALID_TURN_ACTION_SPECIFIER = "invalidTurnActionSpecifier";
public static final String INVALID_STOP_ACTION_SPECIFIER = "invalidStopActionSpecifier";
@Check
public void checkActionSpecifier(Action action) {
    switch (action.getActionName()) {
        case "move": {
            if (!(action.getActionSpecifier().equals("forward") ||
action.getActionSpecifier().equals("backward"))) {
                error("Invalid specifier '" + action.getActionSpecifier() +  "' used for 'move'",
                            action,   SimplemioModelPackage.eINSTANCE.getAction_ActionSpecifier(),
                                                        INVALID_MOVE_ACTION_SPECIFIER);
                }
                break;
            }

        case "led": {
            if (!("redbluegreenoff".contains(action.getActionSpecifier()))) {
                error("Invalid specifier '" + action.getActionSpecifier() +  "' used for 'led'",
                            action, SimplemioModelPackage.eINSTANCE.getAction_ActionSpecifier(),
                                                        INVALID_LED_ACTION_SPECIFIER);
            }
            break;
        }
        case "turn": {
            if (!(action.getActionSpecifier().equals("left") ||action.getActionSpecifier().equals("right"))) {
                error("Invalid specifier '" + action.getActionSpecifier() +  "' used for 'turn'",
                            action,   SimplemioModelPackage.eINSTANCE.getAction_ActionSpecifier(),
                                                        INVALID_TURN_ACTION_SPECIFIER);
            }
            break;
        }
        case "stop": {
            if (action.getActionSpecifier() != null) {
                error("Invalid specifier '" + action.getActionSpecifier() + "' used for 'stop'. Stop does not
have any specifier",
                             action, SimplemioModelPackage.eINSTANCE.getAction_ActionSpecifier(),
                                                        INVALID_STOP_ACTION_SPECIFIER);
            }
            break;
        }
    }
}
```

Invalid sensor specifiers:



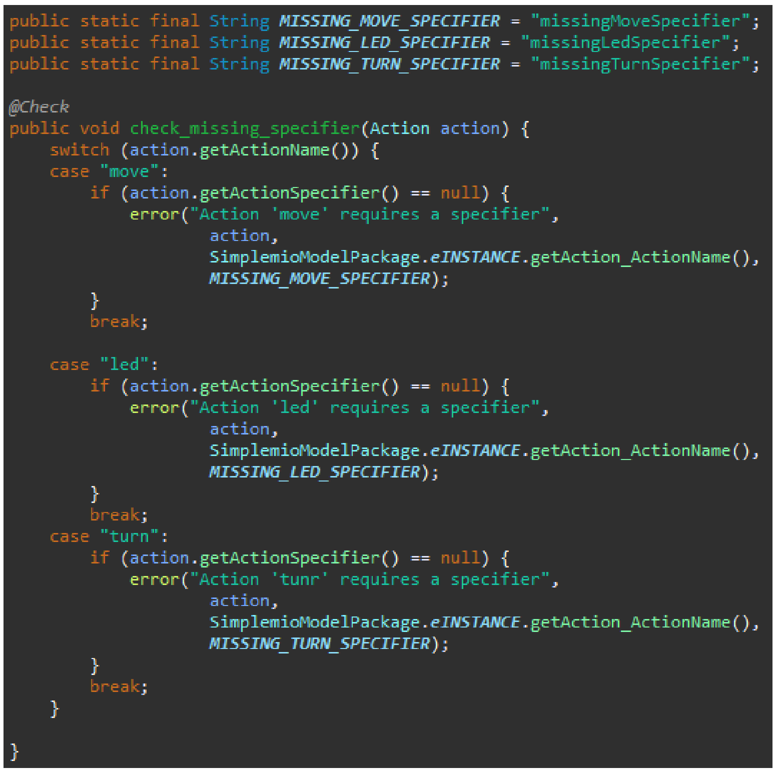
```
public static final String INVALID_OBSTACLE_SENSOR_SPECIFIER = "invalidObstacleSensorSpecifier";
public static final String INVALID_LINE_SENSOR_SPECIFIER = "invalidLineSensorSpecifier";
public static final String INVALID_SOUND_SENSOR_SPECIFIER = "invalidSoundSensorSpecifier";
public static final String INVALID_BUTTON_SENSOR_SPECIFIER = "invalidButtonSensorSpecifier";
public static final String INVALID_MOTOR_SENSOR_SPECIFIER = "invalidMotorSensorSpecifier";

@Check
public void checkSensorSpecifier(Sensor sensor) {
    switch (sensor.getSensorName()) {
        case "obstacle": {
            if (!"frontbackleftright".contains(sensor.getSensorSpecifier())) {
                error("Invalid specifier '" + sensor.getSensorSpecifier() + "' used for 'obstacle",
                        sensor,
                        SimplemioModelPackage.eINSTANCE.getSensor_SensorSpecifier(),
                        INVALID_OBSTACLE_SENSOR_SPECIFIER);
            }
            break;
        }
        case "line": {
            if (!"leftright".contains(sensor.getSensorSpecifier())) {
                error("Invalid specifier '" + sensor.getSensorSpecifier() + "' used for 'line",
                        sensor,
                        SimplemioModelPackage.eINSTANCE.getSensor_SensorSpecifier(),
                        INVALID_LINE_SENSOR_SPECIFIER);
            }
            break;
        }
        case "sound": {
            if (sensor.getSensorSpecifier() != null) {
                error("Invalid specifier '" + sensor.getSensorSpecifier() + "' used for 'sound'. Sound does not have any specifier",
                        sensor,
                        SimplemioModelPackage.eINSTANCE.getSensor_SensorSpecifier(),
                        INVALID_SOUND_SENSOR_SPECIFIER);
            }
            break;
        }
        case "button": {
            if (!"leftrightupdowncenter".contains(sensor.getSensorSpecifier())) {
                error("Invalid specifier '" + sensor.getSensorSpecifier() + "' used for 'line",
                        sensor,
                        SimplemioModelPackage.eINSTANCE.getSensor_SensorSpecifier(),
                        INVALID_BUTTON_SENSOR_SPECIFIER);
            }
            break;
        }
        case "motor": {
            if (sensor.getSensorSpecifier() != null) {
                error("Invalid specifier '" + sensor.getSensorSpecifier() + "' used for 'motor'. Motor does not have any specifier",
                        sensor,
                        SimplemioModelPackage.eINSTANCE.getSensor_SensorSpecifier(),
                        INVALID_MOTOR_SENSOR_SPECIFIER);
            }
            break;
        }
    }
}
```

Invalid intensity and intensity values:



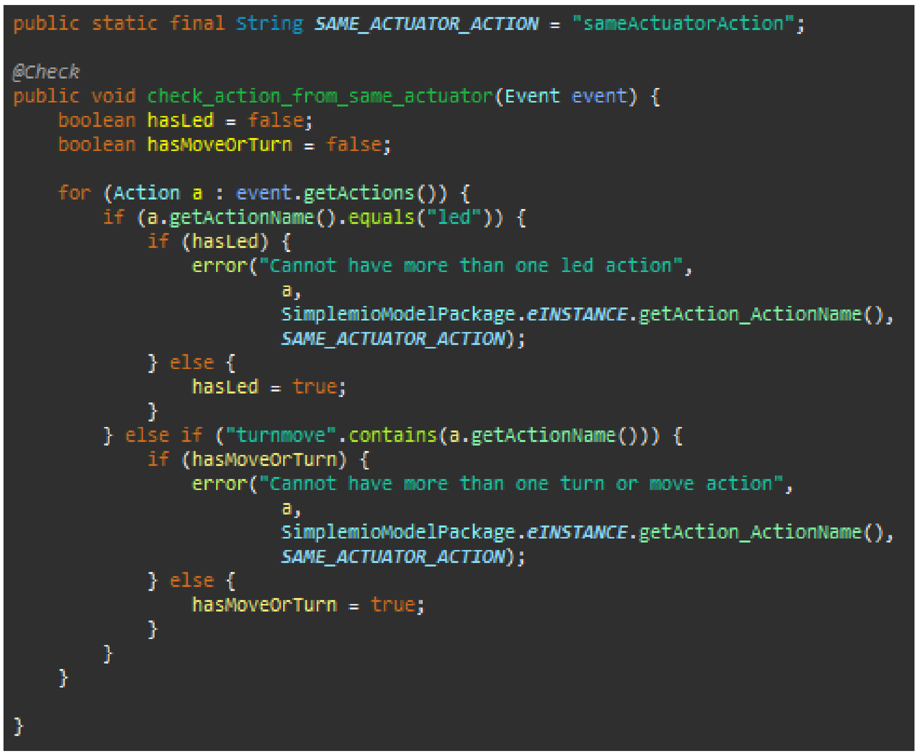
```
public static final String INVALID_INTENSITY = "invalidIntensity";
public static final String INVALID_VALUE_INTENSITY = "invalidValueIntensity";

@Check
public void check_intensity(Sensor sensor) {
    if (sensor.getSensorName().equals("motor") && sensor.getStrength() != 0) {
        error("Motor does not support sensor intensity",
                sensor,
                SimplemioModelPackage.eINSTANCE.getSensor_Strength(),
                INVALID_INTENSITY);
    } else if (sensor.getStrength() < 0 || sensor.getStrength() > 10) {
        error("Invalid value " + sensor.getStrength() + " for sensor intensity",
                sensor,
                SimplemioModelPackage.eINSTANCE.getSensor_Strength(),
                INVALID_VALUE_INTENSITY);

    }
}

@Check
public void check_intensity(Action action) {
    if (action.getStrength() < 0 || action.getStrength() > 10) {
        error("Invalid value " + action.getStrength() + " for action intensity",
                action,
                SimplemioModelPackage.eINSTANCE.getAction_Strength(),
                INVALID_VALUE_INTENSITY);

    }
}
```

Missing specifiers:



```java
public static final String MISSING_MOVE_SPECIFIER = "missingMoveSpecifier";
public static final String MISSING_LED_SPECIFIER = "missingLedSpecifier";
public static final String MISSING_TURN_SPECIFIER = "missingTurnSpecifier";

@Check
public void check_missing_specifier(Action action) {
    switch (action.getActionName()) {
    case "move":
        if (action.getActionSpecifier() == null) {
            error("Action 'move' requires a specifier",
                    action,
                    SimplemioModelPackage.eINSTANCE.getAction_ActionName(),
                    MISSING_MOVE_SPECIFIER);
        }
        break;

    case "led":
        if (action.getActionSpecifier() == null) {
            error("Action 'led' requires a specifier",
                    action,
                    SimplemioModelPackage.eINSTANCE.getAction_ActionName(),
                    MISSING_LED_SPECIFIER);
        }
        break;
    case "turn":
        if (action.getActionSpecifier() == null) {
            error("Action 'tunr' requires a specifier",
                    action,
                    SimplemioModelPackage.eINSTANCE.getAction_ActionName(),
                    MISSING_TURN_SPECIFIER);
        }
        break;
    }

}
```

Invalid same actuator:



```java
public static final String SAME_ACTUATOR_ACTION = "sameActuatorAction";

@Check
public void check_action_from_same_actuator(Event event) {
    boolean hasLed = false;
    boolean hasMoveOrTurn = false;

    for (Action a : event.getActions()) {
        if (a.getActionName().equals("led")) {
            if (hasLed) {
                error("Cannot have more than one led action",
                        a,
                        SimplemioModelPackage.eINSTANCE.getAction_ActionName(),
                        SAME_ACTUATOR_ACTION);
            } else {
                hasLed = true;
            }
        } else if ("turnmove".contains(a.getActionName())) {
            if (hasMoveOrTurn) {
                error("Cannot have more than one turn or move action",
                        a,
                        SimplemioModelPackage.eINSTANCE.getAction_ActionName(),
                        SAME_ACTUATOR_ACTION);
            } else {
                hasMoveOrTurn = true;
            }
        }
    }

}
```

To sum up, these are what each **action** or **sensor** is allowed to be followed by:

- **move** can be followed by forwards or backwards.
- **led** can be followed by red, blue or green.
- **turn** can be followed by right or left.
- **obstacle** can be followed by front, back, left or right.
- **line** can be followed by left or right.
- **button** can be followed by left, right, up, down and center.
- **sound, motor** and **stop** don’t have any specifier.
- The sensor **motor** and the action **stop** don’t support intensity.
- For all other **sensors** and **actions**, which support intensity, the intensity must be a number between 0 and 10.
- **The validator also doesn’t allow overlapping actions** like, more than 1 action led or more than 1 action turn or move in total.

Each error is also identified with a different tag so further it is possible to implement the quickfix.

Code Generation

The code generation is done in a very straight forward way. Every event corresponds to if *boolean expression* then *statements* in the aseba language. The boolean expression is the conditional sensor and the statements are the actions. The aseba language requires onevent *event* blocks and the same onevent cannot repeat, so it is necessary to duplicate code if the sensors in the same condicionalSensor are from different onevents. For example, if we have in simplemio button center and motor -> stop the code generation would be

```
onevent buttons
        if button.center > 0 and motor.speed != 0 then
                motor.left.target = 0
                motor.right.target = 0
onevent motor
         if button.center > 0 and motor.speed != 0 then
                motor.left.target = 0
                motor.right.target = 0
```

During the implementation of the code generation we discover that using if then blocks for multiple events didn't work very well, so the events are actually generated as if else blocks. A program with two events would generate this aseba

```
onvent event
        if 0 != 0 then
        elseif boolean expression then
                statements
        elseif boolean expression then
                statements
        end
```

The if 0 != 0 then is used to facilitate the code generator.

Listed below is how the sensors and actions are translated to aseba.

- obstacle front:
    - prox.horizontal[1] > 2000 or prox.horizontal[2] > 2000 or prox.horizontal[3] > 2000
- obstacle right:
    - prox.horizontal[3] > 2000 or prox.horizontal[4] > 2000
- obstacle left:
    - prox.horizontal[0] > 2000 or prox.horizontal[1] > 2000
- obstacle back:
    - prox.horizontal[5] > 2000 or prox.horizontal[6] > 2000
- line left:
    - prox.ground.delta[0] < 400

- line right:
    - prox.ground.delta[1] < 400
- sound
    - mic.intensity > mic
    - It is added to the top of the file mic.threshold = 250
- motor
    - (motor.left.speed != 0 or motor.right.speed != 0)
- button center
    - button.center > 0
- button up
    - button.up > 0
- button down
    - button.down > 0
- button left
    - button.left > 0
- button right
    - button.right > 0
- move forward
    - motor.left.target = 250
    - motor.right.target = 250
- move backward
    - motor.left.target = -250
    - motor.right.target = -250
- turn left
    - motor.left.target = -250
    - motor.right.target = 250
- turn right
    - motor.left.target = 250
    - motor.right.target = -250
- stop
    - motor.left.target = 0

    - motor.right.target = 0
- led red
    - call leds.top(127, 0, 0)
- led green
    - call leds.top(0, 127, 0)
- led blue
    - call leds.top(0, 0, 127)
- led off
    - call leds.top(0, 0 ,0)

The code generation of the intensity of the sensor or action is done by adding the expression * *intensity* / 5 after the threshold value. For example, the action move forward @10 would be translated to

motor.left.target = 250 * 10 / 5

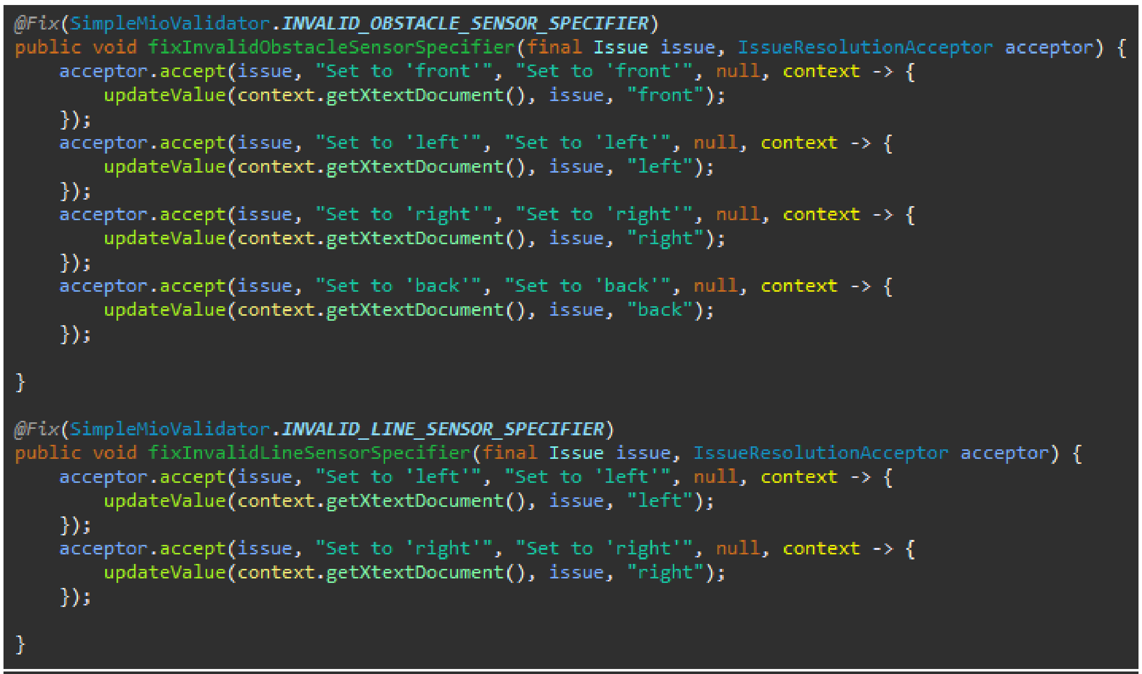
motor.right.target = 250 * 10 / 5

IDE Services

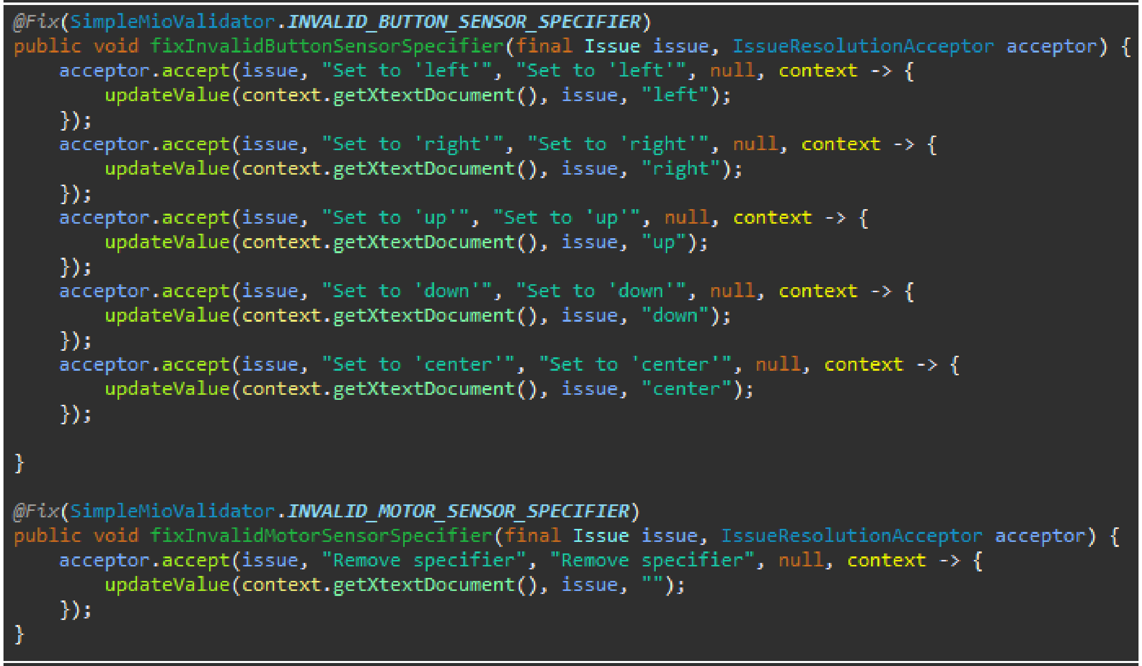
The IDE services that were implemented in the DSL were: the quickfix and the auto-completion

Quickfix

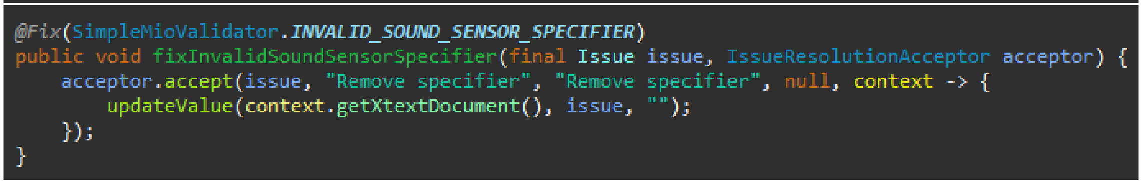
As mentioned before, for the quickfix this was the implementation to correct the errors thrown:



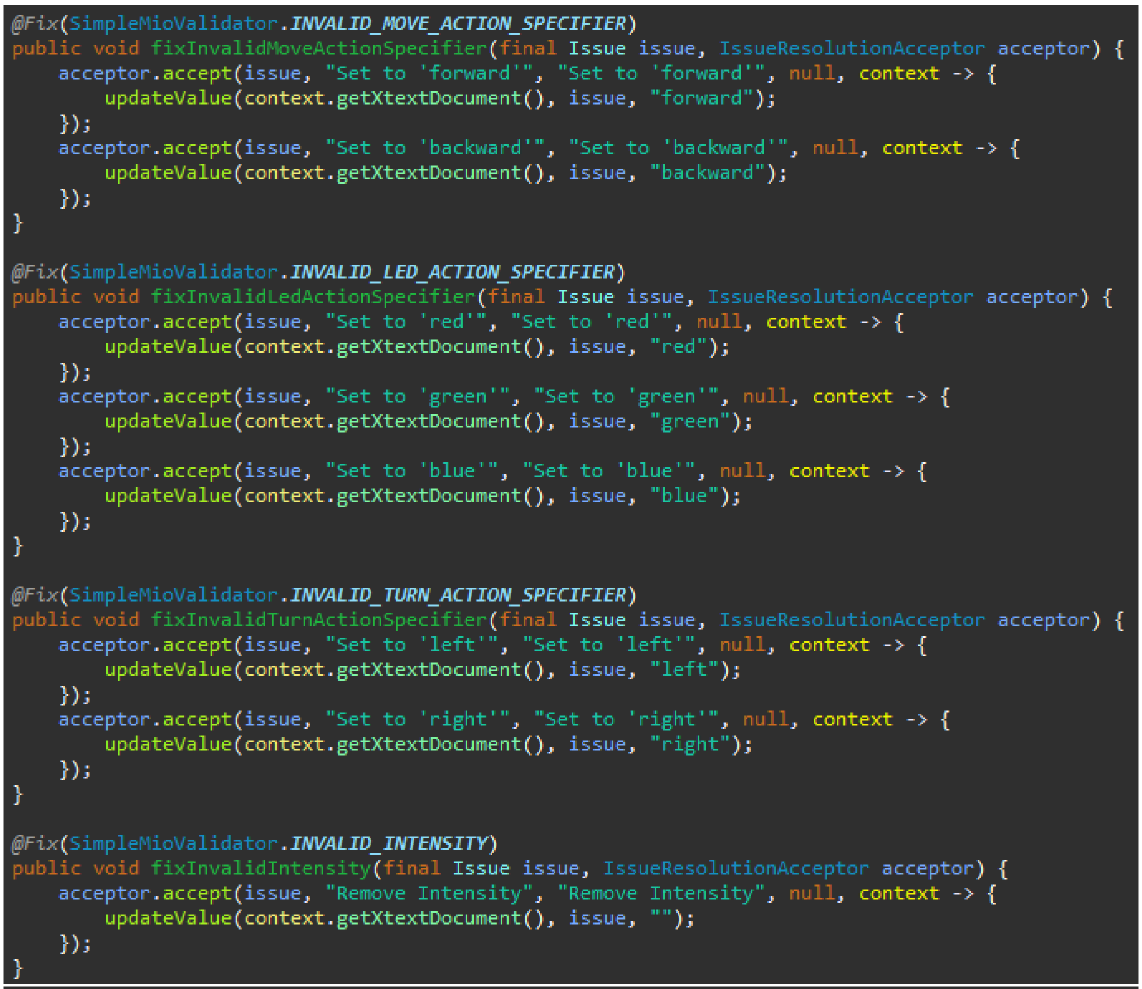
```
@Fix(SimpleMioValidator.INVALID_OBSTACLE_SENSOR_SPECIFIER)
public void fixInvalidObstacleSensorSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Set to 'front'", "Set to 'front'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "front");
    });
    acceptor.accept(issue, "Set to 'left'", "Set to 'left'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "left");
    });
    acceptor.accept(issue, "Set to 'right'", "Set to 'right'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "right");
    });
    acceptor.accept(issue, "Set to 'back'", "Set to 'back'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "back");
    });

}

@Fix(SimpleMioValidator.INVALID_LINE_SENSOR_SPECIFIER)
public void fixInvalidLineSensorSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Set to 'left'", "Set to 'left'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "left");
    });
    acceptor.accept(issue, "Set to 'right'", "Set to 'right'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "right");
    });

}
```



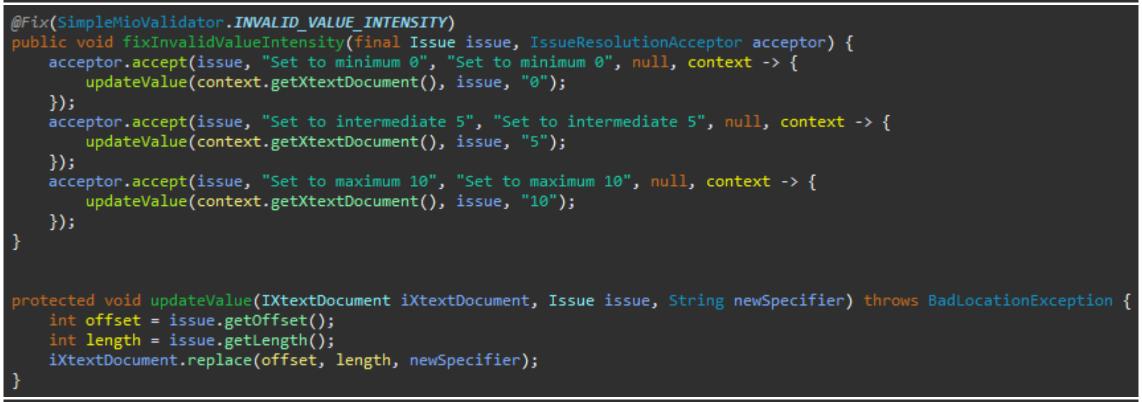
```
@Fix(SimpleMioValidator.INVALID_BUTTON_SENSOR_SPECIFIER)
public void fixInvalidButtonSensorSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Set to 'left'", "Set to 'left'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "left");
    });
    acceptor.accept(issue, "Set to 'right'", "Set to 'right'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "right");
    });
    acceptor.accept(issue, "Set to 'up'", "Set to 'up'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "up");
    });
    acceptor.accept(issue, "Set to 'down'", "Set to 'down'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "down");
    });
    acceptor.accept(issue, "Set to 'center'", "Set to 'center'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "center");
    });

}

@Fix(SimpleMioValidator.INVALID_MOTOR_SENSOR_SPECIFIER)
public void fixInvalidMotorSensorSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Remove specifier", "Remove specifier", null, context -> {
        updateValue(context.getXtextDocument(), issue, "");
    });
}
```



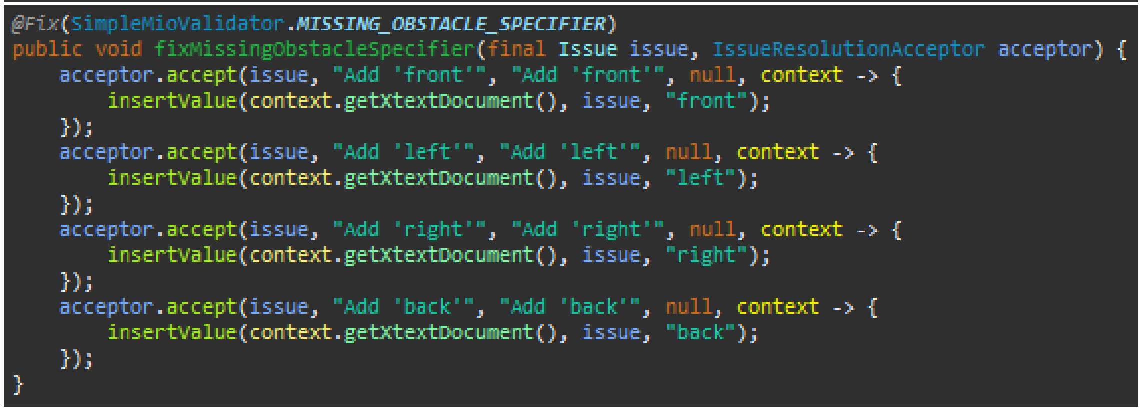
```
@Fix(SimpleMioValidator.INVALID_SOUND_SENSOR_SPECIFIER)
public void fixInvalidSoundSensorSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Remove specifier", "Remove specifier", null, context -> {
        updateValue(context.getXtextDocument(), issue, "");
    });
}
```



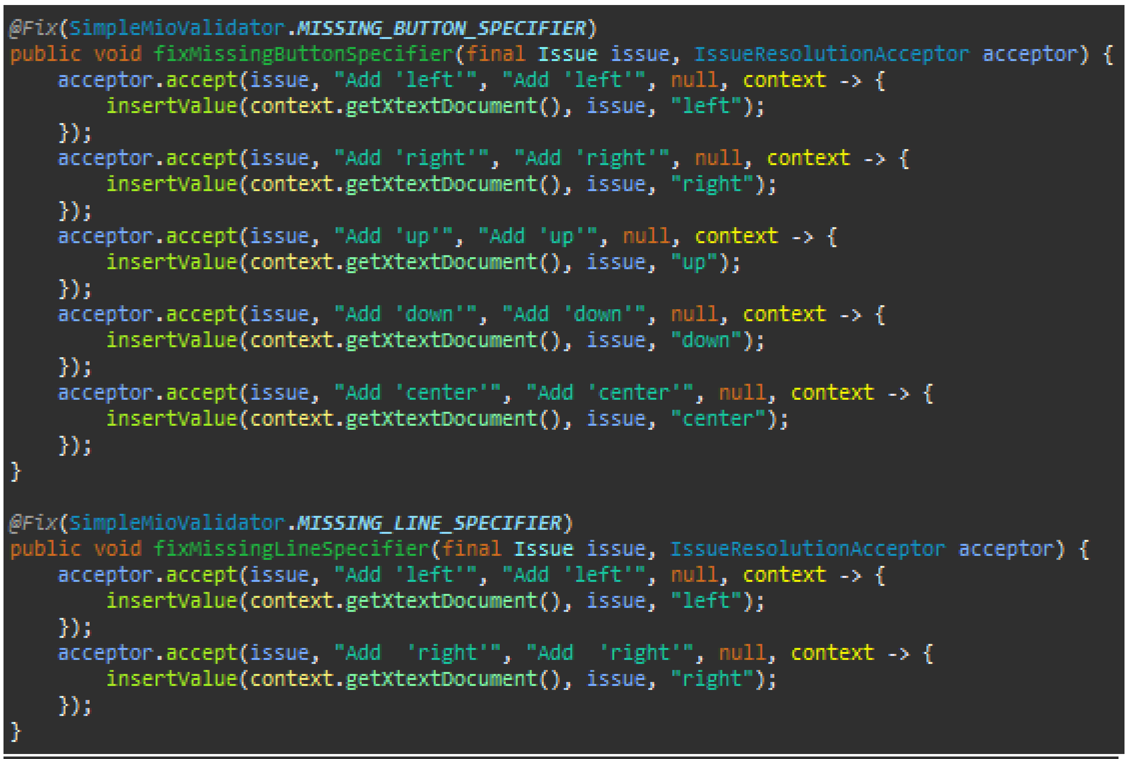
```java
@Fix(SimpleMioValidator.INVALID_MOVE_ACTION_SPECIFIER)
public void fixInvalidMoveActionSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Set to 'forward'", "Set to 'forward'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "forward");
    });
    acceptor.accept(issue, "Set to 'backward'", "Set to 'backward'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "backward");
    });
}

@Fix(SimpleMioValidator.INVALID_LED_ACTION_SPECIFIER)
public void fixInvalidLedActionSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Set to 'red'", "Set to 'red'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "red");
    });
    acceptor.accept(issue, "Set to 'green'", "Set to 'green'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "green");
    });
    acceptor.accept(issue, "Set to 'blue'", "Set to 'blue'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "blue");
    });
}

@Fix(SimpleMioValidator.INVALID_TURN_ACTION_SPECIFIER)
public void fixInvalidTurnActionSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Set to 'left'", "Set to 'left'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "left");
    });
    acceptor.accept(issue, "Set to 'right'", "Set to 'right'", null, context -> {
        updateValue(context.getXtextDocument(), issue, "right");
    });
}

@Fix(SimpleMioValidator.INVALID_INTENSITY)
public void fixInvalidIntensity(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Remove Intensity", "Remove Intensity", null, context -> {
        updateValue(context.getXtextDocument(), issue, "");
    });
}
```



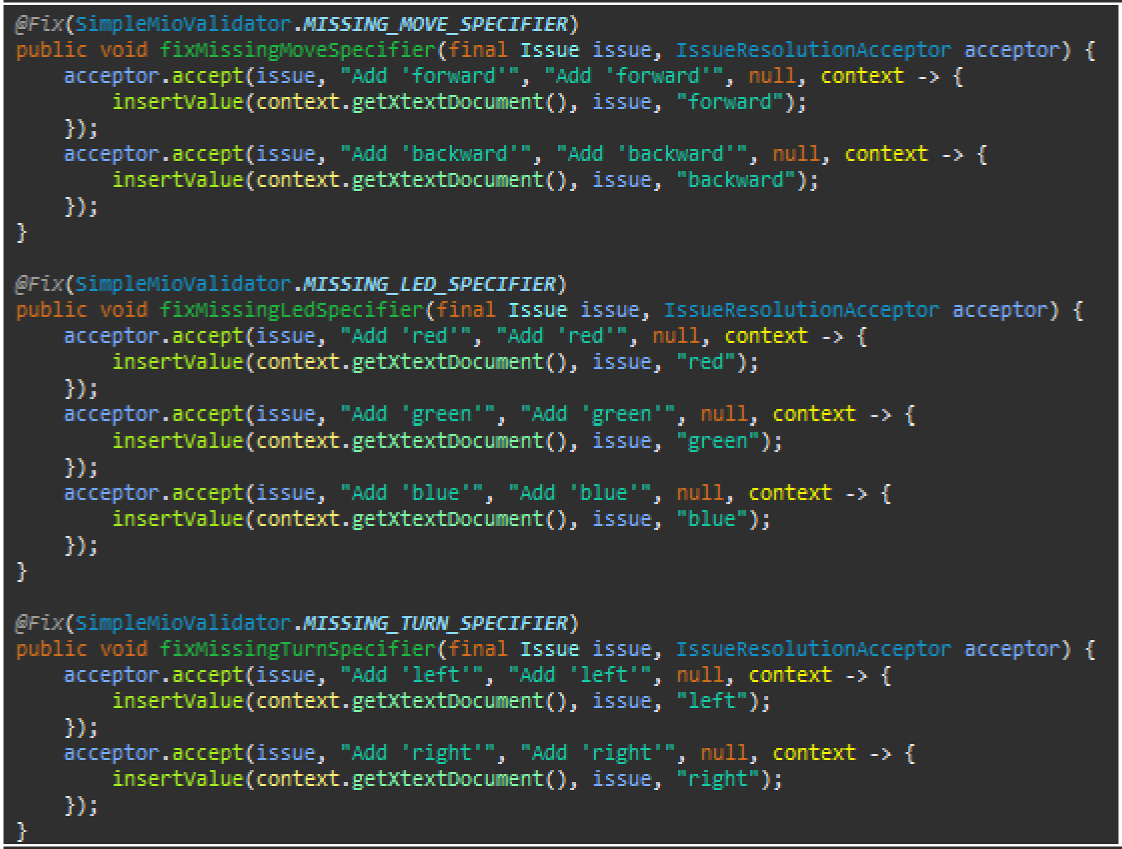
```java
@Fix(SimpleMioValidator.INVALID_VALUE_INTENSITY)
public void fixInvalidValueIntensity(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Set to minimum 0", "Set to minimum 0", null, context -> {
        updateValue(context.getXtextDocument(), issue, "0");
    });
    acceptor.accept(issue, "Set to intermediate 5", "Set to intermediate 5", null, context -> {
        updateValue(context.getXtextDocument(), issue, "5");
    });
    acceptor.accept(issue, "Set to maximum 10", "Set to maximum 10", null, context -> {
        updateValue(context.getXtextDocument(), issue, "10");
    });
}


protected void updateValue(IXtextDocument iXtextDocument, Issue issue, String newSpecifier) throws BadLocationException {
    int offset = issue.getOffset();
    int length = issue.getLength();
    iXtextDocument.replace(offset, length, newSpecifier);
}
```



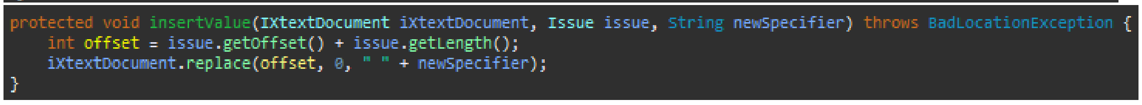
```java
@Fix(SimpleMioValidator.MISSING_OBSTACLE_SPECIFIER)
public void fixMissingObstacleSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Add 'front'", "Add 'front'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "front");
    });
    acceptor.accept(issue, "Add 'left'", "Add 'left'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "left");
    });
    acceptor.accept(issue, "Add 'right'", "Add 'right'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "right");
    });
    acceptor.accept(issue, "Add 'back'", "Add 'back'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "back");
    });
}
```



```java
@Fix(SimpleMioValidator.MISSING_BUTTON_SPECIFIER)
public void fixMissingButtonSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Add 'left'", "Add 'left'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "left");
    });
    acceptor.accept(issue, "Add 'right'", "Add 'right'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "right");
    });
    acceptor.accept(issue, "Add 'up'", "Add 'up'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "up");
    });
    acceptor.accept(issue, "Add 'down'", "Add 'down'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "down");
    });
    acceptor.accept(issue, "Add 'center'", "Add 'center'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "center");
    });
}

@Fix(SimpleMioValidator.MISSING_LINE_SPECIFIER)
public void fixMissingLineSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Add 'left'", "Add 'left'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "left");
    });
    acceptor.accept(issue, "Add  'right'", "Add  'right'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "right");
    });
}
```



```java
@Fix(SimpleMioValidator.MISSING_MOVE_SPECIFIER)
public void fixMissingMoveSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Add 'forward'", "Add 'forward'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "forward");
    });
    acceptor.accept(issue, "Add 'backward'", "Add 'backward'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "backward");
    });
}

@Fix(SimpleMioValidator.MISSING_LED_SPECIFIER)
public void fixMissingLedSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Add 'red'", "Add 'red'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "red");
    });
    acceptor.accept(issue, "Add 'green'", "Add 'green'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "green");
    });
    acceptor.accept(issue, "Add 'blue'", "Add 'blue'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "blue");
    });
}

@Fix(SimpleMioValidator.MISSING_TURN_SPECIFIER)
public void fixMissingTurnSpecifier(final Issue issue, IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue, "Add 'left'", "Add 'left'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "left");
    });
    acceptor.accept(issue, "Add 'right'", "Add 'right'", null, context -> {
        insertValue(context.getXtextDocument(), issue, "right");
    });
}
```



```java
protected void insertValue(IXtextDocument iXtextDocument, Issue issue, String newSpecifier) throws BadLocationException {
    int offset = issue.getOffset() + issue.getLength();
    iXtextDocument.replace(offset, 0, " " + newSpecifier);
}
```

Basically, if any error is detected the IDE will provide a valid solution to correct the program. When an intensity is wrongly inputted the IDE will provide 3 solutions, a minimum value (0), a medium value (5) and a maximum value (10)

Limitations:

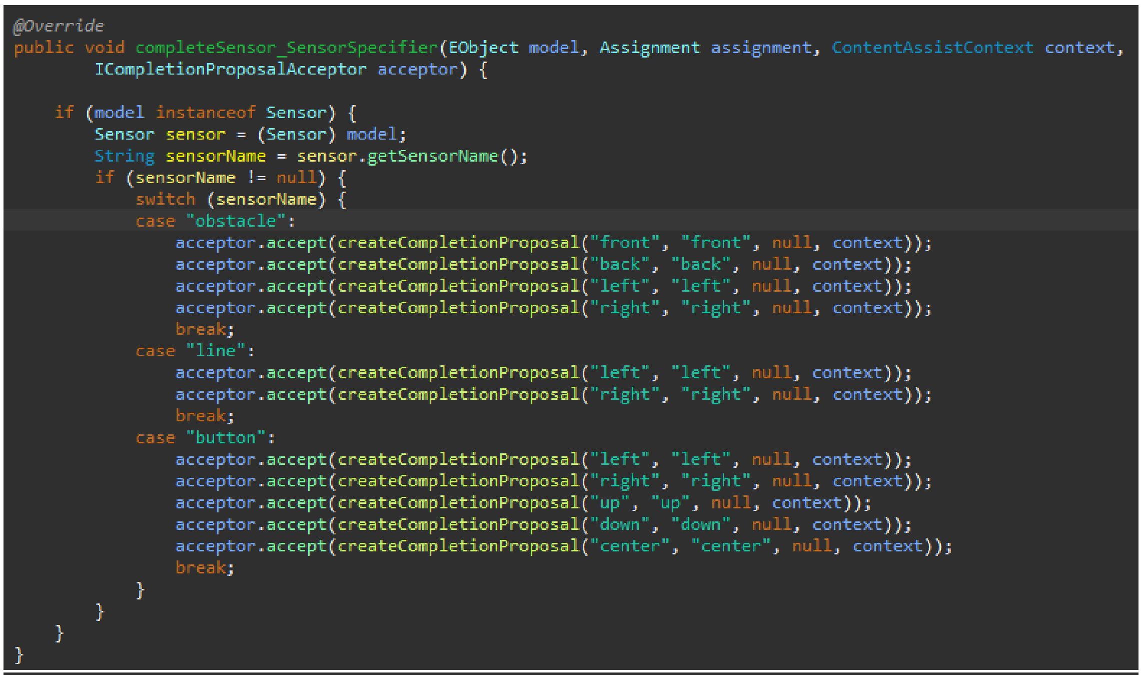
It was not possible do fix the error when we get a duplicate action, for example:

obstacle front ->turn left, move forward

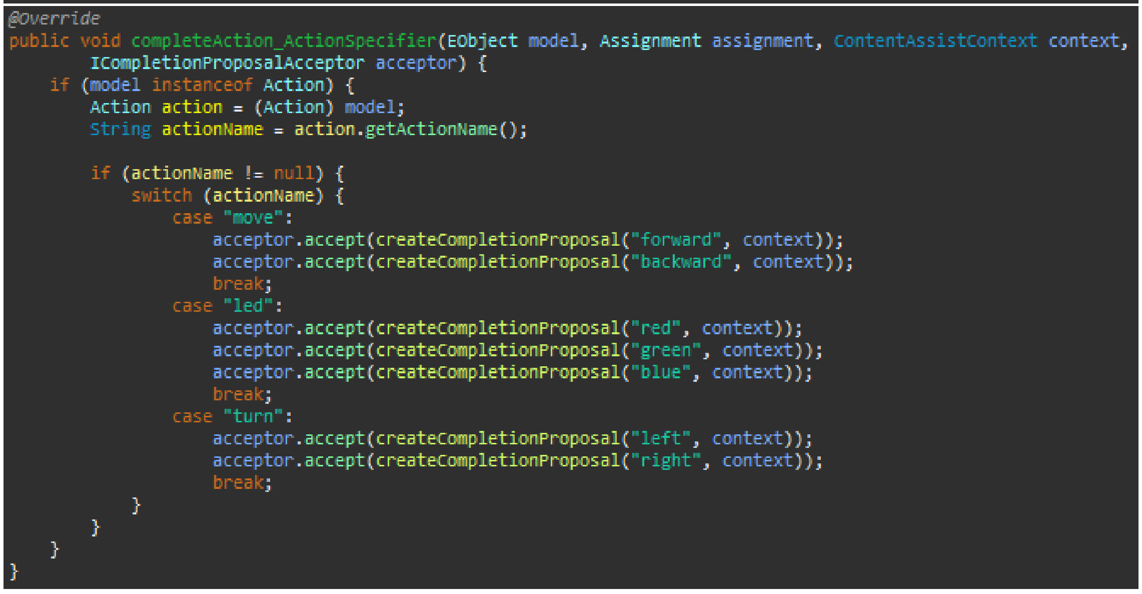
It was not possible to fix this error because only the move forward can be underlined as an error, it is not possible to underline the ",", so it is not possible to remove the duplicate action.

Auto-completion

For the auto-completion, we implemented the auto-completion for sensors and actions and this is what we implemented:



```
@Override
public void completeSensor_SensorSpecifier(EObject model, Assignment assignment, ContentAssistContext context,
        ICompletionProposalAcceptor acceptor) {

    if (model instanceof Sensor) {
        Sensor sensor = (Sensor) model;
        String sensorName = sensor.getSensorName();
        if (sensorName != null) {
            switch (sensorName) {
            case "obstacle":
                acceptor.accept(createCompletionProposal("front", "front", null, context));
                acceptor.accept(createCompletionProposal("back", "back", null, context));
                acceptor.accept(createCompletionProposal("left", "left", null, context));
                acceptor.accept(createCompletionProposal("right", "right", null, context));
                break;
            case "line":
                acceptor.accept(createCompletionProposal("left", "left", null, context));
                acceptor.accept(createCompletionProposal("right", "right", null, context));
                break;
            case "button":
                acceptor.accept(createCompletionProposal("left", "left", null, context));
                acceptor.accept(createCompletionProposal("right", "right", null, context));
                acceptor.accept(createCompletionProposal("up", "up", null, context));
                acceptor.accept(createCompletionProposal("down", "down", null, context));
                acceptor.accept(createCompletionProposal("center", "center", null, context));
                break;
            }
        }
    }
}
```

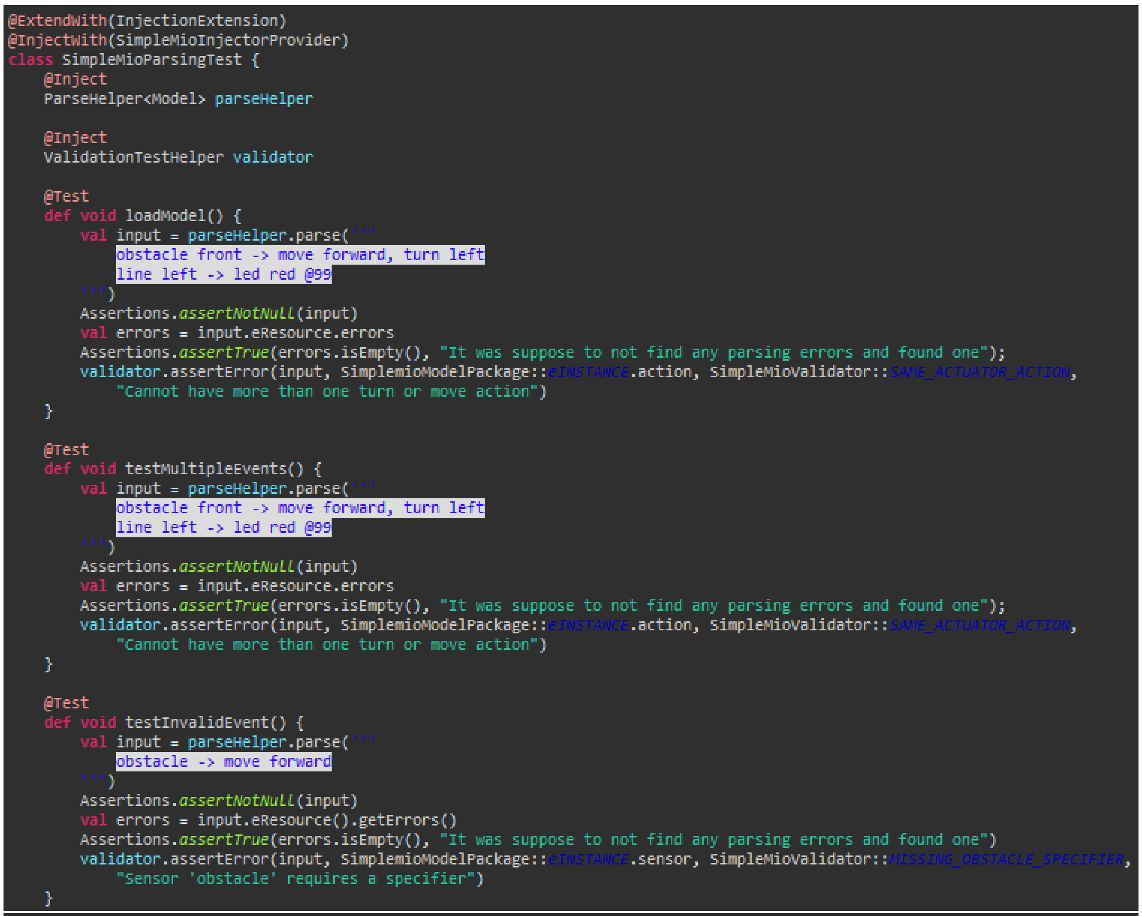


```
@Override
public void completeAction_ActionSpecifier(EObject model, Assignment assignment, ContentAssistContext context,
        ICompletionProposalAcceptor acceptor) {
    if (model instanceof Action) {
        Action action = (Action) model;
        String actionName = action.getActionName();

        if (actionName != null) {
            switch (actionName) {
                case "move":
                    acceptor.accept(createCompletionProposal("forward", context));
                    acceptor.accept(createCompletionProposal("backward", context));
                    break;
                case "led":
                    acceptor.accept(createCompletionProposal("red", context));
                    acceptor.accept(createCompletionProposal("green", context));
                    acceptor.accept(createCompletionProposal("blue", context));
                    break;
                case "turn":
                    acceptor.accept(createCompletionProposal("left", context));
                    acceptor.accept(createCompletionProposal("right", context));
                    break;
            }
        }
    }
}
```

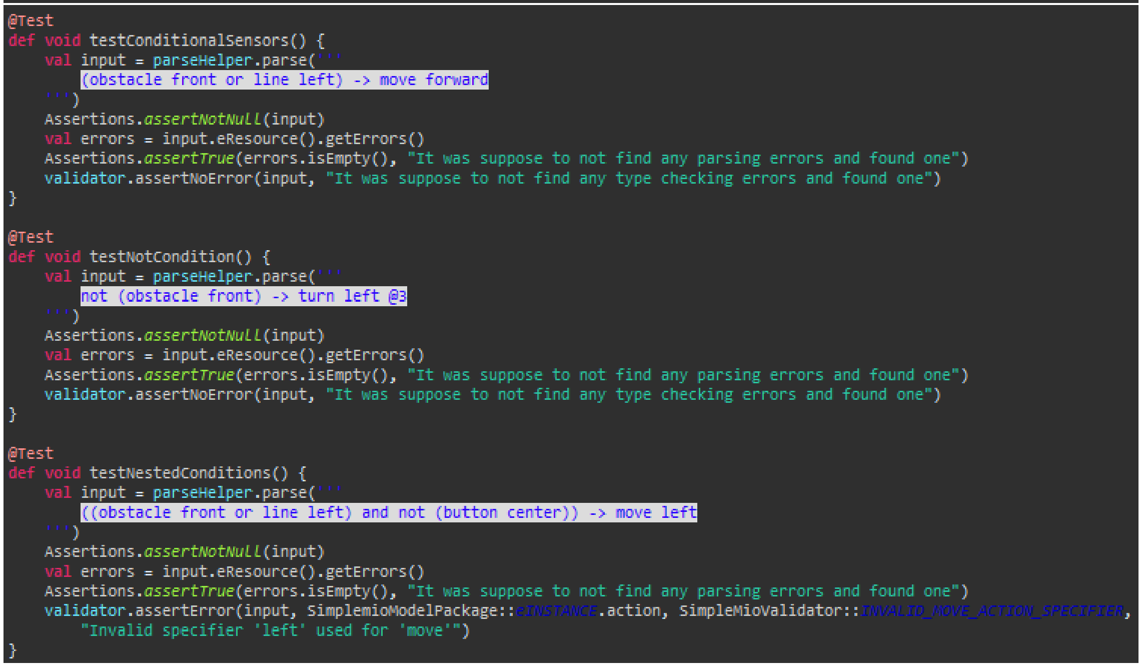
Basically, for each sensor or action that needs a specifier the IDE will suggest a valid specifier.

Tests

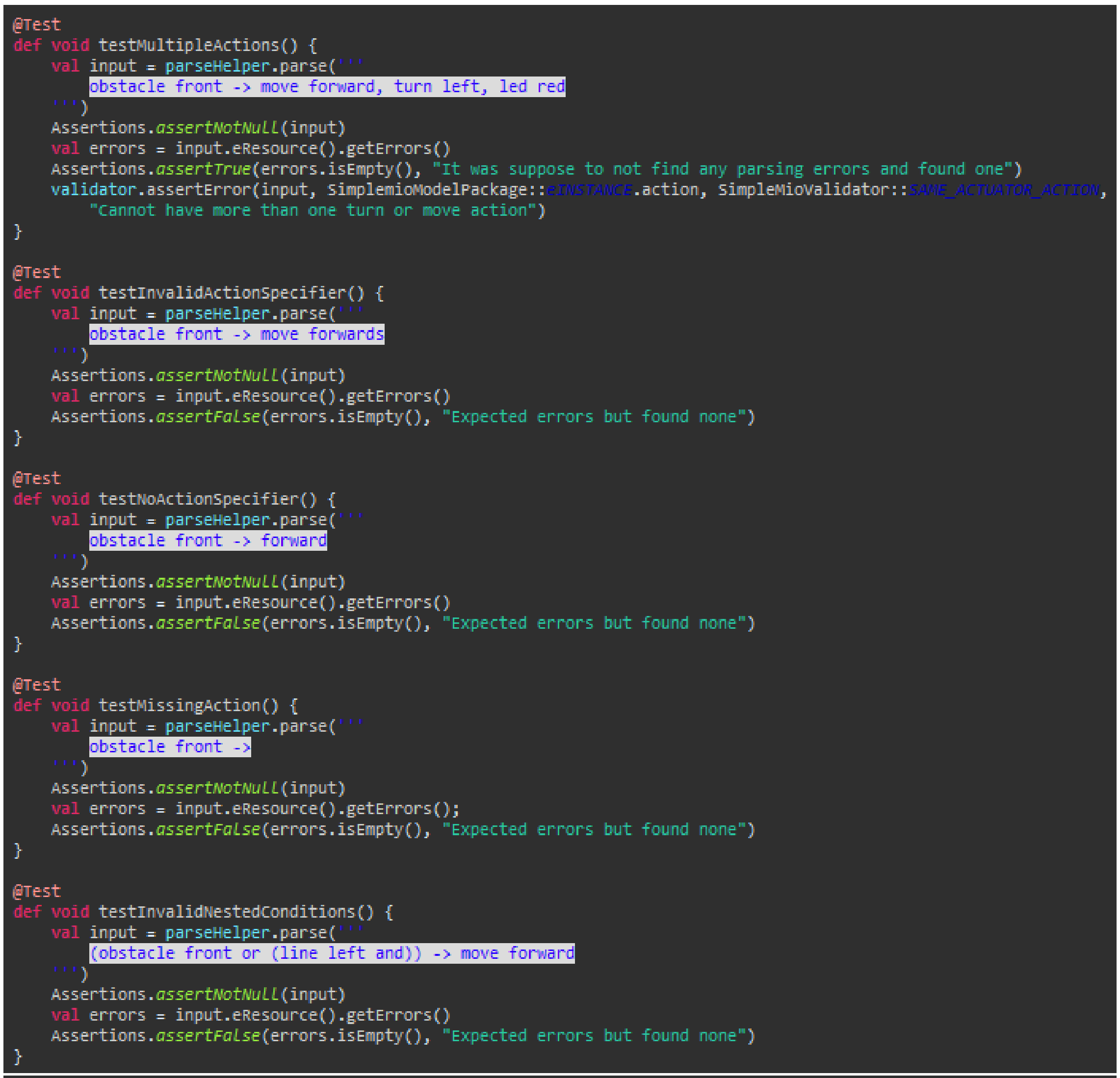
For testing we decided to use Junit tests.
First we decided to test the parser the tests that did pass the parser were used to test the validator.



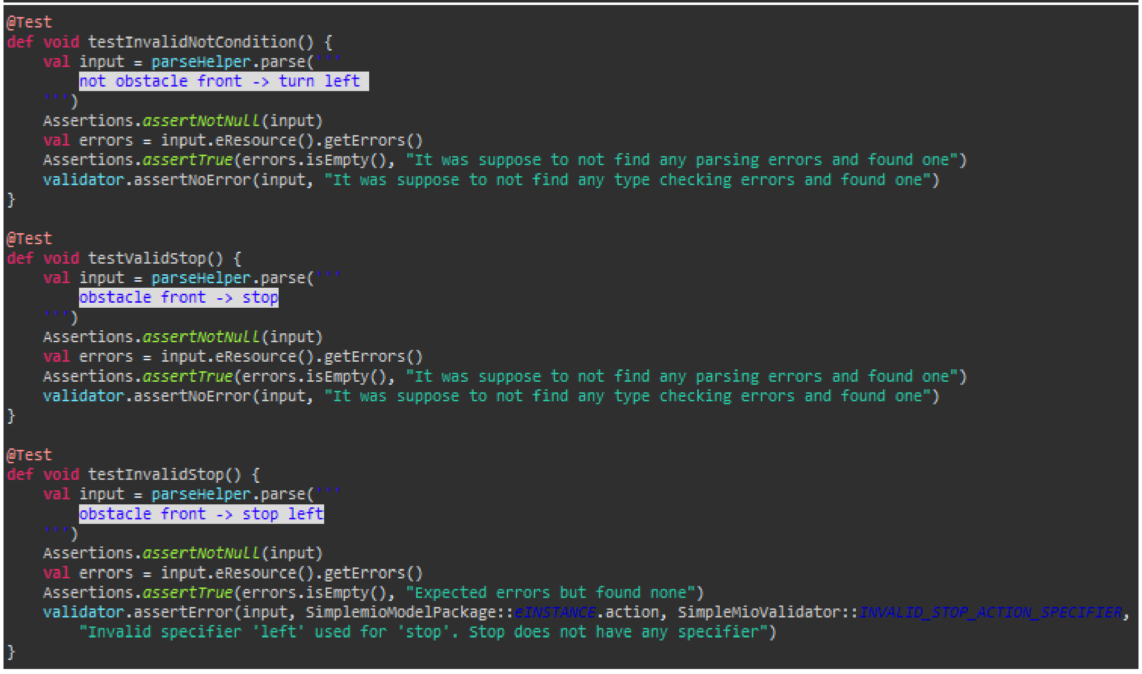
```
@ExtendWith(InjectionExtension)
@InjectWith(SimpleMioInjectorProvider)
class SimpleMioParsingTest {
    @Inject
    ParseHelper<Model> parseHelper

    @Inject
    ValidationTestHelper validator

    @Test
    def void loadModel() {
        val input = parseHelper.parse('''
            obstacle front -> move forward, turn left
            line left -> led red @99
        ''')
        Assertions.assertNotNull(input)
        val errors = input.eResource.errors
        Assertions.assertTrue(errors.isEmpty(), "It was suppose to not find any parsing errors and found one");
        validator.assertError(input, SimplemioModelPackage::eINSTANCE.action, SimpleMioValidator::SAME_ACTUATOR_ACTION,
            "Cannot have more than one turn or move action")
    }

    @Test
    def void testMultipleEvents() {
        val input = parseHelper.parse('''
            obstacle front -> move forward, turn left
            line left -> led red @99
        ''')
        Assertions.assertNotNull(input)
        val errors = input.eResource.errors
        Assertions.assertTrue(errors.isEmpty(), "It was suppose to not find any parsing errors and found one");
        validator.assertError(input, SimplemioModelPackage::eINSTANCE.action, SimpleMioValidator::SAME_ACTUATOR_ACTION,
            "Cannot have more than one turn or move action")
    }

    @Test
    def void testInvalidEvent() {
        val input = parseHelper.parse('''
            obstacle -> move forward
        ''')
        Assertions.assertNotNull(input)
        val errors = input.eResource().getErrors()
        Assertions.assertTrue(errors.isEmpty(), "It was suppose to not find any parsing errors and found one")
        validator.assertError(input, SimplemioModelPackage::eINSTANCE.sensor, SimpleMioValidator::MISSING_OBSTACLE_SPECIFIER,
            "Sensor 'obstacle' requires a specifier")
    }
```



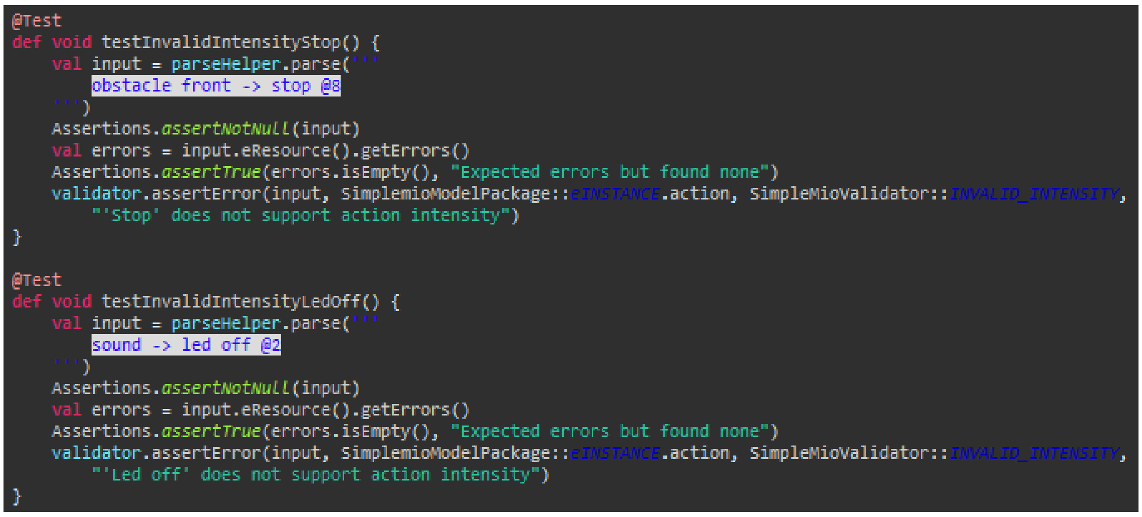
```
@Test
def void testConditionalSensors() {
    val input = parseHelper.parse('''
        (obstacle front or line left) -> move forward
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertTrue(errors.isEmpty(), "It was suppose to not find any parsing errors and found one")
    validator.assertNoError(input, "It was suppose to not find any type checking errors and found one")
}

@Test
def void testNotCondition() {
    val input = parseHelper.parse('''
        not (obstacle front) -> turn left @3
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertTrue(errors.isEmpty(), "It was suppose to not find any parsing errors and found one")
    validator.assertNoError(input, "It was suppose to not find any type checking errors and found one")
}

@Test
def void testNestedConditions() {
    val input = parseHelper.parse('''
        ((obstacle front or line left) and not (button center)) -> move left
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertTrue(errors.isEmpty(), "It was suppose to not find any parsing errors and found one")
    validator.assertError(input, SimplemioModelPackage::eINSTANCE.action, SimpleMioValidator::INVALID_MOVE_ACTION_SPECIFIER,
        "Invalid specifier 'left' used for 'move'")
}
```



```
@Test
def void testMultipleActions() {
    val input = parseHelper.parse('''
        obstacle front -> move forward, turn left, led red
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertTrue(errors.isEmpty(), "It was suppose to not find any parsing errors and found one")
    validator.assertError(input, SimplemioModelPackage::eINSTANCE.action, SimpleMioValidator::SAME_ACTUATOR_ACTION,
        "Cannot have more than one turn or move action")
}

@Test
def void testInvalidActionSpecifier() {
    val input = parseHelper.parse('''
        obstacle front -> move forwards
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertFalse(errors.isEmpty(), "Expected errors but found none")
}

@Test
def void testNoActionSpecifier() {
    val input = parseHelper.parse('''
        obstacle front -> forward
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertFalse(errors.isEmpty(), "Expected errors but found none")
}

@Test
def void testMissingAction() {
    val input = parseHelper.parse('''
        obstacle front ->
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors();
    Assertions.assertFalse(errors.isEmpty(), "Expected errors but found none")
}

@Test
def void testInvalidNestedConditions() {
    val input = parseHelper.parse('''
        (obstacle front or (line left and)) -> move forward
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertFalse(errors.isEmpty(), "Expected errors but found none")
}
```

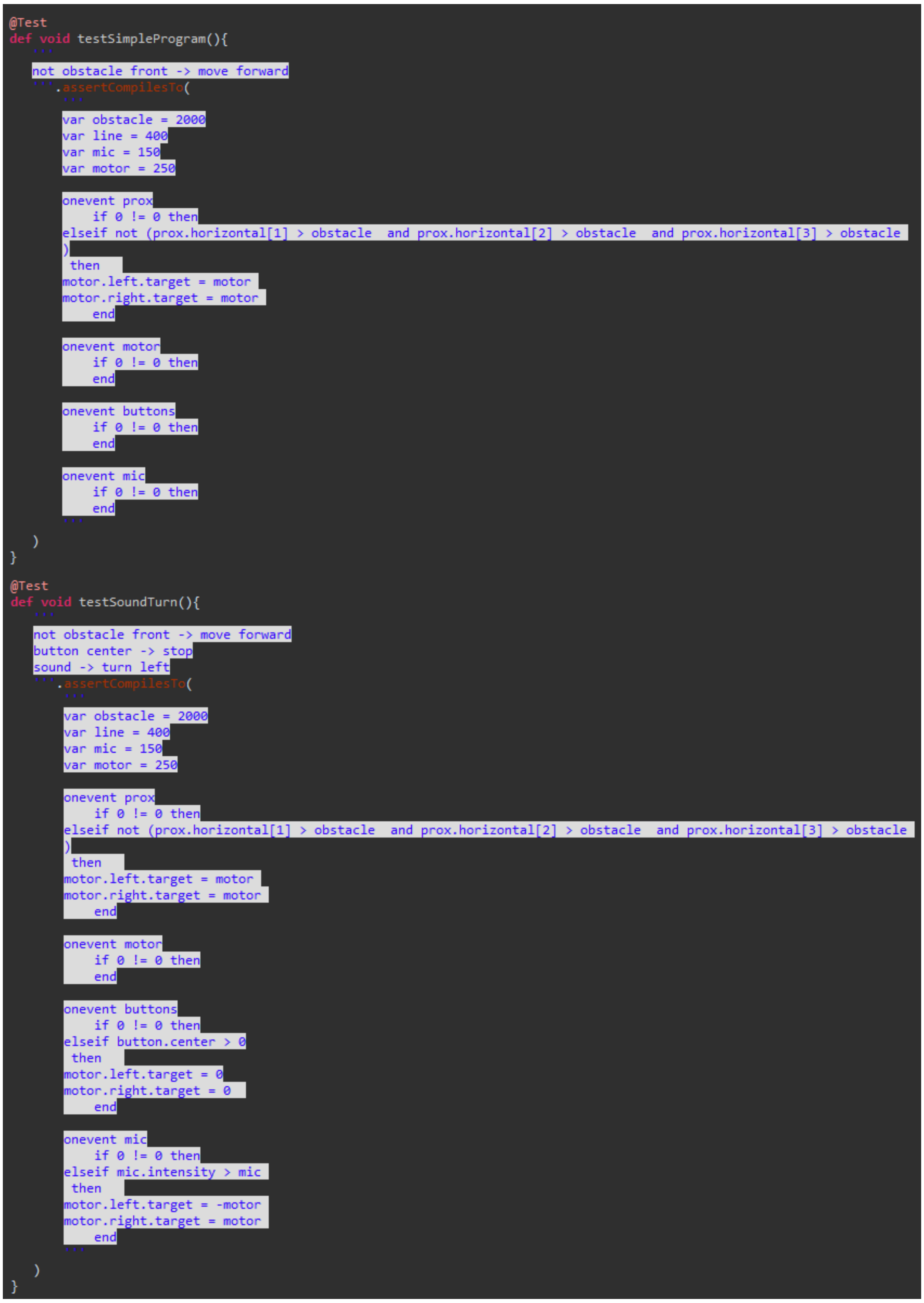


```
@Test
def void testInvalidNotCondition() {
    val input = parseHelper.parse('''
        not obstacle front -> turn left
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertTrue(errors.isEmpty(), "It was suppose to not find any parsing errors and found one")
    validator.assertNoError(input, "It was suppose to not find any type checking errors and found one")
}

@Test
def void testValidStop() {
    val input = parseHelper.parse('''
        obstacle front -> stop
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertTrue(errors.isEmpty(), "It was suppose to not find any parsing errors and found one")
    validator.assertNoError(input, "It was suppose to not find any type checking errors and found one")
}

@Test
def void testInvalidStop() {
    val input = parseHelper.parse('''
        obstacle front -> stop left
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertTrue(errors.isEmpty(), "Expected errors but found none")
    validator.assertError(input, SimplemioModelPackage::eINSTANCE.action, SimpleMioValidator::INVALID_STOP_ACTION_SPECIFIER,
        "Invalid specifier 'left' used for 'stop'. Stop does not have any specifier")
}
```

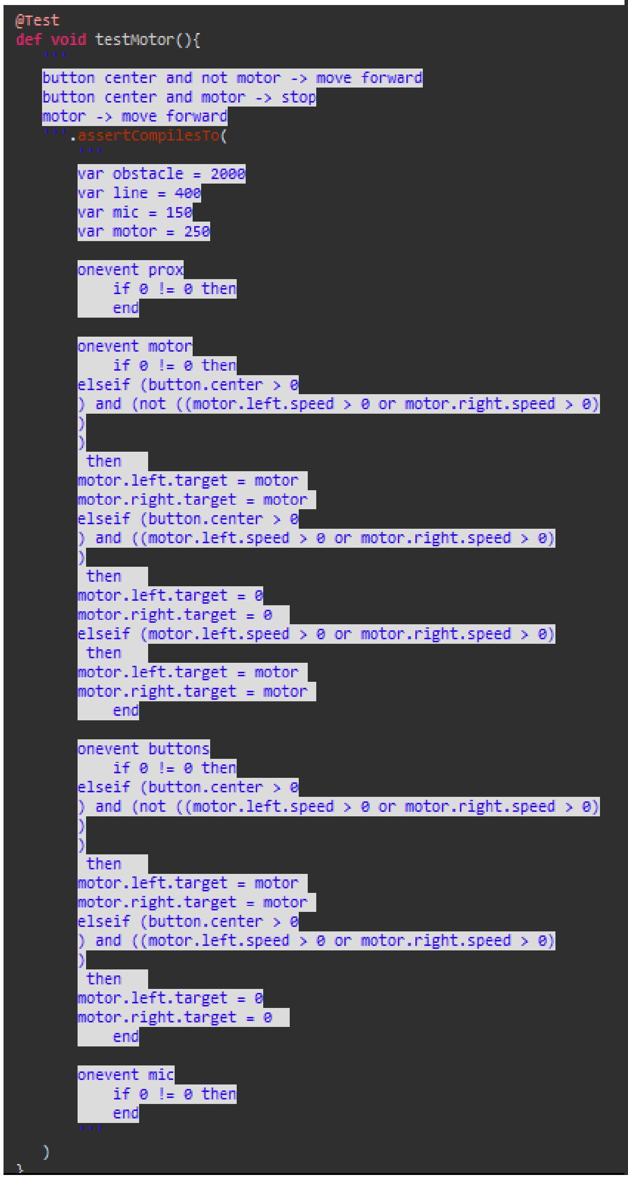


```
@Test
def void testInvalidIntensityStop() {
    val input = parseHelper.parse('''
        obstacle front -> stop @8
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertTrue(errors.isEmpty(), "Expected errors but found none")
    validator.assertError(input, SimplemioModelPackage::eINSTANCE.action, SimpleMioValidator::INVALID_INTENSITY,
        "'Stop' does not support action intensity")
}

@Test
def void testInvalidIntensityLedOff() {
    val input = parseHelper.parse('''
        sound -> led off @2
    ''')
    Assertions.assertNotNull(input)
    val errors = input.eResource().getErrors()
    Assertions.assertTrue(errors.isEmpty(), "Expected errors but found none")
    validator.assertError(input, SimplemioModelPackage::eINSTANCE.action, SimpleMioValidator::INVALID_INTENSITY,
        "'Led off' does not support action intensity")
}
```

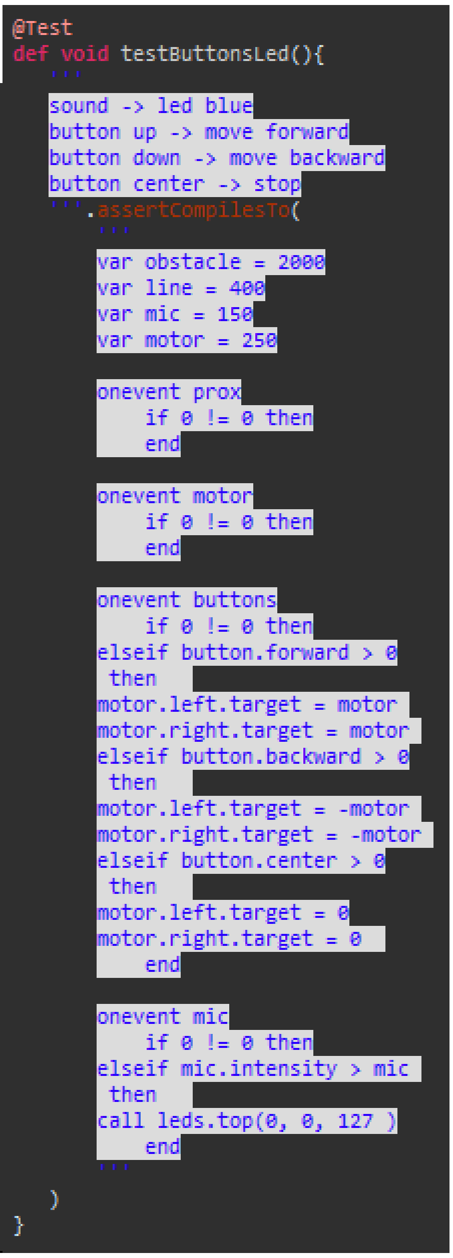
After the tests for the parser and the validator, the tests for the codegen were made:



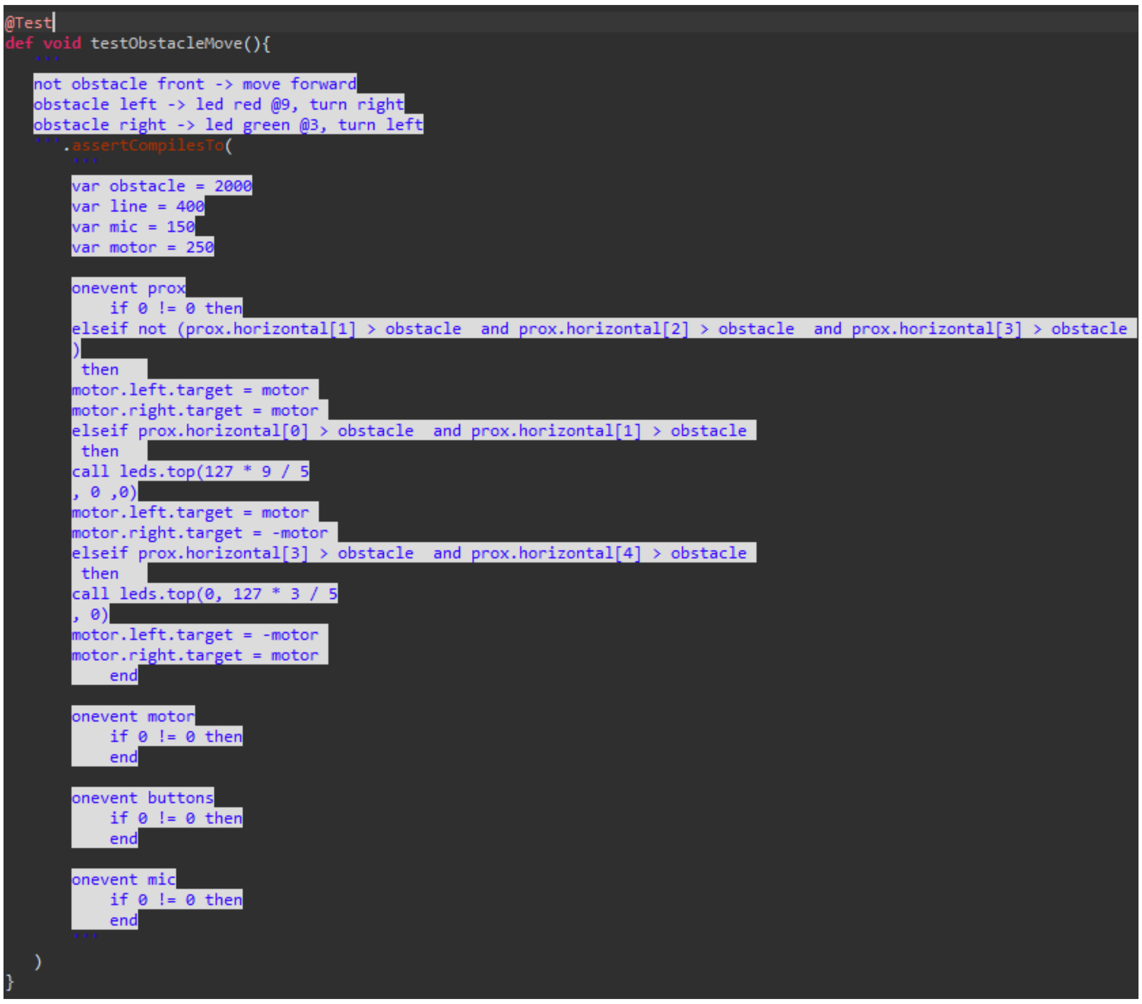
```
@Test
def void testSimpleProgram(){
    '''
    not obstacle front -> move forward
    '''.assertCompilesTo(
        '''
        var obstacle = 2000
        var line = 400
        var mic = 150
        var motor = 250

        onevent prox
            if 0 != 0 then
        elseif not (prox.horizontal[1] > obstacle  and prox.horizontal[2] > obstacle  and prox.horizontal[3] > obstacle
        )
         then
        motor.left.target = motor
        motor.right.target = motor
            end

        onevent motor
            if 0 != 0 then
            end

        onevent buttons
            if 0 != 0 then
            end

        onevent mic
            if 0 != 0 then
            end
        '''
    )
}

@Test
def void testSoundTurn(){
    '''
    not obstacle front -> move forward
    button center -> stop
    sound -> turn left
    '''.assertCompilesTo(
        '''
        var obstacle = 2000
        var line = 400
        var mic = 150
        var motor = 250

        onevent prox
            if 0 != 0 then
        elseif not (prox.horizontal[1] > obstacle  and prox.horizontal[2] > obstacle  and prox.horizontal[3] > obstacle
        )
         then
        motor.left.target = motor
        motor.right.target = motor
            end

        onevent motor
            if 0 != 0 then
            end

        onevent buttons
            if 0 != 0 then
        elseif button.center > 0
         then
        motor.left.target = 0
        motor.right.target = 0
            end

        onevent mic
            if 0 != 0 then
        elseif mic.intensity > mic
         then
        motor.left.target = -motor
        motor.right.target = motor
            end
        '''
    )
}
```



```
@Test
def void testMotor(){
    '''
    button center and not motor -> move forward
    button center and motor -> stop
    motor -> move forward
    '''.assertCompilesTo(
        '''
        var obstacle = 2000
        var line = 400
        var mic = 150
        var motor = 250

        onevent prox
            if 0 != 0 then
            end

        onevent motor
            if 0 != 0 then
        elseif (button.center > 0
        ) and (not ((motor.left.speed > 0 or motor.right.speed > 0)
        )
        )
         then
        motor.left.target = motor
        motor.right.target = motor
        elseif (button.center > 0
        ) and ((motor.left.speed > 0 or motor.right.speed > 0)
        )
         then
        motor.left.target = 0
        motor.right.target = 0
        elseif (motor.left.speed > 0 or motor.right.speed > 0)
         then
        motor.left.target = motor
        motor.right.target = motor
            end

        onevent buttons
            if 0 != 0 then
        elseif (button.center > 0
        ) and (not ((motor.left.speed > 0 or motor.right.speed > 0)
        )
        )
         then
        motor.left.target = motor
        motor.right.target = motor
        elseif (button.center > 0
        ) and ((motor.left.speed > 0 or motor.right.speed > 0)
        )
         then
        motor.left.target = 0
        motor.right.target = 0
            end

        onevent mic
            if 0 != 0 then
            end
        '''
    )
}
```



```
@Test
def void testButtonsLed(){
    '''
    sound -> led blue
    button up -> move forward
    button down -> move backward
    button center -> stop
    '''.assertCompilesTo(
        '''
        var obstacle = 2000
        var line = 400
        var mic = 150
        var motor = 250

        onevent prox
            if 0 != 0 then
            end

        onevent motor
            if 0 != 0 then
            end

        onevent buttons
            if 0 != 0 then
        elseif button.forward > 0
         then
        motor.left.target = motor
        motor.right.target = motor
        elseif button.backward > 0
         then
        motor.left.target = -motor
        motor.right.target = -motor
        elseif button.center > 0
         then
        motor.left.target = 0
        motor.right.target = 0
            end

        onevent mic
            if 0 != 0 then
        elseif mic.intensity > mic
         then
        call leds.top(0, 0, 127 )
            end
        '''
    )
}
```



```
@Test
def void testObstacleMove(){
    '''
    not obstacle front -> move forward
    obstacle left -> led red @9, turn right
    obstacle right -> led green @3, turn left
    '''.assertCompilesTo(
        '''
        var obstacle = 2000
        var line = 400
        var mic = 150
        var motor = 250

        onevent prox
            if 0 != 0 then
        elseif not (prox.horizontal[1] > obstacle  and prox.horizontal[2] > obstacle  and prox.horizontal[3] > obstacle
        )
         then
        motor.left.target = motor
        motor.right.target = motor
        elseif prox.horizontal[0] > obstacle  and prox.horizontal[1] > obstacle
         then
        call leds.top(127 * 9 / 5
        , 0 ,0)
        motor.left.target = motor
        motor.right.target = -motor
        elseif prox.horizontal[3] > obstacle  and prox.horizontal[4] > obstacle
         then
        call leds.top(0, 127 * 3 / 5
        , 0)
        motor.left.target = -motor
        motor.right.target = motor
            end

        onevent motor
            if 0 != 0 then
            end

        onevent buttons
            if 0 != 0 then
            end

        onevent mic
            if 0 != 0 then
            end
        '''
    )
}
```