Model-based Design of Cyber-physical Systems 2025
Assignment 2

Late submissions will not be accepted. Submission is done digitally via Canvas. Submit a zip archive of your meta workspace, a zip archive of your runtime workspace and a 'readme.txt' file that explains your language design in a couple of paragraphs. Clean both workspaces before you export them. Detailed submission instructions are given in the DSL assignment page on Canvas. The models directory of the runtime workspace should include examples programs written in your DSL, correct ones, as well as programs that trigger all the validation checks that you have implemented.

Setup Instructions

Start the Eclipse installation that was provided with the DSL Tutorial, or an identical installation. Create a directory 'GoL' that contains an empty subdirectory 'GoLMeta' outside of Eclipse as you did in the DSL tutorial for the Task example. Open the 'GoLMeta' directory as your meta workspace in Eclipse. You can use different names as long as you use this nested directory structure.

- Create a new Xtext project for the Game of Life DSL, following the instructions in the DSL tutorial. Choose appropriate names for the project and your Game of Life language and select a sensible extension for the language instances.
- Without modifying the grammar of the DSL, generate the 'Xtext Artifacts' as shown in Section 3.1.2 of the tutorial.
- Run the generated DSL as an Eclipse Application, as shown in Section 3.2, starting the run-time workspace in a second instance of Eclipse.

In the run-time workspace import the template project with:

- Choose 'File > Import > General > Existing Projects in Workspace' and select 'Next'.
- Select the archive file 'GoLruntime.zip' file and click 'Finish'.
- Right Click on the 'short_life' project in the Package Explorer and select 'Run as Java Application' to verify that the provided Java Game of Life application works. Create some live cells in the grid with the mouse and start the application with 'Game > Play'. The cells should now evolve according to the hard coded Game of Life rules.
- Create a DSL instance in the 'models' directory and validate that the default 'Hello' grammar works.

You are now ready to create your own Game of Life DSL.

1. Grammar — 3 Points

Define a DSL that allows experiments with variations of:

- the **initial grid**, i.e. which cells are initially dead or alive.
- **evolution rules**, i.e. when cells 1) die, 2) live, and 3) become alive.

Keep the evolution rules simple and avoid complex expressions. It is sufficient to use number of live neighbors <, =, > some defined number in the DSL instance.

It must be possible to specify all variation points mentioned in bullet list above. The grammar must be expressive enough to define any variation of the Game of Life and any initial grid configuration. The grammar must furthermore be readable and avoid unnecessary complexity. The easier the DSL allows you to specify the rules and initialize complex grids, the higher the score.

2. Code Generation 3 Points

Make a code generator that generates Java code that correctly implements the semantics of the DSL you have specified. The generator should generate new versions of the file RulesOfLife.java. The generated file should overwrite the originally provided file, either automatically or by manually moving the file.

The generated code should correctly implement the specified behavior for all DSL instances. The code should furthermore be readable and make good use of Xtend features.

Tip: Start by moving the current implementation of RulesOfLife.java into the code generator and generate the file just like it was originally provided. Then, modify the generator to consider the initial grid and evolution rules you have specified in your language.

3. Model Validation 3 Points

Add three validation rules to the language you have just created. The validation rules should be meaningful and help the user avoid specifying instances of the language that do not make sense. An example could be to make sure the initial grid does not address cells that our outside the grid in the implementation of the game. Another example could be to make sure the number of specified neighbors required to die, live, or become alive is valid. Think carefully about whether a validation failure should be treated as a warning or an error.

Implement at least three meaningful validation rules that significantly aid users in defining correct DSL instances. The implementation of the rules should be readable and make good use of Xtend features. The more useful the validation rules are, the higher the score.

4. Extra Feature 1 Point

Extend your DSL by adding a compelling non-trivial feature of your choice, such as incorporating useful quick fixes for addressing validation errors, enhancing formatting capabilities, or introducing unit testing support. You can refer to the book "Implementing Domain-Specific Languages with Xtext and Xtend-2nd Edition", available on Canvas, for inspiration.