Mars Robot

Sudhakar Gottam Pim Jager

January 29, 2016

Contents

1 Requirements of Mars Robot

The Mars Robot shall adhere to the following requirements, which are set up using the FURPS model, and are prioritized using the MOSCOW method.

Functional requirements		
Identifier	Priority	Description
F1	Must	The robot must not fall of the table
F2	Must	The robot must be able to detect colors
F3	Must	The robot must be able to detect rocks
F4	Must	The robot must be able to “measure” rocks
F5	Must	The robot must be able to push light-weight rocks
F6	Should	The robot should be able to avoid lakes
F7	Could	The robot could be able to park in a corner
F8	Must	The robot must be able to “measure” lakes
F9	Must	The robot must be able to avoid rocks
F10	Could	The generated programs could work on both robots
F11	Must	The DSL must allow control of the three actuators
F12	Must	The DSL must provide readouts of the the sensors
F13	Must	The DSL must provide a method of controlling actuators based on the values of sensors
F14	Must	The DSL must provide a method of prioritizing different actions
F15	Should	The DSL should abstract over commonly used functionality
F16	Could	The DSL could provide a method to specify and execute user specified subroutines
F17	Must	The DSL must generate a valid Java program which can be run on the mars rover.
F18	Must	The DSL must provide readable and writable global variables
F19	Must	The DSL must provide a method to write global variables based on sensor values
F20	Would	The robot can communicate with an external entity through Bluetooth or WiFi.
F21	Should	The DSL should provide a method to sound beeps and buzzes
F22	Should	The DSL should provide a method to write string to the LCD
F23	Could	The DSL could provide a way to write strings with values of sensor readings to the LCD

Table 1: Functional requirements of Mars Rover

Usability requirements		
Identifier	Priority	Description
U1	Should	The mapping between a generated Java program and a specification in the DSL should be easy to understand
U2	Must	The DSL must be easy to read and understand
U3	Would	The robot prints a trace of behaviors and values on the LCD for easy debugging

Table 2: Usability requirements of Mars Rover

Reliability requirements		
Identifier	Priority	Description
R1	Must	The robot must not be able to enter a deadlock state, unless one is specified in the DSL specification
R2	Should	The robot should have reliable and consistent sensor readings
R3	Could	The robot could be able to recover from driving into a lake

Table 3: Reliability requirements of Mars Rover

Performance requirements		
Identifier	Priority	Description
P1	Must	The robot must be able to complete mission in finite time
P2	Must	The robot must complete missions as fast as safely possible
P3	Should	The robot should reach its targets with as little movement as possible

Table 4: Performance requirements of Mars Rover

Supportability requirements		
Identifier	Priority	Description
S1	Would	The robot can be reprogrammed and debugged without a physical connection to it
S2	Could	The robot could be expanded with additional sensors or actuators

Table 5: Supportability requirements of Mars Rover

2 Layout of Mars Robot

The Mars rover uses two separate bricks to connect all peripherals. Because of this the two bricks need to communicate about the status of the different actuators and sensors.

This communication could introduce small delays, and as stated in Section 1 the most imported requirement for the rover is that is always keeps itself safe. Therefore we propose a layout where the most important sensors related to safety are connected to the same block as the two main motors. This ensures that the robot can always keep itself safe, even when the communication between the two bricks fails.

	Brick 1	**Brick 2**
Actuators	Left Motor Right motor Measurement Motor	
Sensors	Light left Light Right Ultrasonic Front Ultrasonic Rear	Color Sensor Gyro Sensor Touch Sensor Left Touch Sensor Right

Table 6: Connection of the sensors and actuators to the Mars Rover

In this layout Brick 1 is the main control brick, it handles the most important safety related sensors and all actuators. Brick 2 connects the other sensors and can then communicate their readings to Brick 1.

The touch sensor are not considered essential safety sensors as the mars rover is very sturdy and contact with blocks can already be mostly avoided by the ultrasonic sensor on the front.

If it turns out that two ultrasonic sensors on the same brick are problematic then the front ultrasonic sensor of brick 1 will be interchanged with the gyro sensor on brick 2.

2.1 Final Layout

After discussion with the other groups the final layout for the Mars Rover has been decided as shown in Table 7.

This layout has the benefit over our proposed layout that it does not have two Ultrasonic sensors attached to the same brick.

port	Brick 1	Brick 2
A	Left Motor	
B	Right motor	
C	Measurement Motor	
S1	Light left	Left touch
S2	Light Right	Right touch
S3	Ultrasonic Rear	Ultrasonic Front
S4	Gyro	Color Sensor

Table 7: Connection of the sensors and actuators to the Mars Rover

3 Development process of Mars Robot DSL and corresponding Missions

The Mars Robot DSL and corresponding missions will be developed in an agile way; the development is split into small sprints. Each sprint consists of designing, building, integrating and testing a particular function. At the end of each sprint a working product (DSL and code generation) is delivered.

These sprints ensure that the development process won't end in some form of integration hell of all the different sub-parts and shows possible errors in the design of the robot as early as possible. It also allows to check the product quick, and often, with the client, which ensures that the developed product matches the clients expectations.

The goal of the first sprint will be to write an proof-of-concept rover-program in Java to test all the different sensors and actuators and the communication between the two bricks. This has two main benefits, firstly it ensures that the design of the robot is correct and all the basic function of the robot work. Secondly, it provides an example for the implementation of the code generation.

Consecutive sprints will consist of implementing the different sensors and actuators, and writing the mission possible with the sensors and actuators implemented so far.

The selection of the goals for the next sprint is guided by the requirements of Section 1.

3.1 Planning

Table 8 shows the planning for development of the Mars Rover by listing which requirements from Section 1 will be implemented in which sprint.

Week	Goals
Week 1 (9th Dec)	In sprint 1 we will implement a proof of concept to test the various sensors and actuators of the mars Rover and to test the communication between the two bricks.
Week 2 (16th Dec)	The goal of the second sprint is to have a robot which can wonder around and does not fall of the table. • F1 • F11 (only the two driving actuators) • F12 (only the rear ultrasonic and light sensors) • F13 • F14 • F17 • U2 • R1
Week 3 (6th Jan)	The goal of the third sprint is to implement the detection , measuring and pushing of rocks and lakes. • F2 • F3 • F4 • F5 • F8 • F9 • F11 (the measurement actuator) • F12
Week 4 (13th Jan)	The goal of this sprint is to implement the actual missions • F18 • P1 • P2

Table 8: Planning for development of the mars rover

If there is time left after implementing all the requirements with priority 'must' then the following requirements with priority 'should' shall be implemented first:

- F21
- F22
- U2

4 A DSL for Mars Rover missions

To program the Mars Rover in an efficient manner we have constructed a DSL which allows the programmer to write missions for the mars rover. This DSL abstracts over the various specifics which are needed for every program for the mars rover, such as managing the Bluetooth connection, setting up sensors, reading sensor values, etc.

The DSL is a very powerful one, which allows the programmer to write complicated programs using variables, constants, conditional statements and subroutines. This way the DSL does not restrict the programmer too much in the type of programs that can be expressed, but instead supports the programmer by abstracting over the commonly used functionality.

4.1 Syntax and semantics

In this subsection we will describe the syntax and the semantics of our mars rover DSL and why certain design decisions were made. Code **??** shows the syntactic rule for a program in the dsl. Below we will descibe each of the refrenced rules further:

Code 1: Main syntactic rule

```
1  Robot:
2      'Behaviors:' behaviorOrder += BehaviorName+
3      ('Variables:' globals += Global*)?
4      ('Constants:' statics += Static*)?
5      ('Stops when:') stopBehaviour = ValueExpression
6      behaviors += Implementation+
7      subRoutines += SubRoutine*
8  ;
9
10 BehaviorName : name = ID;
11 Global : name = ID;
```

Behaviours In the syntax choices we've made for our DSL we assume that the programmer writing programs for the mars rover wants to write missions using the subsumption pattern. Therefore the building blocks of a program the mars rover DSL are behaviours. A program starts with a list of behaviours which form the mission. The implementation of these behaviours must be later specified in the program. The list of behaviours is ordered by importance of the behaviour, with the most important behaviour being the last behaviour in the list.

Variables We also assume that the who programmers want to write programs using our mars rover DSL are experienced programmers who would like to have extended control over the rover and want to be able to write more complex control or data flow than that offered by basic subsumption pattern. Therefore our DSL has variables.

These variables are global to all behaviours. Variables are untyped and can be assigned Integer or Boolean values. Boolean values are interpreted as integer values, `True` as 1 and `False` 0. Likewise when a variable is used as a boolean expression then any value > 0 is interpreted as `True` and a value ≤ 0 is interpreted as `False`. Variables need to be declared in the list of variables and have an initial value of 0.

Constants Often various constants are needed when programming a rover robot, for example a safe speed to drive at. These constants are depended on the kind of mission.

To help the programmer with this our DSL allows the definition of arbitrary constants. These constants are avalaible throughout the mars rover program, and have the same semantics regarding type as variables. When defining a constant it has to be assigned a value, this can not be changed later in the program.

Code 2 shows the syntax for defining constants. The rule *ValueExpression* will be discussed below.

Code 2: Syntax for defining constants

```
1  Static : name = ID ' = ' value = ValueExpression ';';
```

Goal A lot of mission have a goal and the robot can stop after completing this goal, eg. measuring all three lakes.

The programmer can define such a goal in our DSL as an expression in the `'Stops when'` rule of the mission. Whenever this expression evaluates to > 0 the robot will halt and no more behaviours will be executed.

Behaviour implementation Code 3 shows the syntax for implementing behaviours. The name of the behaviour to be implemented has to be specified, this has to be one of the behaviours in the list of behaviours specified at the start of the mission.

When the expression after `Takes control when:` evaluates to > 0 and this behaviour is the behaviour with the highest priority for which this expression evaluates to > 0, then this behaviour will get control and the list of expression in the `Does:` block will be evaluated in the order in which they occur. When after any of these expressions another behaviour with a higher priority wants control, than this behaviour will stop, the remaninder of the expressions to be evaluated will be discarded and the other behaviour will get control and start execution at the top of its `Does:` block.

Code 3: Syntax for implementing behaviours

```
1 Implementation :
2     'Implementation for' for = [BehaviorName] ':'
3     'Takes control when:'
4         controlCheck = ValueExpression
5     'Does:'
6         expressions += Expression+
7 ;
```

Subroutines Often behaviours share similar subroutines, eg. drive backwards and then turn. To lessen the amount of code a programmer needs to write our DSL allows for the specification of arbitrary subroutines.

Code 4 shows the syntax for defining these subroutines. Subroutines should be uniquely defined using a name, which can be used to later call the subroutine. When a subroutine is called all its expressions are executed in the order in which they occur. Unlike behaviours, subroutines are atomic, which means always all their expressions are executed, even if this gets the robot in dangerous situations. Therefore a programmer needs to take care when using subroutines.

Code 4: Syntax for subroutine definition

```
1 SubRoutine :
2     'Routine ' name = ID ':'
3     expressions += Expression+
4 ;
```

Expressions Expressions are statements to be executed.[1] Code 5 shows the syntax for expressions. Expresisons have the following semantics:

ValueExpression This is just a ValueExpression without any side effects.

Action An Action to be executed by the robot, see below.

AssignExpression Assigns the result of a ValueExpression to a variable.

IFExpression If the first ValueExpression evaluates to > 0 then execute the fist list of Expressions, otherwise execute the second list of Expressions if it is provided, otherwise do noting.

WHILEExpression If the ValueExpression evaluates to > 0 execute the Expressions and reevaluate the complete WHILEExpression, otherwise do nothing.

[1] For historic reasons they are called expressions, however, statements might be a more accurate name.

Code 5: Syntax for expressions

```
Expression :
    ((
      ValExpr
    | Action
    | AssignExpression
    ) ';')
    | IFExpression
    | WHILEExpression
;

ValExpr : vExpr = ValueExpression;
IFExpression :
    'IF' c=ValueExpression
        '{' t+=Expression+ '}'
        ('ELSE' '{' e+=Expression+ '}')?
;
WHILEExpression: 'WHILE' c=ValueExpression '{' b+=Expression+ '}' ;
AssignExpression: global = [Global] ' = ' v = ValueExpression ;
```

Action Actions are things done by the Robot. Code 6 shows the syntax for actions. The actions have the following semantics *-unless otherwise specified motors only include the left and right driving motor-*:

ForwardAction Set the specified motor to moving forward, if no motor is specified both will start moving forward.

RotateAction Rotate the specified motor the specified value, interpreted as degrees. If `wait` is specified then execution of the next statement will hold until the rotation has completed, otherwise it will continue immediately.

StopAction Stop the specified motor, if no motor is specified both motors are stopped.

SAccelerationAction Set the acceleration for the specified motor to the specified value, if no motor is specified the acceleration is set for both motors.

SSpeedAction Like SAccelerationAction, but for the speed of the motor(s).

SubRoutineAction Execute the subroutine identified by its name.

MeasureAction Lowers the measurement arm and then rises it again.

ShowAction Print either a string, or the value of a sensor to the LCD.

SoundAction Sound either a Beep or a Buzz

FreeAction Set the specified motor to free rolling.

Code 6: Syntax for actions

```
Action :
      ForwardAction
    | RotateAction
    | StopAction
    | SAccelerationAction
    | SSpeedAction
```

```
7       | SubRoutineAction
8       | MeasureAction
9       | ShowAction
10      | SoundAction
11      | FreeAction
12  ;
13  ForwardAction : {ForwardAction} 'Forward' (motor = Motor)? ;
14  RotateAction :  'Rotate' motor = Motor degrees = ValueExpression (
        blocking ?= 'wait')? ;
15  StopAction : {StopAction} 'Stop' (motor = Motor)? ;
16  SAccelerationAction : 'Set Acceleration' (motor = Motor)? v =
        ValueExpression ;
17  SSpeedAction : 'Set Speed' (motor = Motor)? v = ValueExpression ;
18  SubRoutineAction : 'Do' routine = [SubRoutine] ;
19  MeasureAction : {MeasureAction} 'Measure';
20  ShowAction : 'Show' (string = STRING | sensor = Sensor);
21  SoundAction : 'Sound' sound = Sound;
22  FreeAction : 'Free' motor = Motor;
23
24  Motor : m = EMotor;
25  enum EMotor :
26        LEFTMOTOR = 'LeftMotor'
27      | RIGHTMOTOR = 'RightMotor'
28  ;
29
30  enum Sound :
31        BEEP = 'Beep'
32      | BUZZ = 'Buzz'
33  ;
```

ValueExpressions ValueExpressions allow for the access of sensor values, constants, variables, literals and allows for comparison of these values. Code 7 Shows the syntax for defining ValueExpressions. The syntax seems complex at first to allow infix with priority, however the semantics are rather straightforward and will therefore not explained in more detail.

Code 7: Syntax for ValueExpression

```
1   ValueExpression :   Blevel1;
2
3   Blevel1 returns ValueExpression:
4       Blevel2
5       ( {ExpressionBinOp.left = current} bop = BBinaryOp right=Blevel2)*
6   ;
7
8   Blevel2 returns ValueExpression :
9       BNotExpr | Blevel3
10  ;
11
12  BNotExpr : "NOT" sub = Blevel3;
13
14  Blevel3 returns ValueExpression :
15      Blevel4
16      ( {ExpressionBinComp.left = current} bcomp = CompareOp right=
            Blevel4)*
```

```
;

Blevel4 returns ValueExpression :
      BVLiteral
    | BBLiteral
    | BVarLiteral
    | BSensorLiteral
    | BVBracket
    | ColorLiteral
;

BVLiteral : neg ?= ('neg')? aValue = INT;
BBLiteral : bValue = BOOL_LITERAL;
BVarLiteral : var = ID ;
BSensorLiteral : sensor = Sensor;
BVBracket : '(' bsub = ValueExpression ')' ;
ColorLiteral : color = Color;


enum BBinaryOp :
    AND = '&&' | OR = '||'
;
enum CompareOp :
    EQ = "equals" | NEQ = "!=" | GEQ = ">=" | GT = ">" | LEQ = "<=" |
        LT = "<"
;

enum Color :
      BLACK = 'BLACK'
    | BLUE = 'BLUE'
    | BROWN = 'BROWN'
    | CYAN = 'CYAN'
    | DARK_GRAY = 'DARKGRAY'
    | GRAY = 'GRAY'
    | GREEN = 'GREEN'
    | LIGHT_GRAY = 'LIGHTGRAY'
    | MAGENTA = 'MAGENTA'
    | ORANGE = 'ORANGE'
    | PINK = 'PINK'
    | RED = 'RED'
    | WHITE = 'WHITE'
    | YELLOW = 'YELLOW'
;

//terminals
terminal ALPHA : ('A'..'Z');
terminal BOOL_LITERAL returns ecore::EBoolean: 'True' | 'False' | '
   TRUE' | 'FALSE' ;
```

sensors Sensor values are accessed by the name of the respective sensor (Code 8). Sensor values are only updated at the start of a `Does:` block and before evaluating a `Takes Control:` expression. The sensor values used in the DSL are normalized.

`LeftLight, RightLight, FrontUS, RearUS` values are the raw sensor values multiplied by

100. `LeftTouch, RighTouch` are either > 0 when the respective sensor is touched or < 0 when it is not. `ColorID` is just the color ID seen by the color sensor, these can be compared to the colorLiterals. `Angle` just is the value of the angle of the gyro.

Code 8: Sensor names

```
enum Sensor :
       COLORIDSENSOR = 'ColorID'
     | LEFTLIGHTSENSOR = 'LeftLight'
     | RIGHTLIGHTSENSOR = 'RightLight'
     | FRONTULTRASONICSENSOR = 'FrontUS'
     | REARULTRASONICSENSOR = 'RearUS'
     | TOUCHSENSORL = 'LeftTouch'
     | TOUCHSENSORR = 'RightTouch'
     | ANGLESENSOR = 'Angle'
;
```

4.2 Instances

To complete mission set out in the requirements there are three instances of our DSL. `MeasureBlocks` which drives around and measures encountered blocks, `PushBlock` which pushes a light block of the table and `ParkInCorner` which drives around, measures the three lakes exactly once and when all three lakes are found parks the robot in a corner of the table.

4.2.1 MeasureBlocks

Attachement 1 shows the implementation of a mission which drives forward and measures any encountered blocks. When the robot is in danger of falling of the table it will get itself to safety and turn a bit, because of this turning the robot covers the whole table when driving around.

This mission has six behaviours, `DetectCliff` and `DetectOutsideline` keep the robot from falling of the table. The `MeasureBlockBoth, MeasureBlockRight, MeasureBlockLeft` behaviours measure a block when it is detected using the touch sensors, `MeasureBlockLeft` and `measureBlockRight` will reposition the robot so the block is below the arm before measuring. `DriveForward` is the default behaviour, which just makes the robot drive forward in a straight line.

4.2.2 PushBlock

Attachment 2 shows the implementation of a mission which drives around and pushes light blocks of the table. This mission uses the fact that light blocks do not trigger the touch sensors, while heavy blocks do. Therefore the robot only tries to detect blocks using the touch sensor. Heavy blocks are then avoided and light ones are pushed of the table as a biproduct of driving around.

This mission has four behaviours. `DetectCliff` and `DetectOutsideLine` behave similar as in the `MeasureBlocks` mission. `AvoidHeavyBlock` drives away from a heavy block once it has detected one using either touch sensor. `DriveForward` behaves the same as in the `MeasureBlocks` mission.

4.2.3 ParkInCorner

Attachment 3 shows the implementation of the ParkInCorner mission. This mission is the most complex of the three missions and describes a mission consisting of the parts, first measuring all three lakes exactly once ,while keeping the robot safe from cliffs, driving into lakes or blocks, and then parking the robot in a corner of the table.

To achieve this the mission has 13 behaviours. The mission also uses 4 variables to track state. For each lake there is a variable which indicated whether it has been found or not, and there is a variable which indicates if the robot is currently tracking a line in search for a corner.

Safety `DetectCliff` and `DetectOutsideLine` work similar to the other two missions. `AvoidBlock` avoids a block when one is detected using the front ultrasonic sensor or the either touch sensors.

Detecting lakes `MeasurBlueLake, MeasureRedLake` and `MeasureGreenLake` measure a lake when the color sensor is positioned over a lake line and the respective lake has not been found (determined using the respective variable). After measuring the variable which indicates that that lake is measured is set to `True`.

`DetectLakeLineLeft` and `DetectLakeLineRight` take control when respectively the left or the right sensor is above a lake line and move the robot left or right to try and get the robot in position for a successful measurement of the lake. If the color sensor get above the lake line then the measurement behaviour for that color will take control.

`DetectLakeColor` takes control when the light sensor is above a lake line and none of the measurement behaviours take control. This behaviour steers the robot away from the lake if it doesn't need to be measured, to make sure the robot doesn't drive into the lake.

Parking in the corner When `DetectOutsideLine` is active it checks if all three lakes have been found. If they are then the variable for tracking line is set to `True`. This causes `FollowLine1` to take control, which will align the robot parallel next to the outside line, `FollowLine2` then takes control and drives the robot to the corner very carefully. When any of the safety behaviours take control they set the variable for line tracking to false an take the robot to safety.

Stopping When the right light sensor detects the outside line and the robot is currently tracking the outside line then the mission is complete.

Because the rotate action of the robot is clockwise the outside line followed will always be on the left hand side of the robot, thus once the right light sensor detects an outside line a corner is found.

4.3 Code generation

The program a mars rover using our DSL two Java files are generated; one for each brick on the rover.

4.3.1 Right Brick

The Java file for the right brick is a static file containing a Java class which controls the right brick. As per Section 2 the right brick contains only sensors and actuators. Therefore the sole task of the right brick is to continuously read its sensor values and send them to the left brick via Bluetooth.

4.3.2 Left Brick

The Java file for the left brick contains a mixture of generated and static code. It contains a main class and classes for each specified behaviour. Attachment 4 shows this file for the MeasureBlocks mission, with each generated line prefixed with a >. Below we will describe the general layout of the code in a pseudo Java syntax.

```
1 < list of necesarry imports >
2
3 public class Robot {
4
5     < variables needed for actuators and sensors >
6
7     < storage of sensor values send via Bluetooth from right brick >
8
```

```
9       < variables needed for Bluetooth connection >
10
11      < GENERATED list of constants eg. >
12      public static int CONSTANT_NAME = (40);
13
14      //is the robot running?
15      public static int _running = 1;
16
17      < GENERATED list of variables eg. >
18      public static int variableName;
19
20      public static void main(..) {
21          init();
22          Behavior[] bList = {
23                              < GENERATED list of behaviours eg. >
24                              new BehaviourName(),
25                              < Every robot has this behaviour for '
                                    stops when:' >
26                              ,new DefaultQuitBehaviour()
27                              };
28          Arbitrator ar = new Arbitrator(bList); ar.start();
29      }
30
31      < init() function: sets up Bluetooth, resets sensors and
32              actuators, starts the bluetooth and lcd threads. >
33
34      < updateSensors() update the sensors attached to this brick >
35
36      < normalisation functions to allow typeless variables in the DSL
37              in the typed JAVA language >
38
39      < Thread which prints the current sensor values and active
40              behaviour to the LCD >
41
42      < Thread wich reads incoming bluetooth messages from the right
43              brick and saves the sensor values >
44
45      < function for the measurement action >
46
47      < GENERATED list of subroutines eg. >
48      public static function routineName(){
49          Robot.rightMotor.stop();
50          variableName = CONSTANT_NAME;
51      }
52  }
53
54  class DefaultQuitBehaviour implements Behavior {
55      @Override
56      public boolean takeControl() {
57          return < GENERATED 'stops when' expression >;
58      }
59      < action() and suppress() functions >
60  }
61
```

```
< GENERATED Behaviour classes which have the following form >
class < GENERATED name of behaviour > implements Behaviour {
    private boolean _supressed = true;
    public boolean takeControl() {
        Robot.updateSensors();
        return Robot.makeBool(Robot._running) &&
                < GENERATED 'takes control when' expression >;
    }
    public void action() {
        _supressed = false;
        Robot.updateSensors();
        < GENERATED list of expressions for this behaviours 'Does:'
          with a supression check before every expression >
        if(_supressed) return;
        Robot.routineName;
    }
    public void suppress() {_supressed = true;}
}
```

Note that when generating expressions care has to be taken that all expressions have the correct type. Therefore the robot contains some normalisation functions which convert from bool to int and from int to bool.

5 Realisation

Section 3 described our plan for developing the mars rover DSL and its instances.

In the actual development phase we have moved faster than planned. At the end of week 2 we had already implemented everything from the second sprint and F2, F3, F4,, F5 and F9 from sprint 3. At the end of week 3 we had completely implemented sprint 3 and sprint 4. Week 4 was then spend implementing F21, F22, and F7.

5.1 Final result

Tables 9, 10, 11, 12 and 13 show the conformance of the final product to the requirements of section 1. In these tables 'V' is means the product met the requirement, 'X' means it does not meet the requirement and '?' means it is not objectively clear if the product meets the requirement. Each requirement which does not have a 'Would' priority and is not marked with 'V' will be discussed.

Functional requirements			
Identifier	Priority	Description	resu
F1	Must	The robot must not fall of the table	V
F2	Must	The robot must be able to detect colors	V
F3	Must	The robot must be able to detect rocks	V
F4	Must	The robot must be able to “measure” rocks	V
F5	Must	The robot must be able to push light-weight rocks	V
F6	Should	The robot should be able to avoid lakes	V
F7	Could	The robot could be able to park in a corner	V
F8	Must	The robot must be able to “measure” lakes	V
F9	Must	The robot must be able to avoid rocks	V
F10	Could	The generated programs could work on both robots	V
F11	Must	The DSL must allow control of the three actuators	V
F12	Must	The DSL must provide readouts of the the sensors	V
F13	Must	The DSL must provide a method of controlling actuators based on the values of sensors	V
F14	Must	The DSL must provide a method of prioritizing different actions	V
F15	Should	The DSL should abstract over commonly used functionality	V
F16	Could	The DSL could provide a method to specify and execute user specified subroutines	V
F17	Must	The DSL must generate a valid Java program which can be run on the mars rover.	V
F18	Must	The DSL must provide readable and writable global variables	V
F19	Must	The DSL must provide a method to write global variables based on sensor values	V
F20	Would	The robot can communicate with an external entity through Bluetooth or WiFi.	X
F21	Should	The DSL should provide a method to sound beeps and buzzes	V
F22	Should	The DSL should provide a method to write string to the LCD	V
F23	Could	The DSL could provide a way to write strings with values of sensor readings to the LCD	V

Table 9: Realisation of functional requirements of Mars Rover

Usability requirements			
Identifier	Priority	Description	result
U24	Should	The mapping between a generated Java program and a specification in the DSL should be easy to understand	V
U25	Must	The DSL must be easy to read and understand	V
U26	Would	The robot prints a trace of behaviors and values on the LCD for easy debugging	V

Table 10: Realisation of usability requirements of Mars Rover

Reliability requirements			
Identifier	Priority	Description	result
R1	Must	The robot must not be able to enter a deadlock state, unless one is specified in the DSL specification	V
R2	Should	The robot should have reliable and consistent sensor readings	?
R3	Could	The robot could be able to recover from driving into a lake	X

Table 11: Realisiation of reliability requirements of Mars Rover

Performance requirements			
Identifier	Priority	Description	result
P1	Must	The robot must be able to complete mission in finite time	V
P2	Must	The robot must complete missions as fast as safely possible	?
P3	Should	The robot should reach its targets with as little movement as possible	X

Table 12: Realisation of performance requirements of Mars Rover

Supportability requirements			
Identifier	Priority	Description	result
S1	Would	The robot can be reprogrammed and debugged without a physical connection to it	X
S2	Could	The robot could be expanded with additional sensors or actuators	X

Table 13: Realisation of supportability requirements of Mars Rover

R2 - '?' The sensor readings from the robot are reliable and consistent enough. However they are not calibrated at the start of a mission and especially the color sensor can report wrong colors in some light conditions. Therefore the sensor readings aren't as fool proof as we would have liked.

R3 - 'X' The Robot can sometimes drive out of a lake, but under some conditions it can not. Instead we have opted to implement F6 very well, to stop the robot from driving into lakes in the first place.

P2 - '?' The Robot is able to complete missions in a timely fashion, however we have not spend much time optimising this speed, therefore it might be possible that the robot can complete the missions faster while still staying safe.

P3 - 'X' We have spend little effort minimising the robots movement and although the robot doesn't do a lot of unnecessary movement, some assumptions -such as always turning right to avoid danger- cause the robot to move more than necessary.

S2 - 'X' It is currently not possible to add extra sensors or actuators to the robot without changing the DSL.

6 Evaluation

In this section we will evaluate the different aspects of our Mars Rover DSL, the development process and the tools used in this process.

6.1 DSL

In general we are quite happy with our DSL. It supports control statements such as `IF` and `WHILE`, variables and boolean expression it is very powerful. Therefore almost any type of mission can be implemented. We also feel our DSL is very clear and even those not familiar with it can understand missions written in out DSL without much difficulty.

Improvements However, as always, there are always improvements to make. We will list some improvements we would like in the list below:

Code validation Right now code written in the DSL is not validated, this means that missions can generate invalid JAVA code. When looking at the JAVA-error it is often obvious which

error was made in the DSL (e.g. a typo in a variable name), however it would be a welcome addition if code validation was done before code generation.

Missions with multiple stages Right now it is possible to have missions with multiple stages (e.g. PatkInCorner), however this requires the programmer of a mission to keep track of the stages by hand using variables. It would be nice if the DSL would support staged missions where behaviours van be marked as being active only in a certain (set of) state(s) and there would be an action to switch to a different state.

Arithmetics Our current DSL only supports boolean binary operations, it would be nice to have (basic) arithmetic operations.

Functions Right now our DSL supports subroutines and although some basic information exchange between routines is possible using variables, it would be nice to have proper parameterized functions, which would allow for e.g. arbitrary recursion.

Default safety behaviours We have opted to not include default safety behaviour in our DSL to allow for greater flexibility in missions. However we have found that the majority of the missions need this default safety behaviour, therefore it would be nice to have default safety behaviour in the DSL. To still allow for flexibility the behaviour could be made optional.

Flexibility When developing our DSL we have considered the sensors and actuators connected to the robot and their layout as definitive. Changes made to the where sensors are connected are relatively easy to implement, even when the sensors would switch bricks, by changing the code generator. However changes to the actuators would be a lot harder. Changing which ports the actuators are attached to is not very difficult, however the complete DSL and underlying code generation has been designed with the assumption that all actuators are attached to one brick. If actuators would be attached to multiple bricks then that would require a some less than trivial changes to the DSL and code generator to deal with synchronisation and two way communication between the two bricks.

Adding new functionality to the robot, such as new actions to ease the life of DSL programmers is quite easy. The actions just have to be added to the syntax and to the code generator.

6.1.1 DSL for modeling and code generation

Building a DSL for the Mars rover was very useful. Although designing the syntax and implementing the code generation is far from trivial it has made the implementation of missions for the rover a lot easier and faster. When considering the written missions we expect that writing them directly in JAVA would only have been marginally faster than the complete process if implementing the DSL and writing them in the DSL.

However some parts of implementing the DSL were quite difficult, most notably designing the syntax. Designing a syntax which can easily be parsed and is easy for programmers to write in is definitely not trivial and took quite a lot of effort. Our final syntax still contains some imperfections because of this, such as using `equals` as equality operator instead of the more used `==`.

6.2 Development Process

In Section **??** we have already briefly touched upon our development process. We feel that the chosen development process was very suitable for this DSL. Spending the first sprint on a proof of concept of a generated JAVA program allowed us to quickly asses if the assumptions we had made were valid and adjust accordingly. It also greatly eased the impementation code generation in the later sprints.

Dividing the development process in sprints also had the added benefit that we could continuously test a working rover. Each sprint just added more functionality to the working product.

There was however also a downside to this, because some functionality, e.g. the pushing of the blocks, was implemented in an early sprint this functionality did not receive the quality improvements that later sprints added to the robot.