SCIENCES SORBONNE UNIVERSITÉ

DevRep : Projet Webots

Rapport

Webots

robot simulation

Equipe :
Kimmeng Ly
Adam Latiri

Encadrants :
Tewfik Ziadi
Lom Hillah

2020 - 2021

Sommaire

1 Introduction

Dans le cadre de ce projet, nous allons étudier essentiellement la notion de **DSL**. DSL signifie Domain Specific Language. Autrement dit, il s'agit d'un ensemble restreint de vocabulaire et de règles de grammaire qui permet de répondre à un besoin précis de description ou d'organisation de l'information.

Dans notre cas, il s'agit de développer un DSL basé sur les **diagrammes d'Arkin** mais également, par la suite, développer un générateur de code pour faciliter le développement de petites applications robotique mobile dont **Webots**.

1.1 Les diagrammes d'Arkin

Les diagrammes d'Arkin sont le fruit d'une **démarche basée sur les comportements**. Un comportement est défini comme une activité primitive qui prend en entrée les données des capteurs et retourne une action à effectuer.
Le fait que chaque robot possède un ensemble de comportements, un système d'arbitrage est introduit afin d'intercepter les comportements et de sélectionner le comportement ayant la priorité la plus élevée.

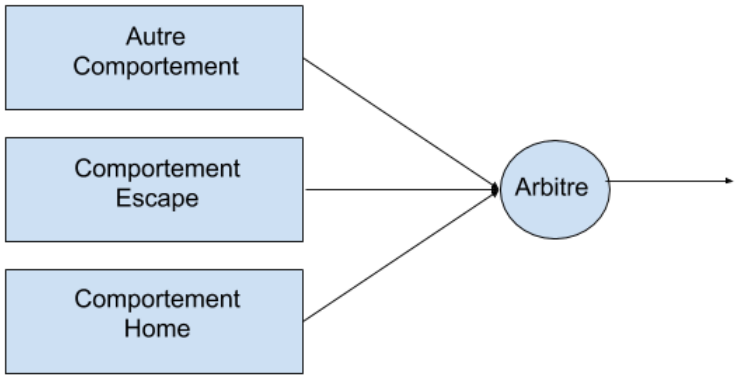

FIGURE 1 – Diagramme Arkin

1.2 Webots

Webots est un simulateur open-source. Il permet de tester des contrôleurs pour des robots évoluant dans un monde que l'on peut modeler (dans notre projet, nous utiliserons E-puck).
Le monde de Webots est en particulier capable de simuler la masse et la collision entre les objets.

2 Méta-modèle

2.1 Syntaxe abstraite

Pour la syntaxe abstraite, nous proposons le méta-modèle suivant :

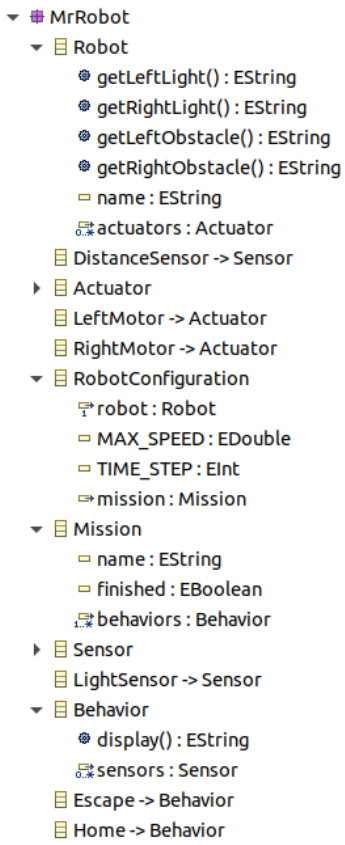

FIGURE 2 – Méta-modèle (syntaxe abstraite)

La classe *RobotConfiguration* est le point d'entrée de notre méta-modèle. Elle contient une instance de *Robot*, une instance de *Mission*, un attribut *MAX_SPEED* et un attribut *TIME_STEP*.
Un *Robot* contient un ensemble d'effecteurs(*Actuators*) et un ensemble de méthodes renvoyant la situation actuelle du robot, c'est-à-dire que s'il y a un obstacle ou une source de lumière sur un coté.
Une mission comporte un nom, *SearchForLight* dans notre cas, mais aussi un ensemble de comportements. Il existe deux comportements dans notre modèle, *Escape* qui consiste à éviter les obstacles et *Home* qui consiste à aller vers une source de lumière, notons que le comportement *Escape* est prioritaire par rapport au comportement *Home*.
Ensuite, nous avons les capteurs. Ils sont divisés en deux catégories, les capteurs de distance et les capteurs de luminosité.
Et enfin, nous avons les effecteurs qui correspondent au moteur du robot. Il y a un moteur droit et un moteur gauche pour chaque robot.

2.2 Une syntaxe concrète

Pour la syntaxe concrète, nous avons utilisé XText. Ci-dessous, une partie de notre syntaxe concrète :

```
RobotConfiguration returns RobotConfiguration:
    'RobotConfiguration'
    '{'
        ('MAX_SPEED' ':' MAX_SPEED=EDouble)?
        ('TIME_STEP' ':' TIME_STEP=EInt)?
        'robot' ':' robot=Robot
        ('mission' ':' mission=Mission)?
    '}';

Actuator returns Actuator:
    LeftMotor | RightMotor;

Behavior returns Behavior:
    Escape | Home;

Sensor returns Sensor:
    DistanceSensor | LightSensor;

Robot returns Robot:
    {Robot}
    'Robot'
    name=EString
    '{'
        ('actuators' ':' actuators+=Actuator ( "," actuators+=Actuator)*)?
    '}';

Mission returns Mission:
    (finished?='finished')?
    'Mission'
    name=EString
    '{'
        'behaviors' ':' behaviors+=Behavior ( "," behaviors+=Behavior)*
    '}';

LeftMotor returns LeftMotor:
    {LeftMotor}
    'LeftMotor'
    name=EString;

RightMotor returns RightMotor:
    {RightMotor}
    'RightMotor'
    name=EString;

Escape returns Escape:
    {Escape}
    'Escape'
    '{'
        ('sensors' ':' sensors+=Sensor ( "," sensors+=Sensor)* )?
    '}';
```

FIGURE 3 – XText(syntaxe concrète)

Concernant la syntaxe concrète sur *XText*, nous l'avons généré automatiquement à partir de notre méta-modèle *ecore*. Par conséquent, nous avons les mêmes classes et chaque classe possède les mêmes attributs.
Une fois générée, nous pouvons apporter des modifications afin de raffiner la syntaxe et donc rendre plus visible le code.
Pour finir, nous devons regénérer les artefacts afin de rendre notre nouveau langage utilisable.

3 Implémentation d'un générateur de code

3.1 Java Emitter Template

JET(Java Emitter Template) est un moteur de modèle générique qui peut être utilisé pour générer du code source SQL, XML, Java et d'autres langages à partir des modèles.
Nous allons donc appliquer une transformation de modèle crée vers du code java exécutable par notre robot sur Webots en utilisant JET.

3.2 Choix du langage et du robot

Pour le choix du langage d'implémentation du controller pour Webots, nous avons choisi java car il nous permet d'avoir une vision objet pour la génération de code.

Pour le choix du robot, nous avons choisi le robot E-puck car celui-ci est simple et convient parfaitement à l'objectif que nous voudrions atteindre.

3.3 Fonctionnement globale

Pour que le robot E-puck accomplisse sa mission, il a besoin de deux types de capteurs, les capteurs de distance et les capteurs de luminosité, qui sont décrit dans notre méta-modèle.

Au début, le générateur instancie la liste des capteurs, ainsi que ses roues. Ensuite, dans une boucle while permettant de contrôler le robot à chaque intervalle de temps, on récupère les valeurs lues par les capteurs (intensité de la lumière et distance par rapport aux obstacles), et selon le comportement souhaité pour le robot, changer en conséquence la direction dans laquelle il se déplace.

```
13 public class SearchForLight {
14
15     public static void main(String[] args) {
16
17         Robot robot = new Robot();
18
19         double MAX_SPEED = <%= config.getMAX_SPEED() %>;
20
21         int TIME_STEP = <%= config.getTIME_STEP()%>;
22         <%int size = config.getMission().getBehaviors().get(0).getSensors().size();%>
23         DistanceSensor[] ps = new DistanceSensor[<%=size%>];
24         String[] psNames = new String[<%=size%>];
25
26         LightSensor[] ls = new LightSensor[<%=size%>];
27         String[] lsNames = new String[<%=size%>];
28             <%for (Behavior b : config.getMission().getBehaviors()) {
29                 for(int i = 0; i< b.getSensors().size(); i++) {
30                     Sensor s = b.getSensors().get(i);
31                     if(s instanceof DistanceSensor) {%>
32                         psNames[<%=i%>] = "<%=s.getName()%>";<%}
33                     if(s instanceof LightSensor) {%>
34                         lsNames[<%=i%>] = "<%=s.getName()%>"; <%}}%>
35
36         for (int i = 0; i < 8; i++) {
37             ps[i] = robot.getDistanceSensor(psNames[i]);
38             ps[i].enable(TIME_STEP);
39
40             ls[i] = robot.getLightSensor(lsNames[i]);
41             ls[i].enable(TIME_STEP);
42         }
43         <%for (Actuator act : config.getRobot().getActuators()) {if(act instanceof LeftMotor) {%>
44         Motor leftMotor = robot.getMotor("<%=act.getName()%>");
45             <%}if(act instanceof RightMotor) {%>
46         Motor rightMotor = robot.getMotor("<%=act.getName()%>");<%}}%>
```

FIGURE 4 – JET : une portion de notre template

4 Tests de validation

4.1 Test de la syntaxe abstraite

Pour valider notre syntaxe abstraite, nous avons crée une instance dynamique de RobotConfiguration. A partir de cette instance nous avons ajouté un robot ainsi qu'une mission comme le montre la figure ci-dessous :

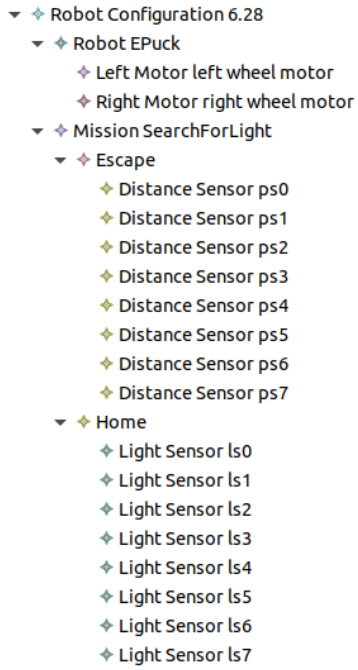

FIGURE 5 – Une instance dynamique de RobotConfiguration

4.2 Test de la syntaxe concrète

Pour valider notre syntaxe concrète, nous avons écrit, depuis une autre instance d'eclipse, un fichier *.mydslx* et notre langage est bien reconnu(les mots clés définit sont colorés).

```
RobotConfiguration {
    MAX_SPEED : 6.28
    TIME_STEP : 64

    robot :
        Robot 'Epuck' {
            actuators :
                LeftMotor 'left wheel motor',
                RightMotor 'right wheel motor'
        }
    mission :
        Mission 'SearchForLight' {
            behaviors :
                Escape {
                    sensors :
                        DistanceSensor 'ps0', DistanceSensor 'ps1', DistanceSensor 'ps2', DistanceSensor 'ps3',
                        DistanceSensor 'ps4', DistanceSensor 'ps5', DistanceSensor 'ps6', DistanceSensor 'ps7'
                },
                Home {
                    sensors :
                        LightSensor 'ls0', LightSensor 'ls1', LightSensor 'ls2', LightSensor 'ls3',
                        LightSensor 'ls4', LightSensor 'ls5', LightSensor 'ls6', LightSensor 'ls7'
```

FIGURE 6 – Utilisation de notre DSL

4.3 Test du générateur de code

A partir d'une méthode intermédiaire, nous avons chargé l'instance dynamique crée précédemment en une instance java et ainsi passer en paramètre à la méthode *generateJavaFile* ,qui permet de créer un fichier java, qui va utiliser notre template en utilisant l'instance fournie.

```java
public class MainClass {

    public static void generateJavaFile(RobotConfiguration config) {
        SearchForLight fileGen = new SearchForLight();
        FileWriter output;
        BufferedWriter writer;
        System.out.println("Creating Stage Java Webots File...");
        try {
            output = new FileWriter("SearchForLight.java");
            writer = new BufferedWriter(output);
            // Appel de la méthode generate de la classe générée par JET
            writer.write(fileGen.generate(config));
            writer.close();
            System.out.println("Done.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Path configPath = Paths.get("/home/kimmeng/eclipse-devrep/Webots/model/RobotConfiguration.xmi");
        RobotConfiguration config = (RobotConfiguration) ModelIO.loadModel(configPath);
        generateJavaFile(config);
    }
}
```

La classe template va ensuite renvoyer ce code ci-dessous qui sera écrit dans un fichier java.

```java
/* Welcome to the search for light mission. */

import com.cyberbotics.webots.controller.Robot;

public class SearchForLight {

    public static void main(String[] args) {

        Robot robot = new Robot();

        double MAX_SPEED = 6.28;

        int TIME_STEP = 64;

        DistanceSensor[] ps = new DistanceSensor[8];
        String[] psNames = new String[8];

        LightSensor[] ls = new LightSensor[8];
        String[] lsNames = new String[8];

        psNames[0] = "ps0";
        psNames[1] = "ps1";
        psNames[2] = "ps2";
        psNames[3] = "ps3";
        psNames[4] = "ps4";
        psNames[5] = "ps5";
        psNames[6] = "ps6";
        psNames[7] = "ps7";
        lsNames[0] = "ls0";
        lsNames[1] = "ls1";
        lsNames[2] = "ls2";
        lsNames[3] = "ls3";
        lsNames[4] = "ls4";
        lsNames[5] = "ls5";
        lsNames[6] = "ls6";
        lsNames[7] = "ls7";

        for (int i = 0; i < 8; i++) {
            ps[i] = robot.getDistanceSensor(psNames[i]);
```

FIGURE 7 – Portion de code généré

4.4 Test du code produit sur Webots avec le robot E-puck

Enfin pour valider le tout, nous avons testé le code généré sur Webots. Pour ce faire, nous avons changé le code du controlleur du robot E-puck.
Et comme nous pouvons le voir, le robot poursuit bien une source de lumière tout en évitant des obstacles si ceux-ci se présentent.

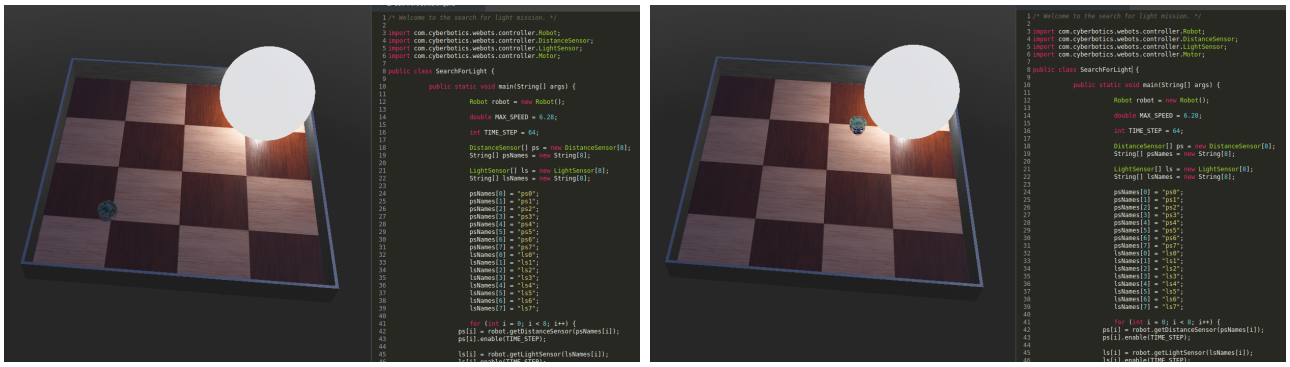

FIGURE 8 – Simulation du robot E-puck avec le code généré

5 Conclusion

En conclusion, nous avons mis au point les fonctionnalités permettant à un robot de suivre une source de lumière, d'éviter les obstacles, et d'adapter sa stratégie en fonction des valeurs lues par ses capteurs. Un utilisateur peut s'en servir avec le générateur de code pour créer un contrôleur fonctionnel, testable sur Webots.

Références

[1] Eclipse Modeling Framework (EMF),
`https://www.eclipse.org/modeling/emf/`

[2] Webots,
`https://cyberbotics.com/`

[3] JET Tutorial Part 1 (Introduction to JET),
`https://www.eclipse.org/articles/Article-JET/jet_tutorial1.html`