Software Design Project Report

Fatemeh Taherifard, Nafiseh Valizadeh Shiran

April 13, 2023

Abstract

This document is a report for the software design project. The objectives of this project are to build a metamodel for a proposed domain-specific language, instatiate from it to make a model, implement validation constraints, and finally transform it to a text file.

1 Introduction

The concept we implement is a trail hiking management system. Most of us have tried to plan for a short hiking trip with a group of people. We have tried to gather people based on their experience in hiking, the difficulty level of the trip, who would join, what food to bring and who would bring them, and etc. We thought it would be better if we could automate these tasks using a DSL.

2 Question 2: Building the Metamodel

We firstly thought about the main concepts of our project. We came up with six main concepts; namely Trail, Location, Equipment, Person, Meal, and Weather. Since we wanted to be able to instantiate a number of trails from the root node, we had to add a root node called TrailManagement. This node is **composed of** more than zero Trails and possible Weathers (I'll discuss this decision later). Composition obviously makes more sense, since Trail and Weather instances exsist because of the TrailManagement. Each (one) Trail **consists of** (many) Meal, Equipment, Location, Person, and Weather. Composition was also a better choice since we wanted the instances to be removed if Trail was removed. Trail also **consists** of a number of Leaders. Since leaders are a type of person, we used **generalization**, so leaders inherit the properties of a person, but also has its own properties. For example, it is very important to check a leader's certificate and for them to provide a certificate and a contact information. However, the level of experience is a Person attribute since we might need to check that for everyone.

We decided to separate medical equipment and cooking equipment due to their important different properties. To be more specific, cooking equipment needs to be **referenced** to Meal, while medical equipment had to be characterized by their level of importance. The first connection helps us to list the equipment based on what the group would want to have for a meal in the trip, while the latter helps us to put constraints on medical equipment with high importance to have a supplier in the model. Suppliers are created by referencing equipment to person. Cooking equipment has a many-to-many relation to meal. We dedicated two labels for this connection since the user might want to access the meal based on the equipment or vise versa. Since these two labels must be synchronised we used **opposite** references. Moreover, since medical and cooking equipment cover everything we wanted to instantiate from the metamodel, we abstracted Equipment.

Weather is an important concept in trip planning. First, we need to specify one Weather for each trail as the current weather condition, as well as each Person references one weather as their tolerable

weather. This enables us to only choose people who can tolerate the current weather condition in the group. Please note that we do not need any **opposite** reference here, since we only need to access Weather from Person or Trail. The **challenge** here was that since person-weather relation was neither a composition nor a generalization, we couldn't instantiate it from the model. Therefore, we decided to make a composition relation from TrailManagement to Weather. This means that each management case builds possible weather conditions which a Person can choose as their tolerable weather. For this matter, to make finding the Weather instances easier for the user, we also defined a weather_id.

Locations are also a part of Trail. We defined booleans to check wheather a stop location is a destination or an origin. In addition, we check if the stop is an emergency spot which would be very important for trails with high level of importance.

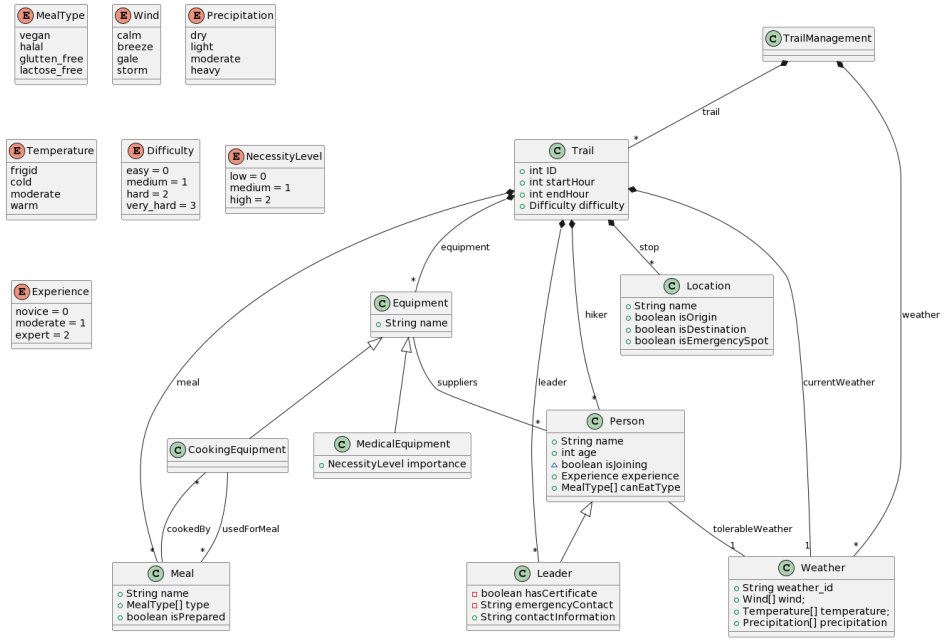

Figure 1: The class diagram of our metamodel. It consists of enums, classes, properties, relations and their labels.

Finally, after creating the EMF code based on the uml model, we generated a `.ecore` file using EMF. The file is an epsilon metamodel defined by Emfatic.

3 Question 3: Implementing the Editor

For this part we had three options: graphical model, EMF reflective tree-based editor, or a dedicated tree-editor.

3.1 Graphical Model

Based on the domain of our language, our metamodel does not need visualization for a user to instantiate from it. For example, it did not seem reasonable to use GMF to make a location or a weather, but rather confusing. Therefore, we decided to use the other options.

3.2 EMF editor

We can generate a `.model` file from the EMF model. For this we had to register the `.ecore` file, so the model would be connected to this ecore file by eclipse.
The **benefits** of using this editor are the simplicity and the fact that it can be used in the same workspace as the file projects. For our project in which we try to make a simple example of the model (prototype) is a very useful and stable approach.

3.3 Dedicated editor

Another approach is to generate a `.genmodel` file and then generate java code from it. Now, we can instantiate our metamodel from an eclipse plug-in which can be run in a new eclipse window. The **advantages** of this approach would be saparating the model from the metamodel codes. In addition, one can change the generated java code which offers more flexibility than a purely tree-based model.

We used both of the last two options. However, since we did not need to change the java code, the second approach worked perfectly for us.

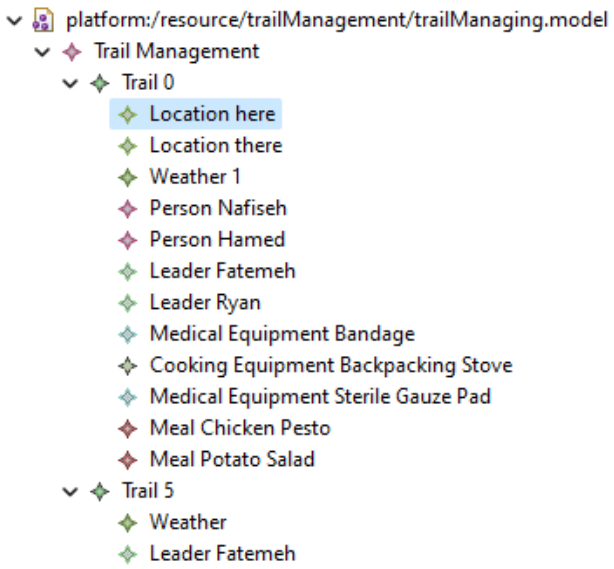

Figure 2: The EMF tree-based editor. An example of meta-model instantiation.

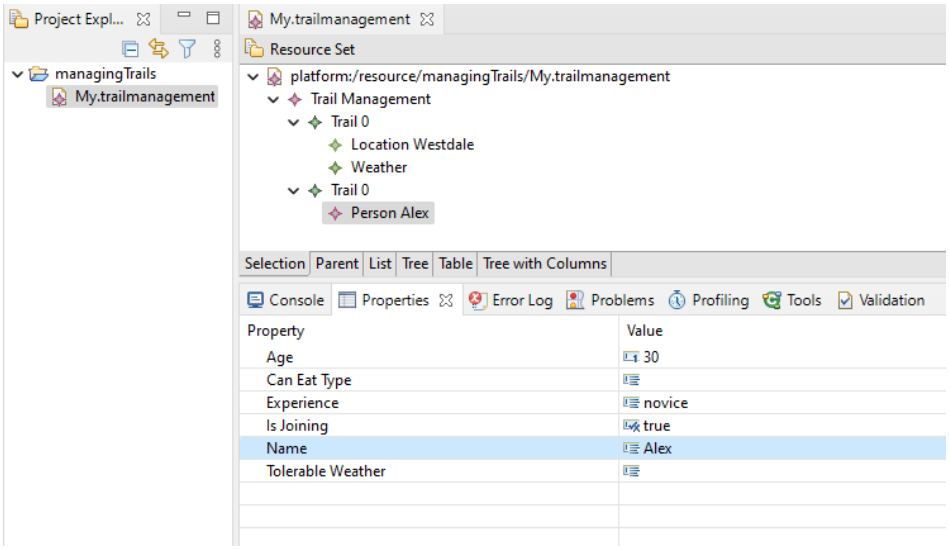

Figure 3: The dedicated tree-based editor opened from another eclipse instance. An example of meta-model instantiation.

As can be seen, we can use this editor to make models and set the properties of our model. However; the higher-level constraints on properties and connections such as people's age or the number of trails in a trailManagement case are not checked at this point. That is why we need another approach to validate more complex constraints.

4 Question 4: Validation

The constraints are divided by classes (contexts). The important constraints and less restrictive ones are defined using the keyword `constraint` and `critique`, respectively.
The first one is the TrailManagement class which has to have at least one trail to manage. This is an important constraint which would cause error, if not satisfied.

4.1 Trail

4.1.1 Constraints

- All instances of Trail must have **a unique ID**. In other words, the number of each instance (available by the keyword `self`) with a specific ID in the list of all instances is one.
- Since the trail is mostly for a short hike, we decided to make the **date property** in military hours between 0 and 1159.
- The **difficulty level**, the **current weather** cannot be null. Additionally, exactly one **origin** and one **destination** must be specified, and the trial must have at least one **joining leader**.
- In case (using the keyword `guard`) the difficulty level is very hard, the user must specify at least one **emergency spot**.

4.1.2 Critiques

- In case the difficulty level is hard, the user should specify at least one **emergency spot**.
- The meal should not be empty, so a person could choose from the possible meals served on the trail.

4.2 Location and Equipment

- The name of the stops and equipment cannot be empty. The users must know where they are going, and what equipment the suppliers must bring.

4.3 MedicalEquipment

The reason why we split the equipment from medical equipment was the significance of this type of equipment.

- It is very important to know how important a medical equipment is. There fore, a user must specify the level of significance, in order to check the next constraint.
- Now that we have the level of importance, we can check that important equipment, at least one supplier exists.

4.4 Weather

4.4.1 Constraint

- Rainfall is a highly important factor in trail management. We decided that it must exist in both current weather condition and tolerable weathers for hikers.

4.4.2 Critique

- Wind and temperature could also be important for some users. That is why we give a warning in case someone thought they cannot be empty.

4.5 Meal

- Each instances must have a name and a type (halal, gluten free, ...). Otherwise, navigating them, and selecting them by the user's preferences are not possible.
- We also check that non-prepared meals such as sandwiches have a cooking equipment connected to them. Please, note that cooking equipment is a generic name we chose for utensils needed to prepare food, not just cooking them.

4.6 Person and Leader

4.6.1 Person Constraint

- The name cannot be empty, and generally people older than 12 can join the trail.
- Based on experience weather is one of the most important reasons why a person will or will not join the trail. For that, we first check if the person is joining the hike and if they specified a tolerable weather. If so, we checked that each weather condition in the current weather instance exist in the tolerable weather specified by the hiker. For that, we had to break down each weather to its three categories (to be more precise, properties) wind, precipitation, and temperature.

4.6.2 Person Critique

- We first check that the types of food a person wants eat is not empty. Then, if the meals specified for the trail is not empty, we check that there is at least one food in each four categoris of halal, gluten free,... .

Leader is a child class of Person. Inheriting all the constraints of Person, it also has other constraints defined on its own properties.

4.6.3 Leader Constraint

- We decided that each leader must be older than 18 and be an expert. They also must have a certificate and contact information since they are responsible for everything during the hike.

4.6.4 Leader Critique

- Since they should be in contact with everyone even before the trail, it is better that they also provide an emergency contact.

4.7 Integration

For a reflective tree-based editor, we made the `.evl` file inside the workspace, and ran the file on our model. For this, we needed to make a new configuration, an EVL validation set to the model registered from the ecore file.
For the dedicated tree editor, we did the same thing, but in a new editor. We did not encounter much of an issue integrating the validation file with the model. [1]

[1] A screen shot of an example validation view is provided in the appendix.

5 Question 5: Model to Text Transformation

For this section, we initially created a model that completely conforms to EVL constraints we explained earlier and gives a good example of a real-world testcase.

- We have implemented the task of Model to Text transformation. We automated the process of migrating the model by the use of EGL and EGX (Epsilon model transformation languages). They consume our EMF metamodel and provides an HTML file showing the model in a text format.
  The reason why we chose HTML is that a web page would be a good idea to represent a trail management system. We envisaged the system to be used as an easily-accessible system. Hikers would use this system to acquire the important information about each trails and even possibly in the future, add information, or communicate with admins. The template and the corresponding target location are both defined in an EGX file, which eventually creates our final HTML output. Moreover, and EGL is used to provide us the Epsilon language syntax in an HTML file.

- Since we wanted the users who are the people joining the hike, to have a holistic view of a trail details (Weather conditions, meal options, other joining hikers and leaders, cooking and medical equipment, start, end and stop locations and the difficulty level of the trail), we created only one HTML file and that satisfies our requirement for having a readable HTML file.

  We also contemplated the idea of having a main page view which links to several HTML files generated by multiple EGL files; however, for the extent of our project, a holistic view made more sense.
  However; we observed that by increasing the number of trails and their contents our HTML preview becomes cluttered and confusing for the user to navigate. Consequently, our decision was to make a drop-down list of available trails which contains all the information for each specific trail. This helps to have a better user experience as the page preview would look more structured.

  With this design decision, not only we simplified the HTML file by having a-whole-view of the model, but also we overcame the problem of a cluttered view.

- Since our idea was that a trail information webpage would be used by hikers, we tried to add a desirable design to attract the target users.
  We wanted to capture the most important connections between the classes. Firstly, for each trail we listed meals, leaders, and hikers. Also for each person, we showed the options they used as their meal preferences.
  Then to show cooking equipment and corresponding suppliers, we used query languages to show who will bring which equipment.
  Finally, we listed the current weather condition and the tolerable weather conditions by each person.

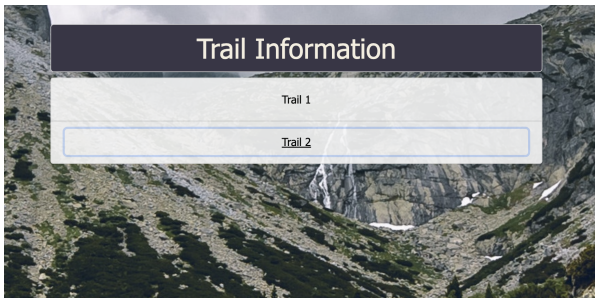

Figure 4: Available Trail Options, drop-down

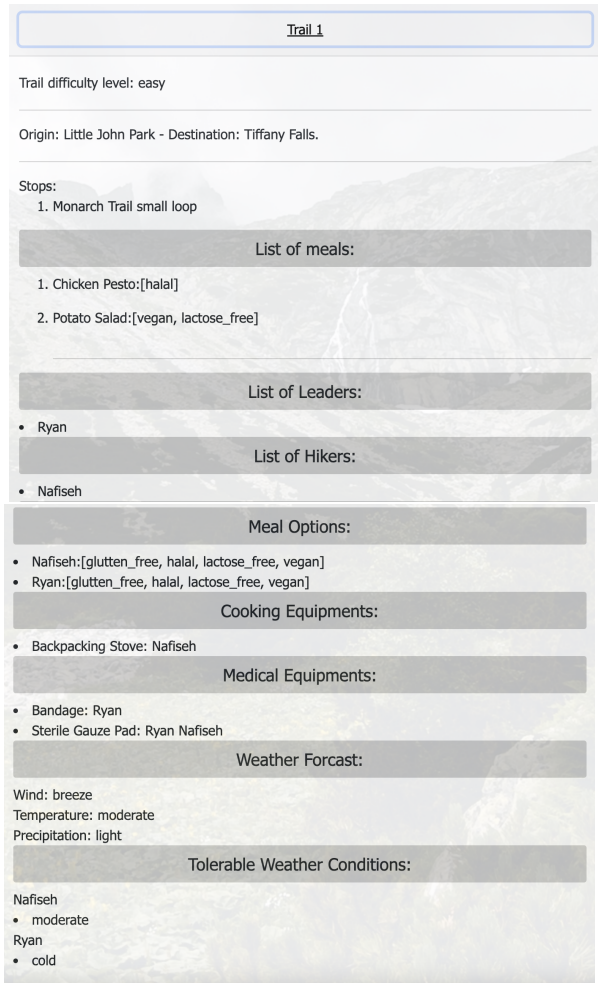

Figure 5: Trail1 detailed view

6 Conclusion and Results

In this report, we first demonstrated how we designed our metamodel for a hiking trail management language. Then, we explained the editor we used for building a model and then validating it using Epsilon evaluation language. Finally, we transformed the model into an HTML text file viewable and navigable by a user.

A Appendix

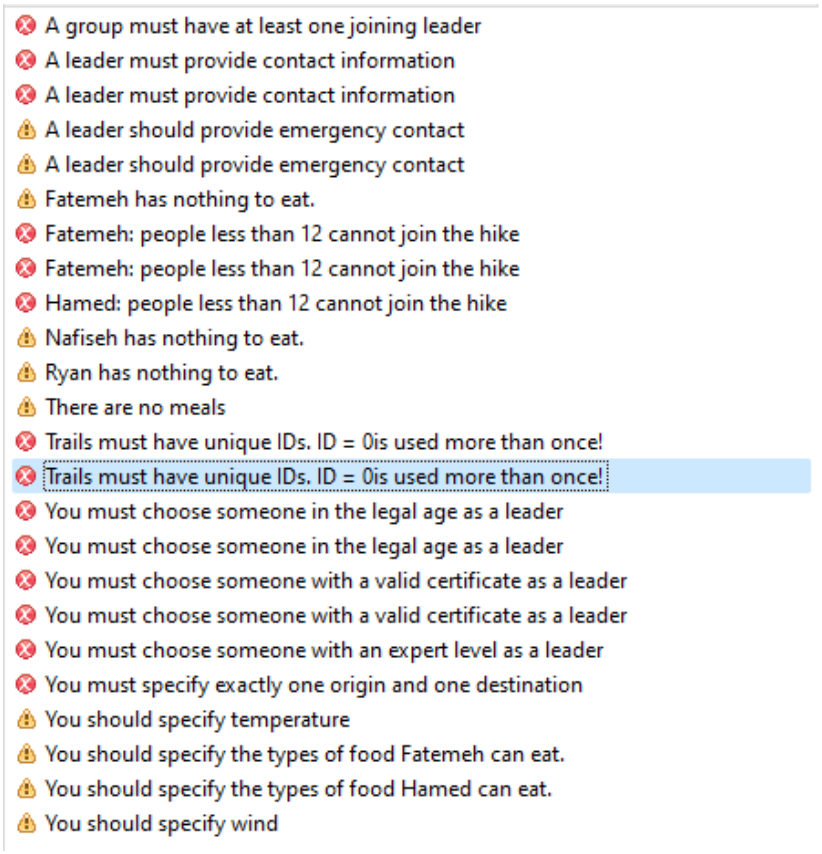

Figure 6: An example validation view.