Projet Arduino

ÉQUIPE PANCAKE
Laurie FERNANDEZ – Emma GLESSER – Arthur SOENS – Vincent TUREL

17/12/2022

Lien de notre solution

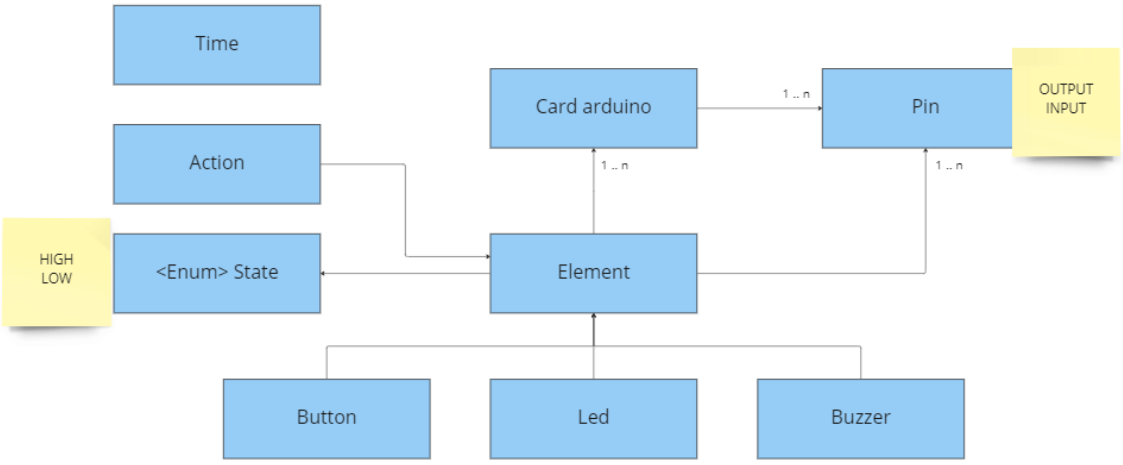
Vous pouvez retrouver notre solution sur le repository Github suivant :
https://github.com/Emma-Glesser/DSL_Team_Pancake

Langages de développement choisis

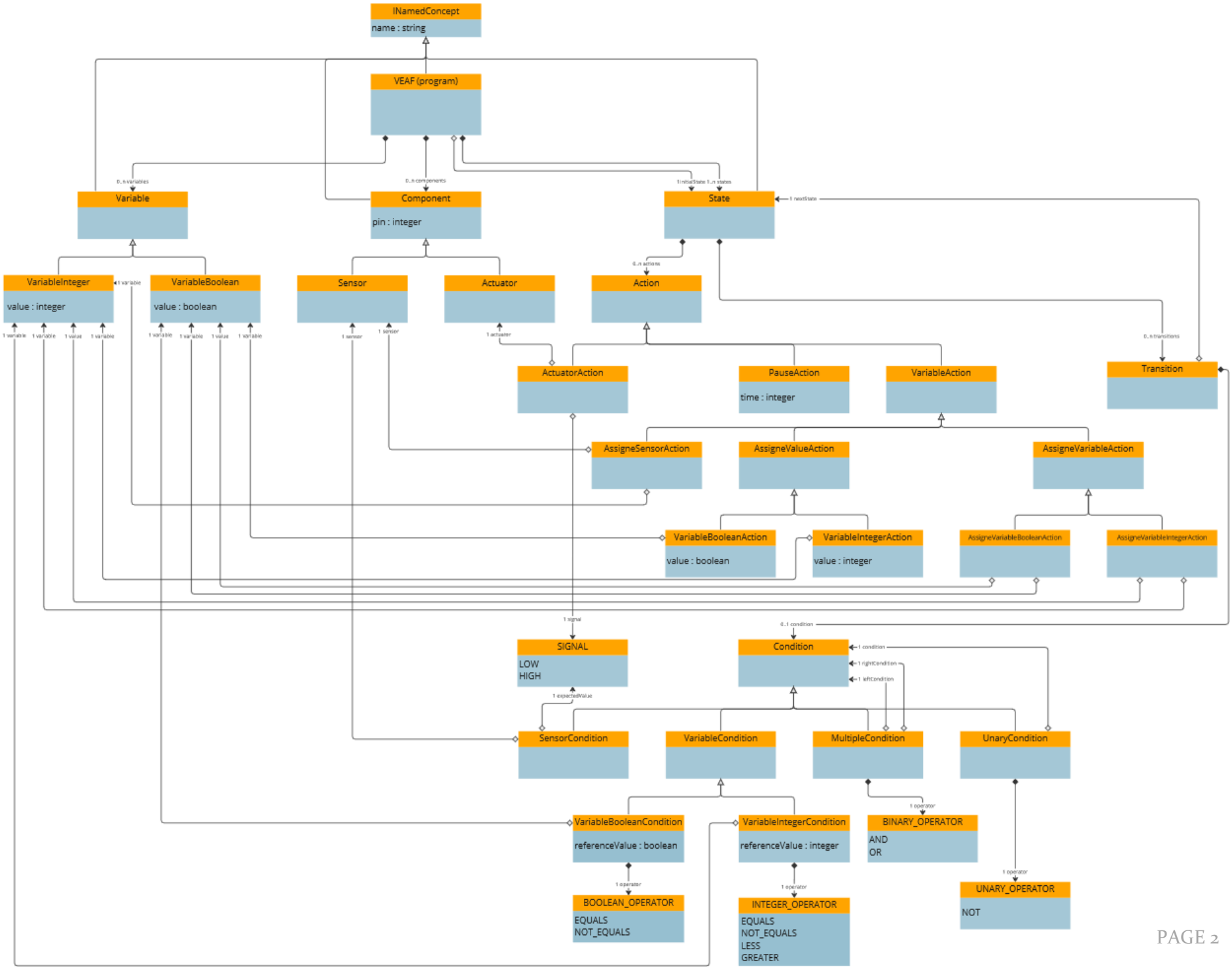
Nos choix de technologies pour les différents DSL se sont portés sur Java avec Groovy pour notre DSL interne et sur MPS pour notre DSL externe. Le DSL externe étant le plus complexe à développer, il a été le premier sur lequel nous nous sommes penchés.

Domain model de notre solution

Notre vision de la solution à développer a bien évolué avec le temps.



On peut noter une complexification certaine de notre domain model entre la phase de conception de celui-ci et sa dernière version. Cela est dû au fait que notre vision au début du projet ne prenait pas en compte l'ensemble des contraintes de notre code et des extensions à développer. De plus, ce premier domain model ne permettait pas de rendre visible notre gestion des cas particuliers comme les passages de condition, ce point a été tout particulièrement discuté par l'équipe.


INamedConcept
name : string
VEAF (program)
Variable
Component
pin : integer
State
VariableInteger
value : integer
VariableBoolean
value : boolean
Sensor
Actuator
Action
ActuatorAction
PauseAction
time : integer
VariableAction
Transition
AssigneSensorAction
AssigneValueAction
AssigneVariableAction
VariableBooleanAction
value : boolean
VariableIntegerAction
value : integer
AssigneVariableBooleanAction
AssigneVariableIntegerAction
SIGNAL
LOW
HIGH
Condition
SensorCondition
VariableCondition
MultipleCondition
UnaryCondition
VariableBooleanCondition
referenceValue : boolean
VariableIntegerCondition
referenceValue : integer
BINARY_OPERATOR
AND
OR
UNARY_OPERATOR
NOT
BOOLEAN_OPERATOR
EQUALS
NOT_EQUALS
INTEGER_OPERATOR
EQUALS
NOT_EQUALS
LESS
GREATER

Ainsi notre version finale du domain model est celle visible ici. En plus des spécifications des différents types de variables booléennes et entières et des conditions qui y sont associées que l'on peut retrouver en différents points de l'architecture, des opérateurs applicables à nos conditions ont été définis. Des distinctions ont également été faites entre nos capteurs ou senseurs et actionneurs ou actuateurs. Les capteurs prennent compte de phénomènes physiques sur la machine et les transforment en signal d'entrée sur la carte Arduino comme l'appui d'un bouton pressoir ou le mouvement ; les actionneurs transforment cette information du signal reçue en action d'activation comme un allumage de led ou le déclenchement du son d'un buzzer.

Le but de notre DSL est de simplifier l'écriture de code pour une personne experte dans le domaine de l'électronique embarquée sur Arduino mais novice en termes de développement logiciel. Pour cela, nous avons essayé de nous rapprocher de la description que ferait une personne de son système. Un système est souvent décrit à l'aide d'état et d'actions ou conditions qui entrainent un changement d'état. Nous avons choisi de reprendre cette vision en implémentant des états où des actions peuvent être effectuées et des transitions entre états. Une personne pourra alors décrire son système et la manière de passer d'un état à un autre.

De plus, afin d'éviter la génération d'un code Arduino incorrect par une personne novice, nous avons fait le choix de passer certaines contraintes à un niveau structurel. De cette manière, nous évitons les défauts d'une validation au moment de la génération du code .ino car la personne ne pourra pas provoquer ces erreurs. Par exemple, nous avons séparé le concept de capteur de celui d'actionneur afin d'éviter le cas où un utilisateur tenterait de lire sur une led ou d'allumer un bouton.

Un autre exemple est celui où la personne est obligée de renseigner l'état initial au programme plutôt que de décrire si un état est initial ou non. De cette manière le programme aura toujours un unique état initial. Cela permettra d'éviter les confusions lors de l'écriture du programme par un utilisateur et de nous éviter l'écriture de validateurs complexes susceptibles de contenir des erreurs.

Nous aurions pu faire un DSL ayant une syntaxe beaucoup plus compacte mais nous avons fait le choix d'avoir une syntaxe légèrement verbeuse afin de structurer le code. Cela amène selon nous plus de rigueur et aide à la compréhension. Pour assister cette rigueur, nous avons également rajouté des vérifications à la compilation permettant d'éviter dans un premier temps de créer un script illisible et dans un deuxième temps de générer un programme Arduino incorrect.

Par exemple si un programme débute ainsi :

```
program _: "Scenario4" being {
    variables {
        variable name: "state" value value: 0
        variable name: "if" value value: 0
        variable name: "oldState" value value: 0
    }
```

La compilation en .ino va provoquer l'erreur suivante :

```
vincent@LAPTOP-VINCENT:/mnt/d/Documents/1_Cours/5A/DSL/DSL_Team_Pancake/Internal_DSL$ ./run.sh scripts/Scenario4.groovy
Exception in thread "main" java.lang.RuntimeException: Variable name can't be set as a C++ keyword
        at kernel.App.check(App.java:70)
        at dsl.Arduino_DSL_Model.generateCode(Arduino_DSL_Model.java:26)
```

Également, un programme structuré comme cela :

```
program _: "Scenario4" being {
    variables {
        variable name: "state" value value: 0
    }

    components {
        sensor name: "button" pin pin: 10
    }

    variables {
        variable name: "oldState" value value: 0
    }
```

Provoquera l'erreur suivante :

```
vincent@LAPTOP-VINCENT:/mnt/d/Documents/1_Cours/5A/DSL/DSL_Team_Pancake/Internal_DSL$ ./run.sh scripts/Scenario4.groovy
Exception in thread "main" java.lang.RuntimeException: Variables can only be defined once
```

Il existe beaucoup d'autres vérifications mais même s'il en manque certainement encore cela nous permet d'illustrer notre volonté de cadrer l'utilisation de notre DSL pour aider l'utilisateur.

Enfin, nous avons fait le choix de ne pas utiliser certaines techniques de Groovy qui empêchent l'analyse statique par défaut de Groovy de fonctionner. Ainsi, notre DSL possède de l'auto-complétions avec un IDE et de l'analyse statique qui met en évidence certaines erreurs présentes. C'est pour cela que le programme ci-dessous est coloré. On peut alors immédiatement voir que le mot clé ***actuator*** ligne 15 n'est pas reconnu et qu'il s'agit d'une erreur.

```
1  import static dsl.Arduino_DSL.*
2
3  program _: "Scenario1" being {
4      components {
5          sensor name: "button" pin pin: 10
6          actuator name: "led" pin pin: 11
7          actuator name: "buzzer" pin pin: 12
8      }
9
10     initialState name: "initial"
11
12     states {
13         state name: "initial" being {
14
15             actudator "led" becomes SIGNAL.LOW
16             actuator name: "buzzer" becomes SIGNAL.LOW
17
18             next_state_is stateName: "state1" when { EQUALS left: "button", SIGNAL.HIGH }
19         }
20
21         state name: "state1" being {
22             actuator name: "led" becomes SIGNAL.HIGH
23             actuator name: "buzzer" becomes SIGNAL.HIGH
24
25             next_state_is stateName: "initial" when { EQUALS left: "button", SIGNAL.LOW }
26         }
27     }
28 }
```

Syntaxe concrète de notre DSL sous format de formulaire

SYNTAXE DU DSL EXTERNE

```
VEAF :
    'Program' String ':\n'
    '\tinitial state is ' String '\n'
    '\n'
    '\tvariables : {\n'
    (Variable '\n')*
    '\t}\n'
    '\n'
    '\tcomponents: {\n'
    (Component'\n')*
    '\t}\n'
    '\n'
    '\tstates: {\n'
    State ('\n', State)*
    '\t}\n'

Variable :
    VariableInteger | VariableBoolean

VariableInteger :
    '\t\tinteger' String '->' Integer

VariableBoolean:
    '\t\tboolean' String '->' Boolean

Component :
    Sensor | Actuator

Sensor :
    '\t\tsensor' String 'on pin' Integer

Actuator :
    '\t\tactuator' String 'on pin' Integer

State :
    '\t\tstate' String ':\n'
    (Action,'\n')*
    (Transition,'\n')*

Action :
    ActuatorAction | PauseAction | VariableAction

ActuatorAction :
    '\t\t\t' String '->' SIGNAL

SIGNAL :
  'LOW' | 'HIGH'

PauseAction :
    '\t\t\tpause' Integer 'milliseconds'

VariableAction :
    AssigneSensorAction | AssigneValueAction | AssigneVariableAction
```

```
AssigneSensorAction :
    '\t\t\t' String '=>' String

AssigneValueAction :
    VariableIntegerAction | VariableBooleanAction

VariableIntegerAction :
    '\t\t\t' String '->' Integer

VariableBooleanAction:
    '\t\t\t' String '->' Boolean

AssigneVariableAction :
    AssigneVariableIntegerAction | AssigneVariableBooleanAction

AssigneVariableIntegerAction :
    '\t\t\t' String '=>' String

AssigneVariableBooleanAction:
    '\t\t\t' String '=>' String

Transition :
    '\t\t\tnext state is' String ('when' Condition)?

Condition :
    SensorCondition | VariableCondition | MultipleCondition | UnaryCondition

SensorCondition :
    String 'is' SIGNAL

VariableCondition :
    VariableIntegerCondition | VariableBooleanCondition

VariableIntegerCondition :
    String INTEGER_OPERATOR Integer

VariableBooleanCondition :
    String BOOLEAN_OPERATOR Boolean

VariableBooleanCondition :
    String BOOLEAN_OPERATOR Boolean

MultipleCondition :
    '(' Condition BINARY_OPERATOR Condition')'

UnaryCondition :
    '(' UNARY_OPERATOR Condition ')'

INTEGER_OPERATOR :
  'EQUALS' | 'NOT_EQUALS' | 'LESS' | 'GREATER'

BOOLEAN_OPERATOR :
  'EQUALS' | 'NOT_EQUALS'

BINARY_OPERATOR :
  'AND' | 'OR'

UNARY_OPERATOR :
  'NOT'
```

SYNTAXE DU DSL INTERNE

```
VEAF :
    'program' String 'being {\n'
    '\tvariables : {\n'
    (Variable '\n')*
    '\t}\n'
    '\n'
    '\tcomponents: {\n'
    (Component'\n')*
    '\t}\n'
    '\n'
    '\tinitialState ' String '\n'
    '\n'
    '\tstates: {\n'
    State ('\n', State)*
    '\t}\n'

Variable :
    VariableInteger | VariableBoolean

VariableInteger :
    '\t\variable' String 'value' Integer

VariableBoolean:
    '\t\variable' String 'value' Boolean

Component :
    Sensor | Actuator

Sensor :
    '\t\tsensor ' String ' pin ' Integer

Actuator :
    '\t\tactuator ' String ' pin ' Integer

State :
    '\t\tstate ' String ' being {\n'
    (Action,'\n')*
    (Transition,'\n')*

Action :
    ActuatorAction | PauseAction | VariableAction

ActuatorAction :
    '\t\t\tactuator ' String 'becomes' SIGNAL

SIGNAL :
  'SIGNAL.LOW' | 'SIGNAL.HIGH'

PauseAction :
    '\t\t\tpause' Integer ('milliseconds')?

VariableAction :
    AssigneSensorAction | AssigneValueAction | AssigneVariableAction

AssigneSensorAction :
    '\t\t\tvariable ' String 'becomes' String

AssigneValueAction :
    VariableIntegerAction | VariableBooleanAction
```

```
VariableIntegerAction :
    '\t\t\tvariable ' String 'becomes' Integer

VariableBooleanAction:
    '\t\t\tvariable ' String 'becomes' Boolean

AssigneVariableAction :
    AssigneVariableIntegerAction | AssigneVariableBooleanAction

AssigneVariableIntegerAction :
    '\t\t\tvariable ' String 'becomes' String

AssigneVariableBooleanAction:
    '\t\t\tvariable ' String 'becomes' String

Transition :
    '\t\t\tnext_state_is ' String ('when {' Condition '}')?

Condition :
    SensorCondition | VariableCondition | MultipleCondition | UnaryCondition

SensorCondition :
    RELATIONAL_OPERATOR String ',' SIGNAL

VariableCondition :
    VariableIntegerCondition | VariableBooleanCondition

VariableIntegerCondition :
    RELATIONAL_OPERATOR String ',' Integer

VariableBooleanCondition :
    RELATIONAL_OPERATOR String ',' Boolean

MultipleCondition :
    LOGICAL_OPERATOR '{\n'
      (Condition '\n)*
    '\n}\n'

UnaryCondition :
    UNARY_OPERATOR '{\n'
      Condition
    '\n}\n'

RELATIONAL_OPERATOR :
  'EQUALS' | 'NOT_EQUALS' | 'LESS' | 'GREATER'

LOGICAL_OPERATOR :
  'AND' | 'OR'

UNARY_OPERATOR :
  'NOT'
```

Scénarios

Dans un premier temps les scénarios ont été définis en .ino avec Arduino IDE pour déterminer le code que nous devrions obtenir grâce à notre DSL. Cela nous a permis de mettre en avant des points complexes du code tel que le rebond du bouton pouvant fausser les valeurs récupérées lors de son appui et comment le gérer.

Ainsi les scénarios définis et qui ont pu être générés ensuite par notre DSL sont les suivants :

SCÉNARIO 1 : Very Simple Alarm

```
Program scenario1 :
  initial state is initial

  variables : {
  }

  components : {
    sensor button on pin 10
    actuator led on pin 11
    actuator buzzer on pin 12
  }

  states : {
    state initial :
      led -> LOW
      buzzer -> LOW
      next state is state2 when { button is HIGH }
    state state2 :
      led -> HIGH
      buzzer -> HIGH
      next state is initial when { button is LOW }
  }
```

```
Groovy
program "Scenario1" being {
  components {
    sensor "button" pin 10
    actuator "led" pin 11
    actuator "buzzer" pin 12
  }

  initialState "initial"

  states {
    state"initial" being {
      actuator "led" becomes SIGNAL.LOW
      actuator "buzzer" becomes SIGNAL.LOW

      next_state_is "state1" when { EQUALS "button", SIGNAL.HIGH }
    }

    state "state1" being {
      actuator "led" becomes SIGNAL.HIGH
      actuator "buzzer" becomes SIGNAL.HIGH

      next_state_is "initial" when { EQUALS "button", SIGNAL.LOW }
    }
  }
}
```

```
/* Components */
const byte button = 10;
const byte led = 11;
const byte buzzer = 12;

/* Variables */

/* States prototypes */
void initial();
void state2();

void setup() {
  pinMode(button,INPUT);
  pinMode(led,OUTPUT);
  pinMode(buzzer,OUTPUT);
}

void loop() {
  initial();
}

void initial() {
  delay(300);
  while(1) {
    digitalWrite(led,LOW);
    digitalWrite(buzzer,LOW);
    if(digitalRead(button) == HIGH) {
      state2();
    }
  }
}

void state2() {
  delay(300);
  while(1) {
    digitalWrite(led,HIGH);
    digitalWrite(buzzer,HIGH);
    if(digitalRead(button) == LOW) {
      initial();
    }
  }
}
```

SCÉNARIO 2

```
Program scenario2 :
  initial state is initial

  variables : {
  }

  components : {
    sensor button1 on pin 10
    sensor button2 on pin 11
    actuator buzzer on pin 12
  }

  states : {
    state initial :
      buzzer -> LOW
      next state is state2 when { ( button1 is HIGH AND button2 is HIGH ) }
    state state2 :
      buzzer -> HIGH
      next state is initial when { ( NOT ( button1 is HIGH AND button2 is HIGH ) ) }
  }
```

Groovy

```groovy
program "Scenario2" being {

    components {
        sensor "button1" pin 10
        sensor "button2" pin 11
        actuator "buzzer" pin 12
    }

    initialState "initial"

    states {
        state "initial" being {
            actuator "buzzer" becomes SIGNAL.LOW

            next_state_is "state1" when {
                AND {
                    EQUALS "button1", SIGNAL.HIGH
                    EQUALS "button2", SIGNAL.HIGH
                }
            }
        }

        state "state1" being {
            actuator "buzzer" becomes SIGNAL.HIGH

            next_state_is "initial" when {
                NOT {
                    AND {
                        EQUALS "button1", SIGNAL.HIGH
                        EQUALS "button2", SIGNAL.HIGH
                    }
                }
            }
        }
    }
}
```

```
/* Components */
const byte button1 = 10;
const byte button2 = 11;
const byte buzzer = 12;

/* Variables */

/* States prototypes */
void initial();
void state2();

void setup() {
  pinMode(button1,INPUT);
  pinMode(button2,INPUT);
  pinMode(buzzer,OUTPUT);
}

void loop() {
  initial();
}

void initial() {
  delay(300);
  while(1) {
    digitalWrite(buzzer,LOW);
    if(digitalRead(button1) == HIGH && digitalRead(button2) == HIGH) {
      state2();
    }
  }
}

void state2() {
  delay(300);
  while(1) {
    digitalWrite(buzzer,HIGH);
    if(!(digitalRead(button1) == HIGH && digitalRead(button2) == HIGH)) {
      initial();
    }
  }
}
```

SCENARIO 3

```
Program scenario3 :
  initial state is initial

  variables : {
    integer state -> 0
    integer oldState -> 0
  }

  components : {
    sensor button on pin 10
    actuator led on pin 11
  }

  states : {
    state initial :
      led -> LOW
      oldState => state
      state => button
      next state is state2 when { ( state EQUALS 1 AND oldState NOT_EQUALS 1 ) }
    state state2 :
      led -> HIGH
      oldState => state
      state => button
      next state is initial when { ( state EQUALS 1 AND oldState NOT_EQUALS 1 ) }
  }
```

Groovy

```
program "Scenario3" being {
    variables {
        variable "state" value 0
        variable "oldState" value 0
    }

    components {
        sensor "button" pin 10
        actuator "led" pin 11
    }

    initialState "initial"

    states {
        state "initial" being {
            actuator "led" becomes SIGNAL.LOW
            variable "oldState" becomes "state"
            variable "state" becomes "button"

            next_state_is "state1" when {
                AND {
                    EQUALS "state", 1
                    NOT_EQUALS "oldState", 1
                }
            }
        }

        state "state1" being {
            actuator "led" becomes SIGNAL.HIGH
            variable "oldState" becomes "state"
            variable "state" becomes "button"

            next_state_is "initial" when {
                AND {
                    EQUALS "state", 1
                    NOT_EQUALS "oldState", 1
                }
            }
        }
    }
}
```

Arduino

```
/* Components */
const byte button = 10;
const byte led = 11;

/* Variables */
int state = 0;
int oldState = 0;

/* States prototypes */
void initial();
void state2();

void setup() {
  pinMode(button,INPUT);
  pinMode(led,OUTPUT);
}

void loop() {
  initial();
}

void initial() {
  delay(300);
  while(1) {
    digitalWrite(led,LOW);
    oldState = state;
    state = digitalRead(button);
    if(state == 1 && oldState != 1) {
      state2();
    }
  }
}

void state2() {
  delay(300);
  while(1) {
    digitalWrite(led,HIGH);
    oldState = state;
    state = digitalRead(button);
    if(state == 1 && oldState != 1) {
      initial();
    }
  }
}
```

SCENARIO 4

```
Program scenario4 :
  initial state is initial

  variables : {
    integer state -> 0
    integer oldState -> 0
  }

  components : {
    sensor button on pin 10
    actuator led on pin 11
    actuator buzzer on pin 12
  }

  states : {
    state initial :
      led -> LOW
      buzzer -> LOW
      oldState => state
      state => button
      next state is state2 when { ( state EQUALS 1 AND oldState NOT_EQUALS 1 ) }
    state state2 :
      buzzer -> HIGH
      led -> LOW
      oldState => state
      state => button
      next state is state3 when { ( state EQUALS 1 AND oldState NOT_EQUALS 1 ) }
    state state3 :
      buzzer -> LOW
      led -> HIGH
      oldState => state
      state => button
      next state is initial when { ( state EQUALS 1 AND oldState NOT_EQUALS 1 ) }
  }
```

Groovy

```
program "Scenario4" being {
    variables {
        variable "state" value 0
        variable "oldState" value 0
    }

    components {
        sensor "button" pin 10
        actuator "led" pin 11
        actuator "buzzer" pin 12
    }

    initialState "initial"

    states {
        state "initial" being {
            actuator "led" becomes SIGNAL.LOW
            actuator "buzzer" becomes SIGNAL.LOW
            variable "oldState" becomes "state"
            variable "state" becomes "button"

            next_state_is "state1" when {
                AND {
                    EQUALS "state", 1
                    NOT_EQUALS "oldState", 1
                }
            }
        }

        state "state1" being {
            actuator "led" becomes SIGNAL.LOW
            actuator "buzzer" becomes SIGNAL.HIGH
            variable "oldState" becomes "state"
            variable "state" becomes "button"

            next_state_is "state2" when {
                AND {
                    EQUALS "state", 1
                    NOT_EQUALS "oldState", 1
                }
            }
        }

        state "state2" being {
            actuator "led" becomes SIGNAL.HIGH
            actuator "buzzer" becomes SIGNAL.LOW
            variable "oldState" becomes "state"
            variable "state" becomes "button"

            next_state_is "initial" when {
                AND {
                    EQUALS "state", 1
                    NOT_EQUALS "oldState", 1
                }
            }
        }
    }
}
```

```
/* Components */
const byte button = 10;
const byte led = 11;
const byte buzzer = 12;

/* Variables */
int state = 0;
int oldState = 0;

/* States prototypes */
void initial();
void state2();
void state3();

void setup() {
  pinMode(button,INPUT);
  pinMode(led,OUTPUT);
  pinMode(buzzer,OUTPUT);
}

void loop() {
  initial();
}

void initial() {
  delay(300);
  while(1) {
    digitalWrite(led,LOW);
    digitalWrite(buzzer,LOW);
    oldState = state;
    state = digitalRead(button);
    if(state == 1 && oldState != 1) {
      state2();
    }
  }
}

void state2() {
  delay(300);
  while(1) {
    digitalWrite(buzzer,HIGH);
    digitalWrite(led,LOW);
    oldState = state;
    state = digitalRead(button);
    if(state == 1 && oldState != 1) {
      state3();
    }
  }
}

void state3() {
  delay(300);
  while(1) {
    digitalWrite(buzzer,LOW);
    digitalWrite(led,HIGH);
    oldState = state;
    state = digitalRead(button);
    if(state == 1 && oldState != 1) {
      initial();
    }
  }
}
```

SCENARIO 5

Un scénario supplémentaire couvrant la mise en place de delay dans le code a également été implémenté.

```
Program scenario5 :
  initial state is initial

  variables : {
    integer state -> 0
    integer oldState -> 0
  }

  components : {
    sensor button on pin 10
    actuator led on pin 11
  }

  states : {
    state initial :
      led -> LOW
      oldState => state
      state => button
      next state is state2 when { ( state EQUALS 1 AND oldState NOT_EQUALS 1 ) }
    state state2 :
      led -> HIGH
      pause 800 milliseconds
      next state is initial
  }
```

Groovy

```
program "Scenario5" being {
    variables {
        variable "state" value 0
        variable "oldState" value 0
    }

    components {
        sensor "button" pin 10
        actuator "led" pin 11
    }

    initialState "initial"

    states {
        state "initial" being {
            actuator "led" becomes SIGNAL.LOW
            variable "oldState" becomes "state"
            variable "state" becomes "button"

            next_state_is "state1" when {
                AND {
                    EQUALS "state", 1
                    NOT_EQUALS "oldState", 1
                }
            }
        }

        state "state1" being {
            actuator "led" becomes SIGNAL.HIGH
            pause 800 milliseconds
            variable "oldState" becomes "state"
            variable "state" becomes "button"

            next_state_is "initial"
        }
    }
}
```

Arduino

```
/* Components */
const byte button = 10;
const byte led = 11;

/* Variables */
int state = 0;
int oldState = 0;

/* States prototypes */
void initial();
void state2();

void setup() {
  pinMode(button,INPUT);
  pinMode(led,OUTPUT);
}

void loop() {
  initial();
}

void initial() {
  delay(300);
  while(1) {
    digitalWrite(led,LOW);
    oldState = state;
    state = digitalRead(button);
    if(state == 1 && oldState != 1) {
      state2();
    }
  }
}

void state2() {
  delay(300);
  while(1) {
    digitalWrite(led,HIGH);
    delay(800);
    initial();
  }
}
```

Extension

L'extension traitée lors de l'implémentation des DSL est celle de "Temporal transitions". Cette extension a été choisie par l'équipe car l'utilisation de delay en Arduino est omniprésente ne serait ce que pour éliminer les bruits lors de l'appui sur un capteur de type bouton.

Pour le mettre en place nous avons eu à ajouter un concept PauseAction qui hérite d'Action. Le concept Action nous permet de choisir entre les actions initialement prévues et une action de pause. Une action de pause générera dans le code un delay() avec autant de millisecondes que ce qu'on a précisé dans la PauseAction. Notre Domain Model avant l'ajout de cette extension permettait de créer un State qui contient plusieurs Actions. Les changements apportés pour mettre en place la temporal transition ont donc été mineurs, en effet, en faisant hériter notre nouveau concept PauseAction du concept Action, on permet à l'utilisateur de déclarer une pause à la place d'une action comme Actuator Action.

Analyse critique

Le choix de MPS nous a séduit par sa structure et sa documentation, il s'est toutefois montré complexe à utiliser parfois comme lors de la génération du code .ino car cette technologie est très vendor lock-in et peut difficilement être réutilisée ailleurs. Dans notre cas, cela ne pose que peu problème mais dans la création d'un véritable DSL pour un client il pourrait s'agir d'un point critique qu'il faudrait justifier et qui mènerait à une étude de marché afin de potentiellement trouver une technologie plus intéressante à utiliser pour l'entreprise.

Nous avons décidé de partir sur du Groovy pour le DSL interne car il permet, grâce à ses libertés syntaxiques et à certaines règles du langage, de faire un DSL très éloigné de la syntaxe d'un langage de programmation traditionnel tout en ne devant pas apprendre énormément de choses puisque Groovy repose sur Java, un langage que nous connaissons bien. Cette hypothèse s'est avérée juste. Nous avons un DSL facile à utiliser pour un novice en programmation logicielle et qui n'a pas demander l'apprentissage d'un nouveau langage complet pour son développement.

Nous avons fait face à un autre souci en définissant notre domain model et en implémentant nos DSL, il s'agit d'un problème connu du domaine de l'électronique à savoir le bruit généré par les composants électroniques comme le bouton. Le bruit perturbe la bonne exécution du programme car les rebonds peuvent faire transiter le code d'un état à l'autre sans que cela soit voulu. Il existe une solution logicielle consistant à ajouter un délai entre deux lectures d'un capteur afin d'éviter de capter les rebonds. Cependant, écrire ce DSL implique de simplifier l'écriture du code afin que des non-experts de l'informatique puissent écrire un programme et l'ajout de délais précis à des endroits bien spécifiques nous semblait trop complexe. Nous avons fait le choix d'ajouter un délai fixe à chaque changement d'état afin d'éviter la lecture des rebonds. Il existe toutefois une solution matérielle qui consisterait à ajouter un condensateur afin de lisser le bruit.

Responsabilité de chacun des membres de l'équipe

Le travail sur ce projet a été divisé entre deux binômes : Emma et Arthur se sont chargés du DSL externe, Laurie et Vincent ont travaillé sur le DSL interne.