Eindhoven University of Technology

2IMP20 Generic Language Technology

Assignment 2
RobotDSL

README

Authors:

Tianyu Liu
Student Number: 0937147
E-mail: t.liu.1@student.tue.nl

Xiayang Hao
Student Number: 0892474
E-mail: x.hao@student.tue.nl

June 8, 2017

1 Model design strategy

The metamodel of this RobotDSL contains the following classes:

- `Script`
  The entry class of the whole metamodel, represents the script itself. The `Script` class has one attribute, namely `name` with type `EString`, represents the name of the script.

  `Script` composites *zero to many* class `Statements`. Since one script can have multiple statements, we used this relation. Also, as designed in the Rascal grammar, we allow so-called 'empty statement list' in the script, so the composition relation is *zero to many*.

- `Statements`
  This is the class represents all the statements in the script. It is the super type (also known as super class) of two classes: `RunningStatements`, the statements that used to operate the robot and `BuildStatements`, the statements that used to build or destroy the scene.

  Since a statement can be either running statement or build statement, we used the super type relation to represent this.

- `RunningStatements`
  This is the sub type of `Statements`, represents all statements apart from the build statements.

  A running statement can be one of the following: command statement, if statement, while statement or repeat statement. As a result, `RunningStatements` is the super class of the mentioned four classes.

- `BuildStatements`
  This class represents the statements to change the scene (build or destroy walls, pick or drop marks). Hence we created 4 sub classes, namely `BuildWall`, `DestroyWall`, `DropMark` and `PickMark`. All of the 4 classes consider `BuildStatements` as its super class.

  Each of the sub class consists of 2 attributes, namely `row` with type `EInt` and `column` with type `EInt`. Since for each of the build statement, a specific position (in row and column) will be given.

- `CommandStatement`
  This class represents the basic command statement as described in the language description. Hence, it has 6 sub classes: `Step`, `TurnLeft`, `Drop`, `Pick`, `TraceMessage` and `Comment`, represents the 6 basic commands of the language, respectively. Since comments are also considered as commands, we put `Comment` as a sub class of `CommandStatement`.

- `IfStatement`
  The `IfStatement` represents the 'if' conditional statement in the robot DSL. It contains exact one *logical expression* (the logical expression can have many binary logical

expressions) so we added a *one to one* composition relation to class `LogicalExps`. A `IfStatemet` can also have many running statements (empty statement also allowed here) in it. Therefore we added a *zero to many* composition relation to `RunningStatements`.

- `WhileStatement`
  This class represents the 'while' loop statement. Similar to `IfStatement`, a logical expression and many running statements is composited in the while statement.

- `RepeatStatement`
  This class represents the 'repeat' statement of the DSL. It has one attribute `time` with type `EInt`, represents the *repeat time*. Since a repeat statement can have multiple command statements, it has a *zero to many* composition relation to `CommandStatement`

- `Basic commands`, 6 classes
  There are 6 basic commands as described, hence we created 6 classes to represent them, namely `Step`, `TurnLeft`, `Pick`, `Drop`, `TraceMessage` and `Comment`.

  `TraceMessage` has one attribute named `message` with type `EString` as this command consists of one string message.

  `Comment` has one attribute named `comment` with type `Estring` to represent the comment message.

- `LogicalExps`
  This class represents the general logical expressions, including single logical expressions as well as binary ones. Therefore, we created two sub classes, namely `SingleLogicalExp` and `BinaryLogicalExp`.

- `SingleLogicalExp`
  This classes contains the 4 basic logical expressions introduced in this robot DSL. 4 classes, `Full`, `WallAhead`, `Mark` and `Heading` are created to represent the basic logical expressions, respectively.

- `Full`, `WallAhead` and `Mark`
  These three classes are sub classes of `SingleLogicalExp`. In order to represent the extended logic of the DSL, we created a class named `Not` to represent the negativity of the single logic.

  Also, a composition relation to class `Not` was added. Since for each single logical expression, there can be at most one 'not' associated with it, we set the composition relation to *zero or one* relation.

- `Heading`
  This class is also the sub class of `SingleLogicalExp` and represents one of the 4 basic logical expression. Different from the other 3, it has one attribute, named `direction` with type `Directions` to show the exact heading direction. `Directions` is an enumeration.

- `Directions`
  This is an enumeration classifier, enumerates 4 directions, namely 'south', 'north', 'west' and 'east'. It is a data type and represents the direction of robot.

- `BinaryLogicalExp`
  This class represents an extension of the basic logical expression. The extended feature allows to consider multiple basic logical expressions as one logical expression.

  Two or more basic logical expressions are connected by a binary operator. Hence we added one *two to many* composition relation to `SingleLogicalExp`. Also, a *one to many* composition relation is added to class `Binaries`, which represents the binary operators.

- `Binaries`
  This class represents the binary operators, namely 'and' and 'or' in this case. Hence two sub classes, `And` and `Or` were created to represent the operators, respectively.

2 Grammar changes

The automatic generated grammar (.xtext file) is not good enough, we modified a little bit in it to make it satisfy the DSL requirement. Since the changes happened in many places, it is not possible to list all changes in this document. Only the general explanation will be described.

- Delete useless blocks
  The grammar generator generates something like `Statements_Impl` and added into the main blocks. However, we do not need such '_Impl' blocks. Hence we simply deleted the reference in the main blocks.

- Modify the sequence of tokens
  For each block, we modify the sequence of tokens to satisfy the requirement. For instance, in `IfStatement` we put logical expressions between token `if` and `do`.

  Also, we added some more tokens like `runs as` in `Script` or `else do` in `IfStatement`.

- Add necessary data type
  For classes like `Script` and `Comments`, they have attributes with type `EString`. However, the default definition of `EString` is not sufficient for us. We created a new *terminal* called `NAME`, returns `EString` that consists of only letters since the script name can only be letters. Also, we created a new *terminal* called `COMMENTS` to represent the comment, namely a string start with a '#' and end with a new line.

  This changes also happened on the trace message.

3 Difficulties with the metamodel

There are following difficulties we faced in making the metamodel:

- Find a proper representation of the context free grammar
  It is difficult to find the representation, especially the relation between classes. We must be very careful to determine whether there should be a normal reference between two classes, or a super class/sub class, or a reference with containment (i.e. composition).

- Extend a meta model
  We found it is difficult to extend a meta model. Originally, we decided to build the extensions (logical extension and build scene extension) by inheritance. However, we had some difficulties in building the .genmodel file as well as the grammar. The grammar only includes the base model but not the extended one. We have no idea on how it works till now. Fortunately, we succeed in building the extended model itself.

4 Difficulties with the grammar

The following difficulties are what we faced in building the grammar:

- The containment relation
  The relation is containment or not affects the grammar a lot. We spent quite a few hours on finding out why our grammar did not work. Finally we found that the problem was in the ecore model.

- The + = operator is a little bit confusing.
  Different from the real context free grammar, the xtext grammar uses operators to represent the existence of some tokens. At the very first moment we found it is confusing because it is quite different from normal grammars. Generally speaking, the learning curve of xtext grammar is flat.

- EString, EInt, INT, STRING can be mixed up
  There are type like EString, EInt but also type like INT and STRING. They are different but with very similar names. EString can be incorrectly considered as normal Strings but this is definitely not the case. We messed up these types at the beginning.

5 Comparison between Xtext and Rascal

In general, the documentation of Xtext is better than Rascal, hence the learning curve of Xtext can be smoother than that of Rascal. It is much easier to get started with Ecore and Xtext than with Rascal. If we did not gain Rascal experience from *Software Evolution* course, we would got stuck at some point in the first assignment.

From the DSL language development point of view Rascal is better than Xtext. As mentioned in the previous section, Rascal uses normal regular expression and context free grammars to define a grammar but Xtext uses an alternated way. To develop in Xtext we need to start with modeling. It may take some effort to transfer a well formed Rascal grammar to a meta model. Also, rascal can generate a parse tree of the input script but Xtext cannot.

From the user's point of view Xtext is much better than Rascal. We do not think Rascal grammar can be applied to some text editor for real use. We have to use Rascal console to test the grammar. So for the developer who will use the DSL, an editor with Xtext plugin can be directly used as the development environment of the DSL, which is much more convenient and applicable.