System Specification

Extended Petri net Simulator

Group A

October 31, 2013

Abstract

This document includes the requirements for the Software Engineering 2 project, which consists of an Extended Petri net software system.

Contents

1 Introduction

Author: ***Thibaud, Albert***

> Petri nets, as graphical and mathematical tools, provide a uniform environment for modelling, formal analysis, and design of discrete event systems. [1]

Petri nets are used as a means to model systems, but, as they are a mathematical concept, they are not always easy to understand for the ordinary user. Complex systems can be modelled with Petri nets, and usually, this would be an engineer's job. Even though engineers can easily create and understand their own Petri nets, every team member in a company would like to be able to understand what a Petri net is about without having any knowledge of the concepts behind them.

Therefore, the project aims at creating a 3D visualisation from a Petri net, to allow non-Petri net experts to actually to understand how the model works and validate a system.

However, Petri nets were not intended to have a 3D representation. For instance, there is no graphical concept or way to say that a particular Petri net would look like a train track because it was used to model a railway system.

Thus, our goal for this project is the following: Providing an extension to Petri net models to make their 3D visualisation possible. For this purpose, we have imagined a simple link between a Petri net and a 3D visualisation.

1.1 Conventions

The priority of the requirements in Section 3 and 4 will be indicated by the keywords **shall**, **should** and **would be nice**:

- **Shall**: Requirements that must be implemented as a minimum, resources must be allocated to these requirements.
- **Should**: Requirements that add more functionality to the software, but are not the core required to work, if possible, resources will be allocated.
- **Would be nice**: Requirements considered good ideas and would be implemented if there are sufficient resources or in later versions of the software.

1.2 Audience

The audience of this document are persons affiliated with the company that models and simulates Petri net, developers of the system and final users of the Extended Petri net Simulator.

[1] Petri Nets and Industrial Applications: A Tutorial. Richard Zurawski, MengChu Zhou.

2 Overall description

This section describes the software and provides a brief explanation on how the system works.

2.1 What are Petri nets?

Petri nets are a graphical and mathematical modelling tool for describing concurrent and distributed systems. Some examples of their applications are workflow management, embedded systems or traffic control. The main advantages of Petri nets are their graphical notation, their simplicity on the semantics, and their rich theory for analysing their behaviour. However, using the Petri net graphical notation for understanding a complex system is quite hard, and thus a user-oriented visualization is required in a way that is understandable to users which are not necessarily familiar with Petri nets.

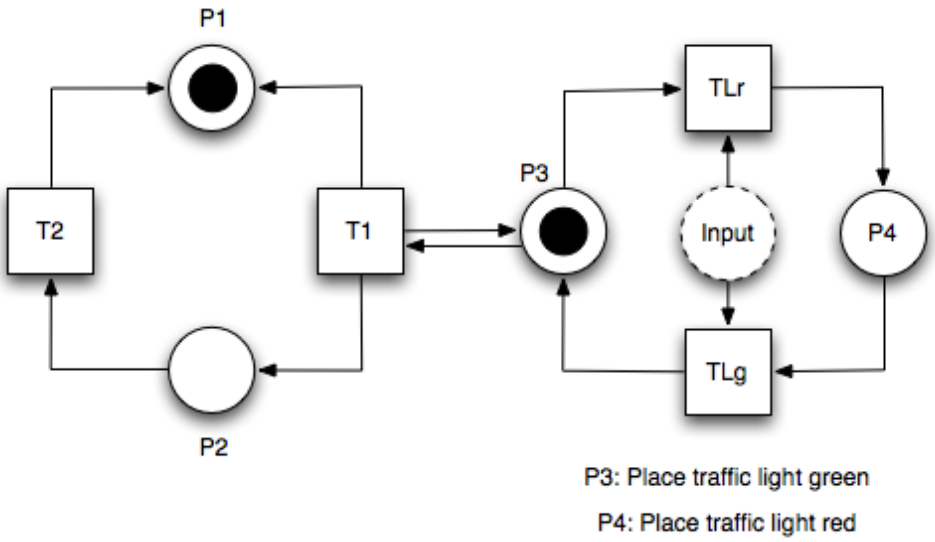

Figure 1: An example of a basic Petri net

Figure 1 presents an example of a basic Petri net. There are four different elements:

- Places: graphically represented by circles, represent conditions.
- Transitions: graphically represented by squares, represent events.
- Arcs: indicate which places are preconditions/postconditions for which transitions.
- Tokens: graphically represented by the dots, represent the elements that move in the Petri net along the places through the transitions.

Furthermore, we can see the example of Figure 1 as a model for a railway with a traffic light. The token in Place P1 represents a train moving on a railway. When this train arrives to Place P1, the transition T will fire if the conditions are met. The only condition for a transition to be fired is that a token should be present on each of the incoming places of the transition.

In this scenario, the conditions are not met, as the required token in Place Ptl is missing. The visualization of this scenario would be a traffic light with a red light. If Place I had a token on it, it would generate a token that will turn the traffic light to green and thus the train will move.

With the aim of creating a visualization of the scenarios for the above Petri net model, an application tool is needed in order to allow the user to define where each of the elements are represented in the 3D world (geometry editor) as well as how these elements are represented (shape) and how they behave (animation).

2.2 Adding geometry to Petri nets

The problem this project is tackled with is simple: We need a way to link the Petri net model to a 3D visualisation, this is, to add extra information so that it can be visualized. Once a Petri net model is created and its real life design is well-designed in the user's mind, what we call a "Geometry" and "Appearance" are created.

For this purpose, there is a need of a geometry editor which will be used to assign a two dimensional location to the elements defined in the Petri net, and of an appearance editor to take care of the shape of the elements.

2.3 Adding appearance to Petri net objects

Once the geometry problem is solved, a shape for each object should be defined. For instance, if places represent tracks, the shape, texture and other attributes should be linked to that object.

For this purpose, there is a need of an appearance editor which will be used to assign 3D visualization features to the elements defined in the Petri net.

2.4 Configuration

Before running the simulation, a definition of how the previous models are connected is needed. This is done in the configuration step as well as the validity check for the Petri net's connections to the geometry.

2.5 Simulating Petri nets

The next step in visualizing Petri nets is add descriptive visuals. With the information provided by the Petri net, geometry and appearance as defined in the configuration file, the simulation is set up.

Using 3D models, textures and animations, the Petri net becomes easier to understand for the user. Continuing with the railway example, tokens become trains that move on tracks, which are places. As the tokens move from place to place, the train is animated in the 3D visualization along the tracks.

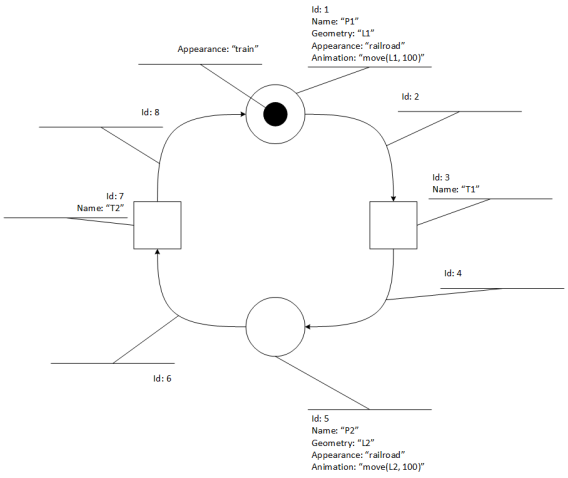

Figure 2: A simple extended Petri net

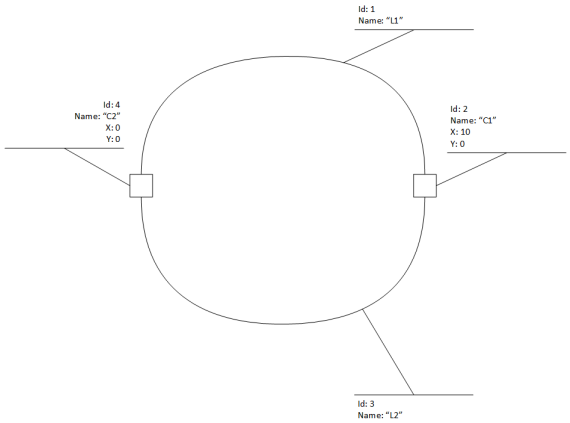

Figure 3: A simple geometry

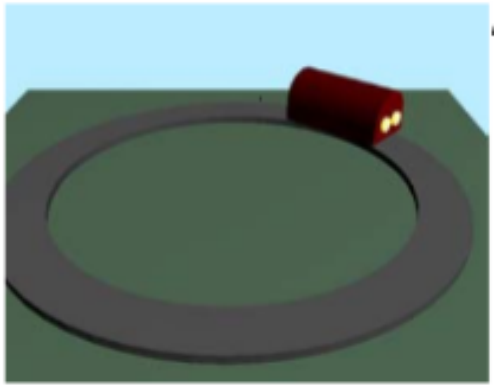

Figure 4: A simple 3D visualization

Figures 2 and 3 show how a simple 3D simulation of a train track and a train (Figure 4) is made out of Petri net model and a simple geometry. For example, in the Petri net model (Figure 2), Place P1 contains a geometry label that references its geometry location L1 in the geometry model (Figure 3).

2.6 General product description

The overall description is a brief introduction to how our software is designed. For this purpose, it contains a description of the actors involved in its use, and also a description of how the software works.

The following persons are all actors of our software:

- Petri net engineer: An engineer whose role is to design and develop a Petri net which suits the company needs.
  For instance, this could be an engineer working at a railway company and in charge of modelling the railway system using Petri nets.
- End user: An actor to whom the Petri net 3D-Visualization would be presented, or an actor who is in charge of presenting the Petri net to another.
  For instance, this could be a manager wanting to see what a Petri net represents.

The software is built around three different concepts:

- Editors: A component responsible for handling the creation and design of one of our models.
  For example: A Petri net editor to create a Petri net model.
  An editor is presented to the user as a GUI in an Eclipse window.
- Simulator: A simulator is, as its name says, a component capable of simulating a Petri net. The simulator is handling all the decisions regarding the Petri net and its behaviour. It then communicates the decisions to a 3D Engine responsible for the visualization.
- 3D Engine: A 3D Engine is responsible for the visualization of a Petri net in three dimensions. It receives input from the Petri net simulator, and also communicates to the simulator when a user performs a certain type of action.
  For example: the user clicks on a specific Place of the Petri net, this triggers a message from the 3D Engine to the Petri net simulator

Figure 5. shows the different components of our system and their connections.

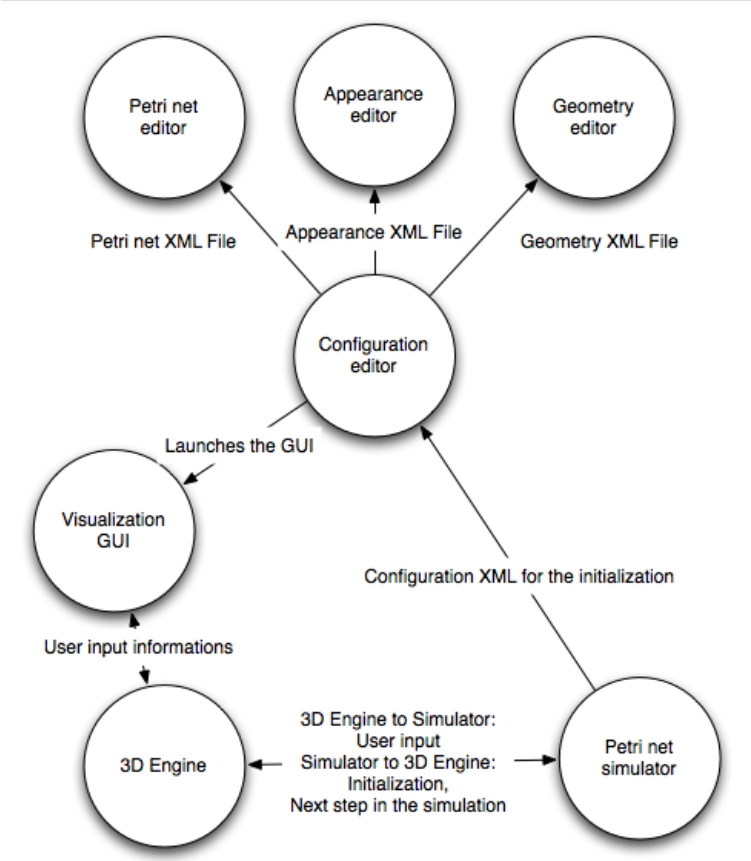

Figure 5: System design

Moreover, this project also introduces new concepts related to Petri nets, because as it has been said, we are creating an extension to Petri nets.

Regarding **Petri nets**:

- **Input Places**: An Input place is a place on which a user can create a token on Runtime, by interacting with the GUI in the form of a mouseclick. This token can then be used to trigger a transition for instance.
- **Geometry label**: A geometry label maps a place to a corresponding Geometry, defined later. A geometry label simply consists of an id.
- **Animation label**: An animation label describes the event happening on a place when a token is present. Animation labels refer to either movement actions or trigger actions.

A **Geometry** allows user to specify details about the 3D Space in which the Petri net is going to be rendered. It must present informations about the positions of the objects in the space but also informations about their looks.

- **Appearance Label**: An appearance label maps a Geometry to a 3D model. One can find informations about appearances by reading further in this section.
- **A position**: The position of a geometry is deduced by the relative positions of the geometry objects in the editor.

Finally, an **Appearance** file maps **Appearance Labels** to **3D models**. To this end, each appearance label used by a Geometry corresponds to a file on the file system. The file is expected to be a 3D model.

2.7 Basic functionality

This software allows the engineers to create a Petri net 3D Visualization using a combination of the different editors built for this project. The files created with each of the following editors are to be combined using the configuration editor:
Petri net editor, Geometry editor, Appearance editor.

In order to launch a simulation, an user should perform the following actions:

- Create a Petri net model using the **Petri net Editor**. This editor is used to configure the Petri net.
- Create a Geometry model using the **Geometry editor**. This editor is used to map objects from the Petri net model to a corresponding Geometry. Geometries include informations about positions/paths in the 3D space. They may also keep a reference to an Appearance to define their texture and shape.
- Create an Appearance file using the **Appearance editor**. This editor is used to link appearance labels to 3D models. A label corresponds to a specific 3D model file.

Once these three models have been created, the **Configuration editor** is used by the user to connect every model file previously created in order to be able to start a Simulation. To run the simulation, a specific "Run" button is added to the Configuration editor, which loads the Simulator.

The simulator then initializes the **3D Engine** with all the informations needed for the visualization. The 3D Engine then relies on the simulator to know which next move it should perform on the Petri net.
To interact with the simulation, the user can either click or press certain keyboard buttons in the GUI for the visualization.

3 System Features

In this section, the main components' features will be discussed, as well as the possible features that may be implemented in the future. It will mainly discuss the functionalities needed by each component in order for the project to work.
Use cases are provided in each section to provide informations on how each component will be used by the final user.

3.1 Petri net editor

Author: ***Albert***

The Petri net editor is a component that will extend the features provided by the *ePNK*, in order to fulfil the requirements of the project. The extended features include *Input Places* and the definition of links between geometry and animation components.

3.1.1 Functional Requirements

1. The Petri net editor **shall** allow the user to create, edit and delete Places.
2. The Petri net editor **shall** allow the user to create, edit and delete Tokens inside Places.
3. The Petri net editor **shall** allow the user to create, edit and delete Transitions.
4. The Petri net editor **shall** allow the user to create, edit and delete Arcs. An arc **shall** connect a Place to a Transition or vice versa.
5. The Petri net editor **shall** allow the user to define a Geometry label to a Place.
6. The Petri net editor **shall** allow the user to define an Appearance label to a Place.
7. The Petri net editor **shall** allow the user to define an Input Place label to a Place.
8. The Petri net editor **shall** allow the user to define an Animation label to a Place.
9. The Petri net editor **shall** allow the user to save and load a Petri net model.
10. The Petri net editor **shall** create a Petri net file in a format that can be read by the Simulator.
11. It **would be nice** that the Petri net editor allowed the user to undo and redo actions.
12. It **would be nice** that the Petri net editor allowed the user to copy and paste.

3.1.2 Use cases

The features are shown in Figure 6.

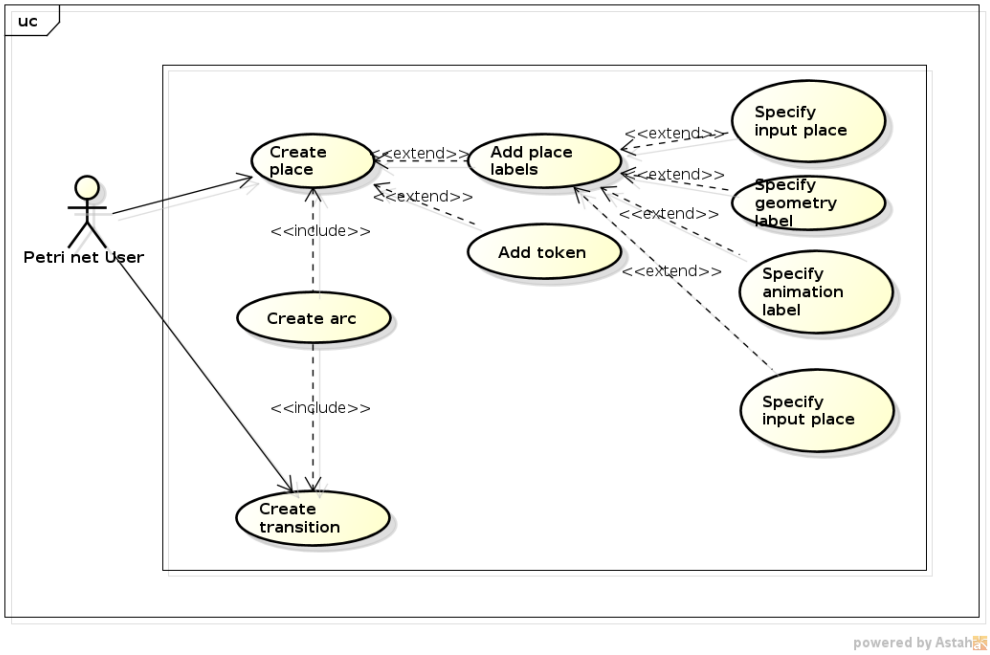

Figure 6: Use cases for the Petri net Editor

3.2 Geometry editor

Geometry editor is a tool to create and edit geometry. Geometry consists of geometry objects. They are either lines or points. Geometry editor has following requirements.

3.2.1 Functional Requirements

1. Geometry editor **shall** allow user to create, edit and delete points.
2. Geometry editor **shall** allow user to create, edit and delete define point as inputPoint, Connector or bendPoint.
3. Geometry editor **shall** allow user to create, edit and delete lines by using connectors and bendPoints.
4. Geometry editor **shall** allow user to load and save geometry file.
5. Geometry editor **shall** allow user to define labels for geometry objects.
6. Geometry editor **shall** create unique IDs for geometry objects.
7. Geometry editor **should** allow user to create parametric curved lines.
8. Geometry editor **should** allow user to use undo and redo.
9. Geometry editor **should** allow user to use copy, paste and cut geometry objects.
10. Geometry editor **should** have a Graphical User Interface.

(a) Geometry editor **should** have drag and drop interface.

(b) Geometry editor **should** enable editing of geometries with a text input.

(c) It **would be nice** if the geometry editor allows user to create different kinds of parametric curved lines, such as Catmull-rom spline and Bézier-curves.

(d) It **would be nice** if the geometry editor allows user to zoom and pan the geometry canvas.

(e) It **would be nice** if the geometry editor allows user select multiple geometry objects simultaneously.

(f) It **would be nice** if the geometry editor allows user to load multiple geometries on same canvas.

(g) It **would be nice** if the geometry editor allows user select multiple geometry objects simultaneously.

(h) It **would be nice** if the geometry editor allows user to rotate and scale geometry objects.

(i) It **would be nice** if the geometry editor allows user to toggle visibility of geometry objects while using geometry editor.

11. It **would be nice** if you could specify a Petri ned model from which the geometry is being specified from and it is validated.

3.2.2 Use cases

The uses of geometry editor is summed up in the use case diagram in Figure 7.

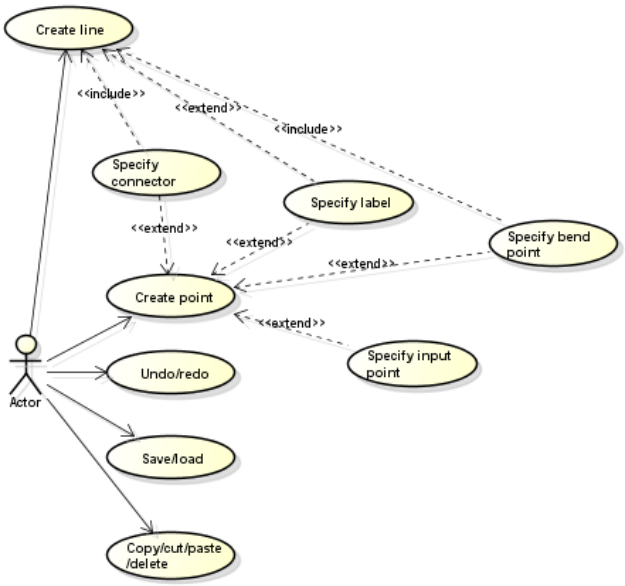

Figure 7: Use cases for the Geometry editor

3.3 Appearance Editor

The Appearance Editor is a component that will allow the user to add visual information such as shapes and textures to the Petri net elements to be simulated. The connection between the appearance files and the Petri net elements will be done via the *appearance label*.

3.3.1 Functional Requirements

1. The Appearance Editor **shall** allow the user to link appearance labels to 3D objects by choosing a predefined file.
2. The Appearance Editor **shall** allow the user to link appearance labels to textures by choosing a predefined file.
3. The Appearance Editor **shall** create an Appearance file in a format that can be read by the Configuration Editor.
4. The Appearance Editor **shall** allow the user to load an Appearance file.
5. The Appearance Editor **shall** allow the user to save an Appearance file.
6. It **would be nice** to allow the user to load a Petri net file in order to retrieve all the appearance labels.

3.3.2 Use cases

The features described above are shown in Figure 8.

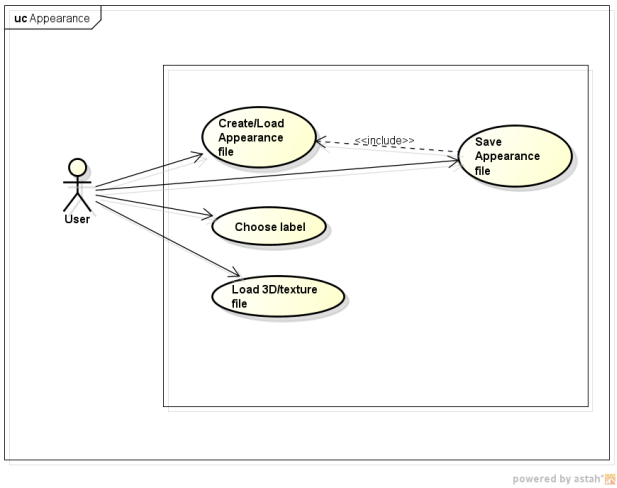

Figure 8: Use cases for the Appearance Editor

3.4 Configuration Editor

Author: ***Albert***

The configuration editor is a component that will work as a connector between the Petri net editor (Section 3.1), the Geometry editor (Section 3.2) and the Appearance editor (Section 3.3).

3.4.1 Functional Requirements

1. The configuration editor **shall** allow the user to input a Petri net file containing its model.
2. The configuration editor **shall** allow the user to input a Geometry file containing its model.
3. The configuration editor **shall** allow the user to input an Appearance file containing its model.
4. The configuration editor **shall** allow the user to start a simulation with the referenced Petri net, Geometry and Appearance models.
5. The configuration editor **shall** allow the user to validate the data.
6. It **would be nice** if the configuration editor to save and load a specific configuration to a file.

3.4.2 Use cases

The features are shown in Figure 9.

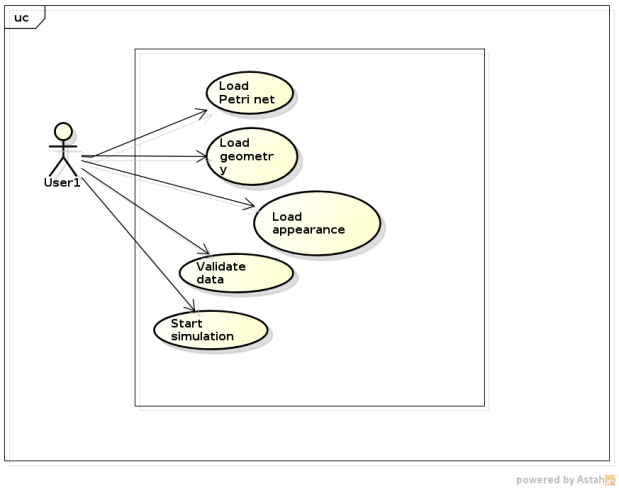

Figure 9: Use cases for the Configuration Editor

3.5 3D Simulator

The 3D Simulator will show the animated Petri net, while allowing the user to control a few things. The functionality of the 3D simulator can be specified using the following statements.

3.5.1 Functional Requirements

1. The simulator **shall** allow the user to play the simulation.
2. The simulator **shall** allow the user to pause the simulation.
3. The simulator **shall** allow the user to reset the simulation.
4. The simulator **shall** allow the user to add tokens on input places.
5. The simulator **shall** allow the user to exit the simulation.
6. The simulator **should** allow the user to change the orientation of the view.
7. It **would be nice** if the simulator allowed the user to take screenshots.
8. It **would be nice** if the simulator allowed the user to forward the simulation.
9. It **would be nice** if the simulator allowed the user to rewind the simulation.

3.5.2 Use cases

The features described above are shown in Figure 10.

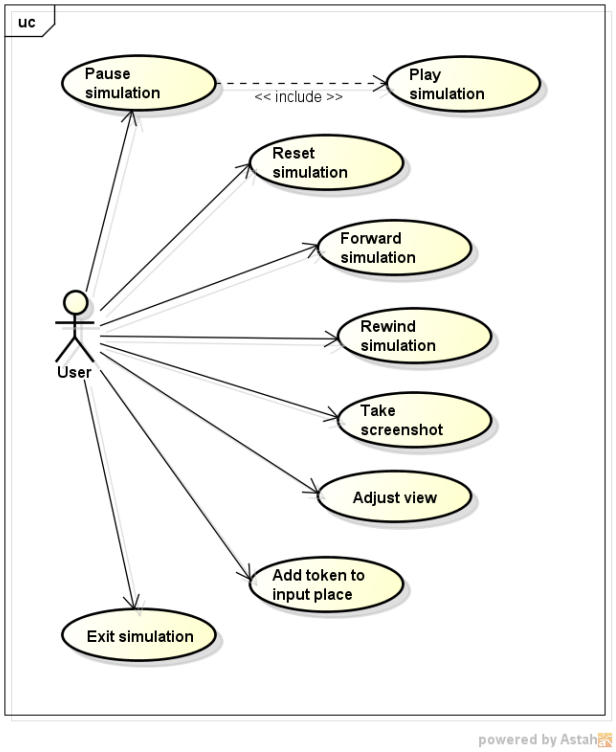

Figure 10: Use cases for the 3D Simulator

4 Non functional requirements

In addition to programming related requirements, number of requirements is identified in the following sections.

4.1 Implementation constraints

The software must be implemented as plug-in to the Eclipse framework. The software is based on following technologies:

1. Eclipse Modeling Framework, EMF v. 2.91
2. Graphical Modeling Framework, GMF 3.91
3. Graphical Editing Framework, GEF 1.70
4. JMonkey v. 3.0

4.2 Documentation

Development must be documented and documentations must be delivered according to following delivery deadlines

1. Project definition, week 39
2. UML diagrams, week 41
3. System specification, week 44
4. Handbook, draft, 47
5. Test documentation, 51
6. Final documentation, 51

4.3 Quality Assurance

Procedures insuring the quality of this software project are presented in the following segment.

1. Quality of documentation
   (a) Each part of the document is produced by multiple persons to decrease amount of typical errors.
   (b) Each part of the documentation is reviewed for feedback. Depending on feedback appropriate measures are taken to edit the documentation.
   (c) Internal deadlines for documentation are set, so that there is a time period allocated for internal reviews before the real deadline.
2. Quality of implementation

(a) Each implementation is reviewed by multiple persons to decrease amount of typical errors.

(b) Commenting on manually generated code shall use javadoc standard.

(c) A test strategy shall be developed to test the implementation. The strategy consists of unit test, acceptance test and system testing.

5 User Interface

The following section will describe how the Graphical User Interface (GUI) of the Petri net editor, Geometry editor, Appearance editor, Configuration editor and the Simulation tool should look like. How to execute different tasks in the editors such as creating and edit geometry and Petri nets, will be described in the following handbook. The different editors will strive towards being intuitive for the targeted user, e.g. the Geometry editor should be easy for the non-technical user to use, while the Petri net editor should be intuitive for the technical user. We are not that far in the actual implementation process to show concrete examples of the GUI. Instead illustrations will be shown of the expected GUI.

5.1 Technologies

The editors will be implemented as extensions to Eclipse, which allows us to use the different functionalities that comes with Eclipse. The functionalities listed below are all part of Eclipse and will be used for this software.

- **Editor:** Used for the Petri net editor and the Geometry editor. Gives the user the ability to create and edit Petri nets and geometries.
- **View:** Used to show properties of the selected object in the editor and to show the project files.
- **Dialog:** A pop-up window used when the user has to specify a property of an object or to select textures or 3D files in the Appearance editor.
- **Wizard:** A pop-up window with several pages that guides the user to fill out information correctly. Used to set up the configuration for the simulator.

The use of these functionalities will all be explained further in the handbook.

5.2 Graphical User Interface

This section will describe the GUI for the editors and the simulation tool.

5.2.1 Petri net editor & Geometry editor

The Petri net editor and Geometry editor are very much alike. They got the same composition and only the objects that can be drawn on the canvas are different. The Petri net editor and the Geometry editor consist of a toolbar, a view with project files, a view with the properties of the selected object, a canvas and a object toolbar belonging to it. Illustration of the Petri net editor can be seen in figure 11 and an illustration of the Geometry editor can be seen in figure 12.

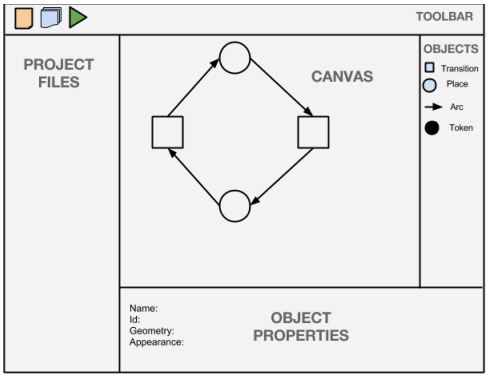

Figure 11: An illustration of the Petri net editor GUI.

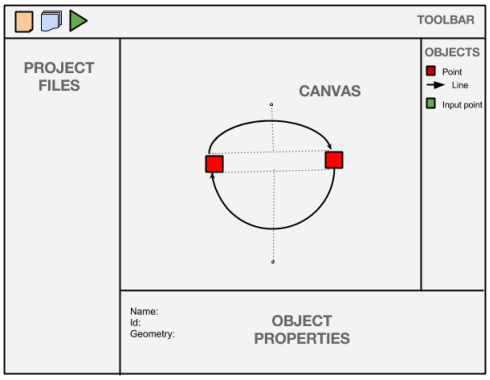

Figure 12: An illustration of the Geometry editor GUI.

5.2.2 Toolbar

The toolbar on top allows the user to **Load** and **Save** Petri nets as well as Geometries. The **Play** button allows the user to run the simulation tool anytime he or she wants given that the configuration is set up correctly.

5.2.3 Project files

On the left side the user will have a view where all the files associated to the project are shown. The view will be able to identify the filetype of each file and when the user opens one of the files, the editor associated to the filetype will open accordingly.

5.2.4 Canvas

The canvas will be where the user can draw the Petri net and Geometry respectively. The user drags an object from the Object toolbar and places it on the canvas. The user can edit the position of the different objects by dragging them around on the canvas. Furthermore the connections between the objects can be edited by dragging lines between two objects (while following the rules of the Petri net). The user will be able to see the graphical representation of the Petri net in the Petri net editor and the graphical representation of the geometry in the geometry editor.

5.2.5 Objects

To create and modify the Petri net and the geometry, the user will be able to drag objects from the objects toolbar. The object toolbar in the Petri net editor only has the objects that belongs to Petri net, and the same goes for the geometry editor. As mentioned the user can drag these objects onto the canvas to insert the specific object in the Petri net or geometry. Each object will have a dedicated icon, which will increase usability.

5.2.6 Object properties

Each object has different properties that can be defined by the user. The user can modify these properties in the view with the object properties whenever an object is selected. The properties of the different objects in each editor is specified in the System Features.

5.2.7 Appearance editor

Figure 13 is an illustration of how the appearance editor will look like. The user has specified in the geometry/petrinet editor the appearance of the object. The appearance can either be a texture to a surface or a 3D object. When the user has specified the appearance with a dedicated label, the user will be able, in the appearance editor, to specify which files should be associated with each label. The user can choose the specific file from the project files.

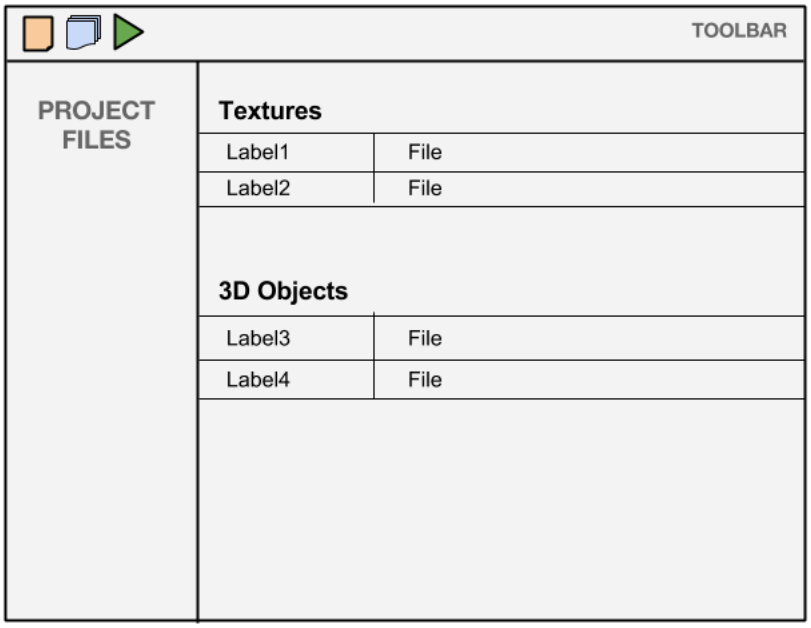

Figure 13: An illustration of the Appearance editor GUI.

5.2.8 Configuration wizard

The configuration wizard tool helps the user to set up the simulation. The configuration wizard will be a pop-up window with several pages where the user has to specify a file for: Petri net, the

geometry and the appearance. The wizard will then validate whether these files match each other. If they do, the user will be able to run the 3D simulation.

5.2.9 Simulation tool

When the user has run the Configuration wizard successfully he will be able to run the 3D simulation. The 3D simulation opens in a new window. This window has a **play/pause** button that changes according to the play state; if the simulation state is play the button will be a pause symbol and vice versa. The window also has a **reset** button that allows the user to reset the simulation. The user will be able to interact with the simulation, therefore the user will be able to use the mouse to click on the different objects the user can interact with. The user will also be able to use the mouse to change the camera perspective.

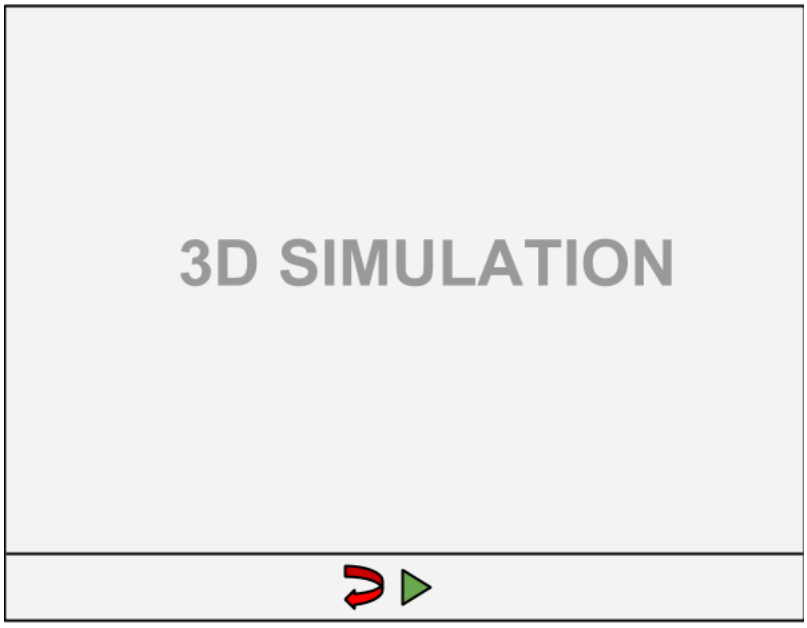

Figure 14: An illustration of the Simulation tool.

5.3 Handbook

Since the implementation of this software is only in the beginning stages, the following topics will be an estimation of how the mentioned tasks will be executed. The actual implementation of the software will strive towards being able to do the tasks in this handbook. The handbook is a list of typical questions a user will have to the most important parts of this software. A step-by-step guide to solve each of these questions is provided to the reader.

5.3.1 How do I create a Petri net?

In the Project files explorer Right-click and choose New > Petri net File ... .
A pop-up window will appear where the user can specify the file name.
The new Petri net file will appear in the Project files explorer.

5.3.2 How do I create a geometry?

In the Project files explorer Right-click and choose New > Geometry File ... .
A pop-up window will appear where the user can specify the file name.
The new geometry file will appear in the Project files explorer.

5.3.3 How do add a transition/place to the Petri net?

Select the transition/place object from the object toolbar on the right. Click and drag the transition/place to the canvas.
The transition/place will now appear on the canvas.
The user can change the position of the object by click and drag to the new position.

5.3.4 How do add a point to the geometry?

Select the point object from the object toolbar on the right. Click and drag the point to the canvas.
The point will now appear on the canvas.
The user can change the position of the point by click and drag to the new position.

5.3.5 How do I create a connection between two objects?

Select the arc/line object from the object toolbar on the right.
Click on the object that is the source and drag the mouse to the target object.
The arc/line will now appear on the canvas.

5.3.6 How do add a bendpoint to a line?

Select the line you want to add a bendpoint to.
Right-click on the canvas where you want the bendpoint to be and select **Add bendpoint**.
The position of bendpoint can be changed by dragging it with the mouse.

5.3.7 How do I add an appearance to an object?

Select an object on the canvas in the Petri net editor / Geometry editor.
In the object properties view in the bottom of the editor, change the name in the appearance label for the object.

Open the appearance editor. Find the appearance label given in the last step.
Click on the **File..** button next to the label.
A pop-up window will appear with a list of the project files. Choose the file you want for the appearance (e.g. a texture file or a 3D object file).
When you run the simulation the texture or 3D model will appear for the specific object.

5.3.8 How do I delete an object in the Petri net editor and Geometry editor?

Select and mark the object in the canvas and press Delete.

5.3.9 How do I save a Petri net model / a geometry model?

In the editor - click on the **Save** button in the top toolbar.

5.3.10 How do I load a Petri net model / a geometry model?

In the project files explorer - double-click on the model file you want to open.
The Petri net editor and geometry editor will open respectively.

The model can also be opened by clicking the **Load** button in the top toolbar. A pop-up window will appear with the project files. Select the file you want to open from the list of project files and press Load.

5.3.11 How do I run the Configuration wizard?

Click on the **Configuration** button on the top toolbar. A pop-up wizard will appear. On the first page in the wizards, select the Petri net file from the project files. On the next page select the geometry model file from the project files. On the third page select the appearance model file.
The wizard will now validate whether the 3 files match. If they do the configuration is complete.

5.3.12 How do I run the 3D simulation?

When you have run the Configuration wizard (see steps in section 5.3.11) correctly, you will be able to run the simulation.
Press the simulation button in top toolbar to run the 3D simulation.

6 Architecture

Author: ***Anders, Albert and Monica***

The architecture of this **Extended Petri net Simulator** has been designed as modular, meaning that the system is divided in different components which are connected through interfaces. In this way, components are independent of each other and can be developed by different members of the team. In addition, the modular design is an advantage for easily testing each component as well as the entire system in a hierarchical structure.

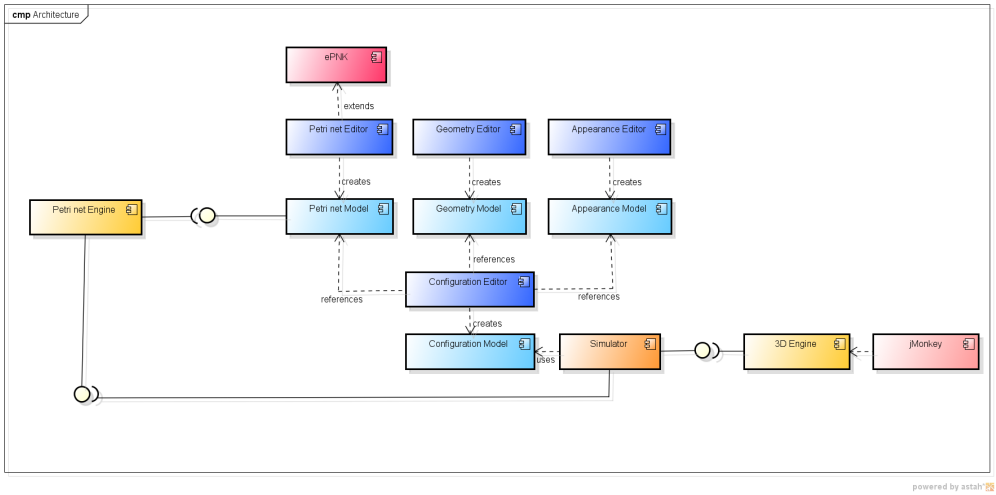

Figure 15: Architecture of the System

6.1 Petri net editor

Author: ***Albert***

The Petri net editor is a component which consists of an extension of the ePNK editor. It contains all the functionality that ePNK provides and also adds the features described previously in this document, and that can be summarized in Figure 16. Its actual implementation is shown in Figure 17.

6.1.1 Petri net editor classes

A description of the classes shown in Figure 17 is provided.

Place The class Place extends from Place in the *pnmlcoremodel*, and contains the following attributes:

- **Token**: an ePNK string label to indicate the presence of one or more tokens in this Place.
- **AppearanceLabel**: an ePNK string label to indicate the shape and texture of the Place.

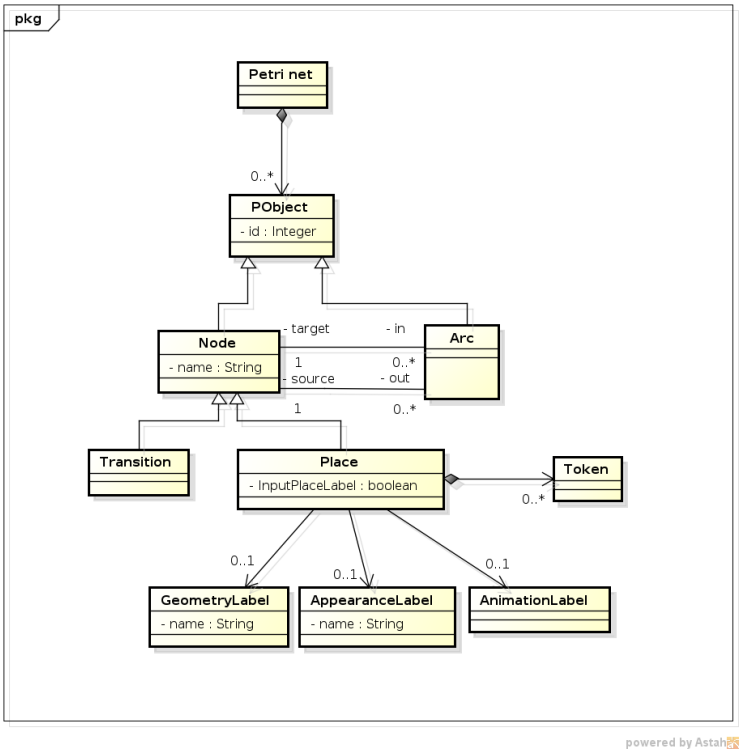

Figure 16: UML for the Petri net Editor

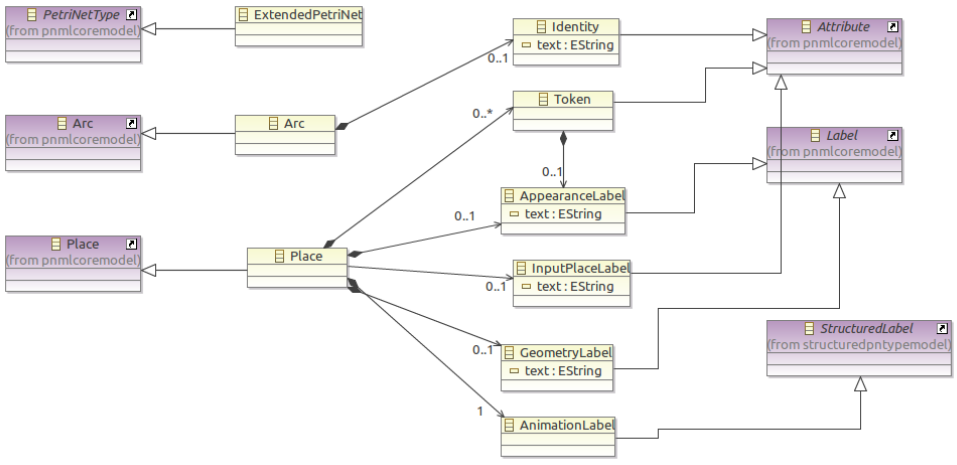

Figure 17: Petri net editor domain model

- **InputPlaceLabel**: an ePNK boolean attribute to indicate if the place is interactive for the user or not (if tokens can be added in runtime by the user).
- **GeometryLabel**: an ePNK string label to indicate the geometry location of the Place in the simulation.
- **AnimationLabel**: : an ePNK structured label which contains an Animation object and that contains all the information of the animation or sequence of animations executed when a Token is in this Place.

Arc The class Arc extends from Arc in the *pnmlcoremodel*, and contains the following attributes:

- **Identity**: an ePNK integer label to indicate the id of an arc, which is required for proper visualization of the Token "movement" during the simulation.

6.2 Geometry editor

Geometry editor is used to define lines and points in two dimensional plane. These geometries are then used to define shape of the visualized simulation. Figure 18 describes the geometry model used by the geometry editor.

Geometry The Geometry class is an abstract definition of a geometry object.

GObject The GObject is a geometry object, which can be either a line or a point. It has attributes **name** and **id**. **name** is used to name individual geometry objects, so that user can easily tell them apart. **id** is used to differentiate geometry objects from each other.

Point Point is a location in two dimensional plane. It has attributes **xLocation** and **yLocation** which refer to horizontal and vertical coordinates respectively.

InputPoint Class InputPoint is the point that petri net model place can have as geometry.

Connector Class Connector is either beginning or ending point of a line. Therefore lines that ends at the Connector are **in** and the lines that starts from Connector are **out**.

BendPoint Class BendPoint is a point used to turn lines into curved lines.

Line Class Line is a parametric curve in a two dimensional plane. The line begins at a point location **begin** and it ends at a point location **end**. **bendPoint** is a point location which defines the curve of the line, unless it doesn't have one, thus the line would be straight.

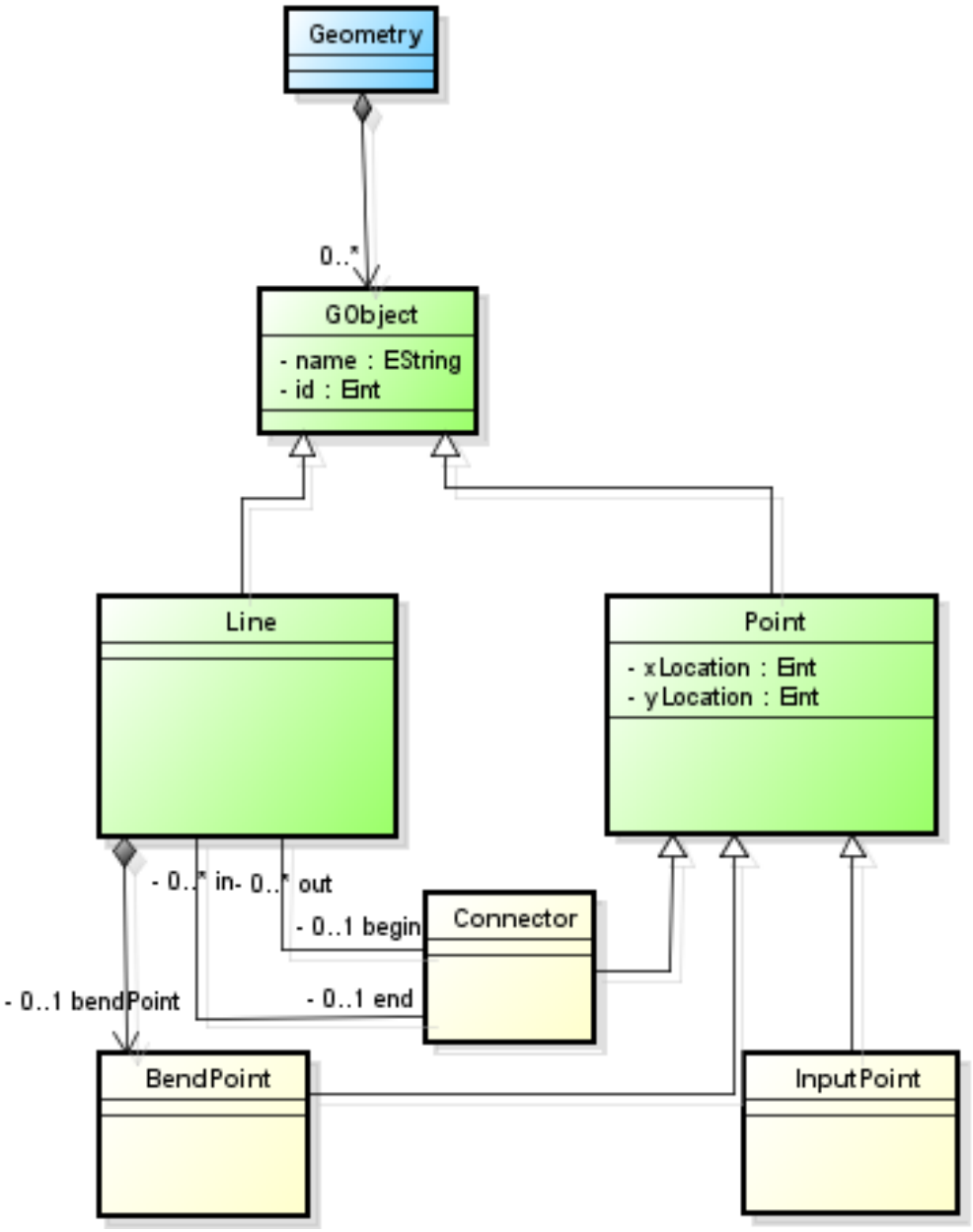

Figure 18: Geometry domain model

6.3 Appearance editor

Author: ***Monica***

The Appearance Editor is the component which allows the user to associate a visual representation to the Petri net elements so that they can be represented in the 3D simulation. The editor can be used by both technical and non-technical users as it only implies linking *Appearance Labels* previously defined in the Petri net with predefined 3D or texture files.

Figure 19 shows the model used by the Appearance Editor component.

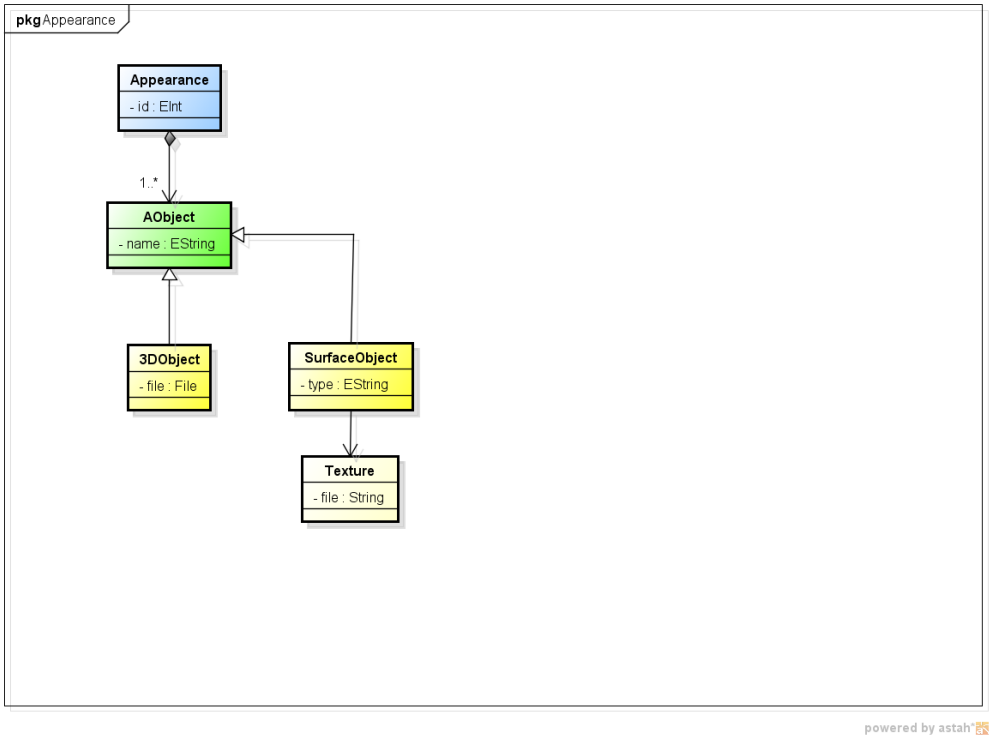

Figure 19: Appearance Domain Model

6.3.1 Appearance Editor classes

Next, the model will be described into more detail.

Appearance The Appearance class is simply an abstract definition of an Appearance object.

AObject The AObject class is general class from which different appearance objects will inherit. Its only attribute, **name**, refers to the name of the object that is represented by this appearance (e.g. *"train"*).

3DObject The 3DObject class consists of only one attribute,**file**, of type String, which represents the path of the 3D model on the hard disk. Also, 3DObject will inherit from AObject.

SurfaceObject The SurfaceObject class consists of only one attribute, **type**, of type String, which represents the shape of the object to be visualized (e.g. *"rectangular"*). SurfaceObject will also inherit from AObject.

Texture The Texture class consists of only one attribute,**file**, of type String, which represents the path of the texture model on the hard disk.

6.4 Configuration editor

Author: ***Monica***

The Configuration Editor is the component which allows the user to create a link between all the models that have been previously created by using the other editors. The editor can be used by both technical and non-technical users since its main functionality is connecting the Petri net, the Geometry and the Appearance models.

Figure 20 shows the model behind the Configuration Editor.

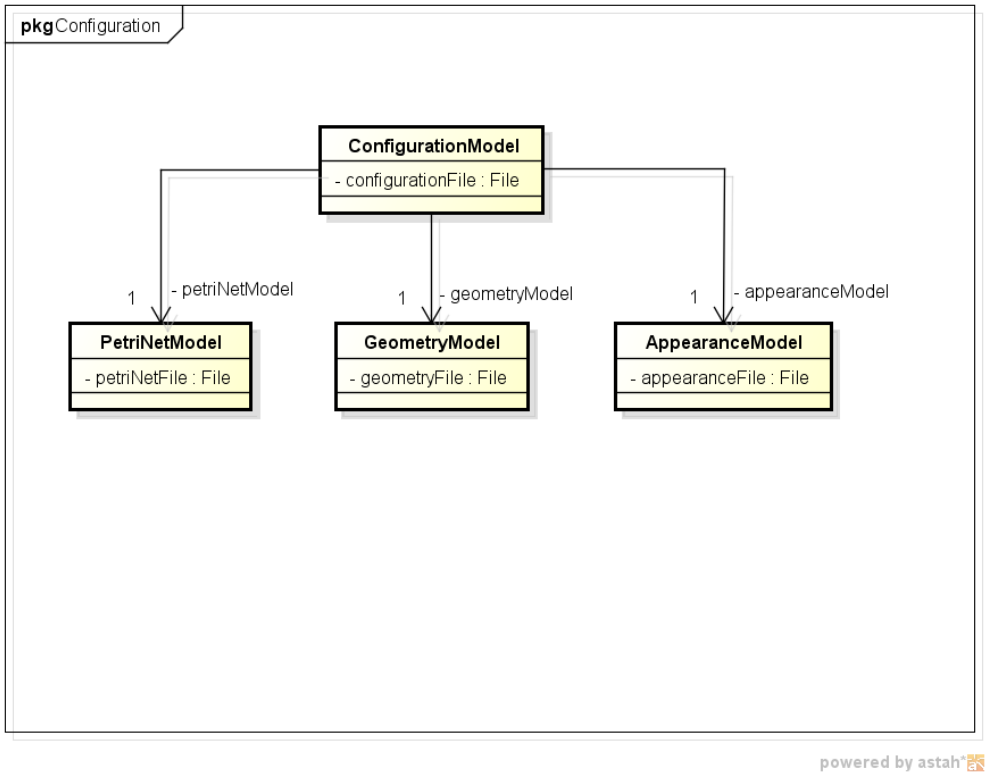

Figure 20: UML for the Configuration Editor

6.4.1 Configuration editor classes

Next, a more detailed description of the model is provided.

ConfigurationModel The ConfigurationModel class has only one attribute, **configurationFile**, of type File, which will include all the information provided by the *PetriNetModel*, *GeometryModel* and *AppearanceModel* objects.

PetriNetModel The PetriNetModel class has only one attribute, **petriNetFile**, of type File, which represents the output file of the Petri net Editor. This file will include the description of all the objects in the petri net and their attributes.

GeometryModel The GeometryModel class has only one attribute, **geometryFile**, of type File, which represents the output of the Geometry Editor. This file will include the description of all geometry objects and their attributes.

AppearanceModel The AppearanceModel class has only one attribute, **appearanceFile**, of type File, which represents the output of the Appearance Editor. This file will include the associations between appearance labels of the Petri net objects and their corresponding 3D objects or textures.

6.5 Petri net engine

6.6 Simulator

The simulator is the main component of the system, it reads the initial configuration and works as a controller between the Petri net engine and the 3D engine.

In order to implement this, we will use the Observer pattern (Figure 21) so that the Simulator can keep track and receive notifications from the 3D engine when some events happen, such as an animation finished or a user interaction.

Figure 22 shows the interaction between the three components involved. The simulator starts by reading the configuration, and initializing the Petri net engine with the initial Petri net model as well as the 3D engine with the geometry and the appearance.

Then, the 3D engine tells the Simulator that has received a *start simulation* input from the user, and thus, awaits for new animations to be played. The simulator asks the Petri net engine to fire transitions and to give back the next sequence of animations to be played by the 3D engine.

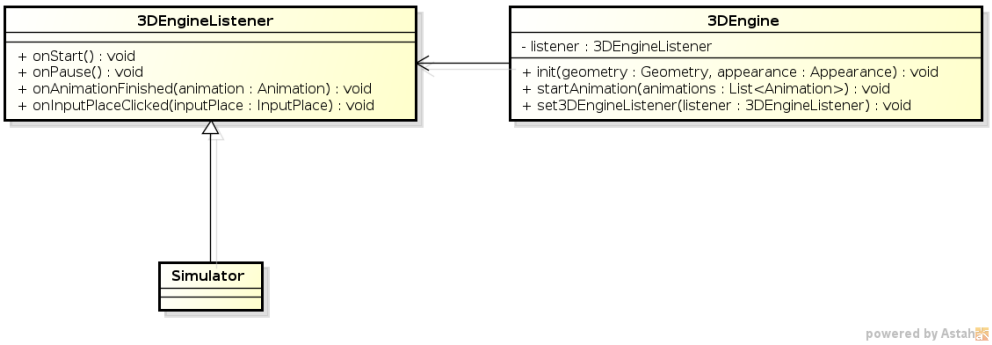

Figure 21: Simulator with the Observer Pattern

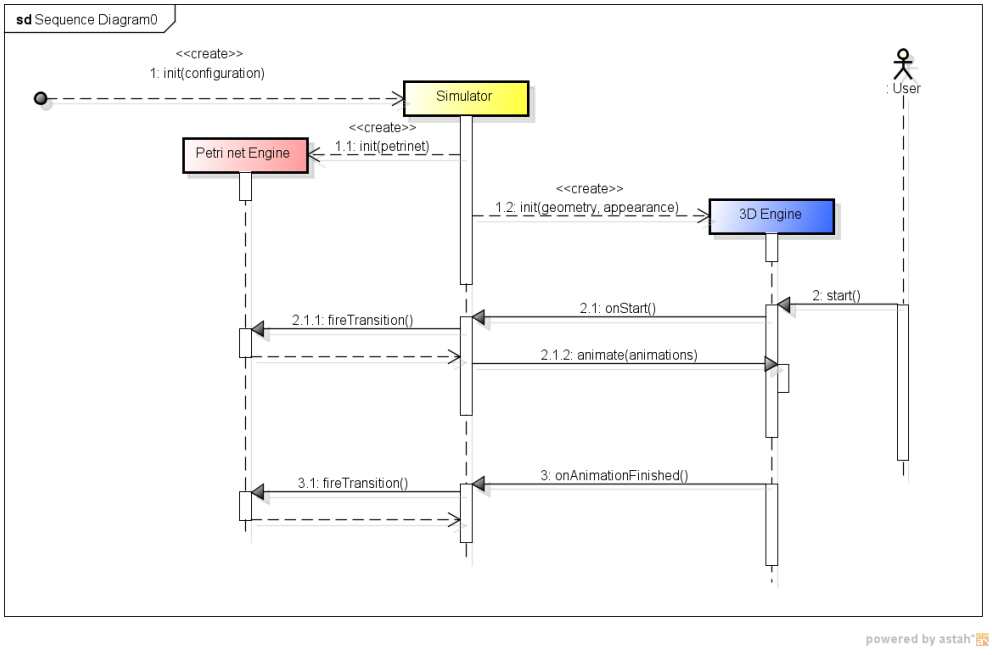

Figure 22: Sequence diagram of the Simulator, Petri net Engine and 3D engine

6.7 3D Engine

The 3D Engine is the component of the application that handles the animation of objects and displays it on the screen for the user. Its main functions are:

- handling the window in which the graphical simulation is shown to the user
- take care of the user interactions with the window buttons (*Play, Pause, Stop*).
- take care of the user inputs in the case of *Input Places*
- draw the graphical objects
- maintain the communication with the *Simulator*
- update the animations according to the information sent by the *Simulator*

In order to build a system that is not technology dependent in terms of its 3D visualization, Interfaces and the Factory design pattern are employed. In this way, the 3D Engine can be easily changed from **jMonkey** to **Java3D** or any other technology desired at some point by the customer.

Figure 23 shows the defined classes in the implementation of the Factory pattern.

Engine3DFactory The Engine3DFactory class will be in charge of creating 3D Engine classes according to the engine name sent as a parameter of the method *getEngine3D()*. This method will be called from the Simulator in the initialization phase, when both the 3D Engine and the Petri net Engine are created.

Engine3D Engine3D is an Interface which will be implemented by the actual 3D Engine of the applicaton, in our case **JMonkey**. The following methods will be provided by the Engine3D interface:

- *init(Geometry geometry, Appearance appearance)* - used to initialize the geometry and appearance models;
- *onStart()* - called when the START button is pressed;
- *onStop()* - called when the STOP button is pressed;
- *onUserClick()* - called when the user clicks on an Input Place;
- *animate(List¡Animation¿ aniamtions* - used by the Simulator to send the list of animations to be performed by the 3D Engine;
- *onAnimationFinished(Animation animation* - used by the 3D Engine to let the Simulator know an animation has finished.

JMonkeyEngine3D As stated before, the JMonkeyEngine3D class will implement all the methods of the Engine3D interface and will perform all the animations using the **jMonkeyengine 3.0** technology.

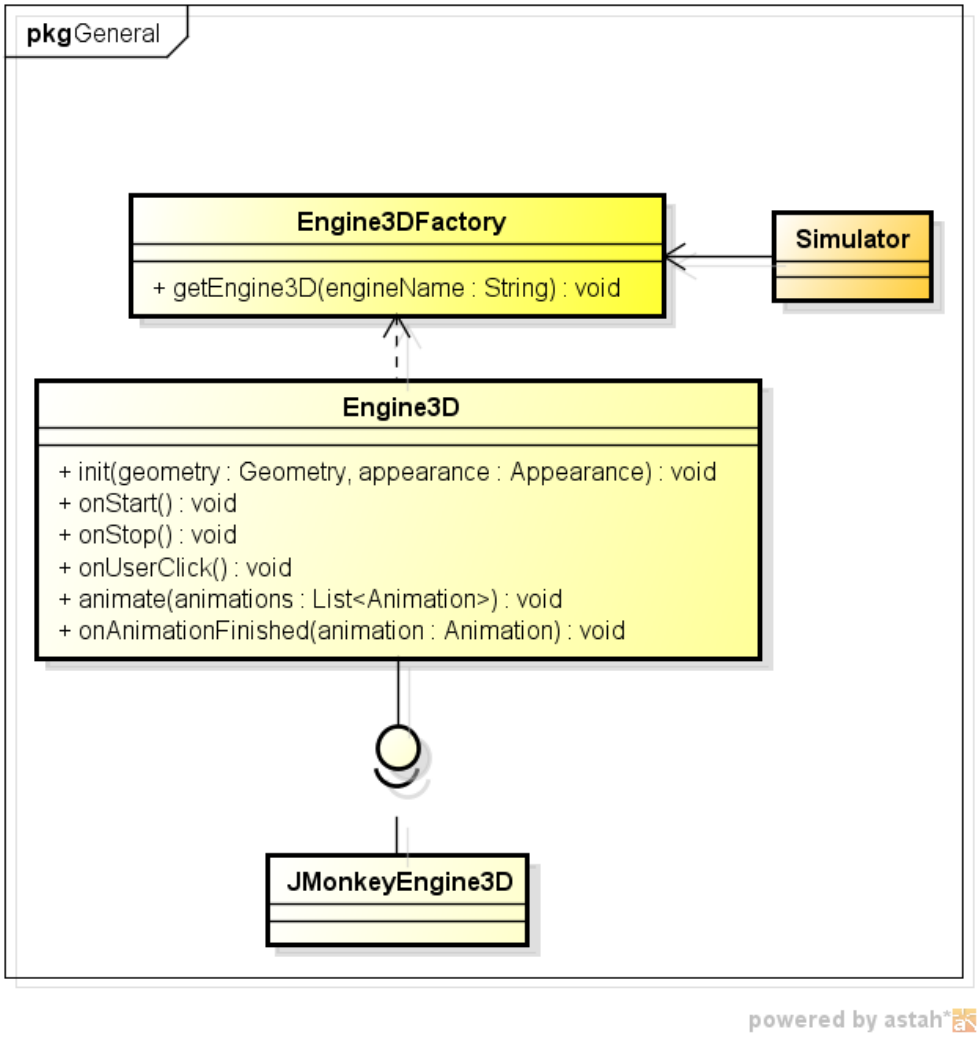

Figure 23: 3D Engine Factory Pattern

6.8 jMonkey

The jMonkeyEngine 3.0 provides libraries that ease the implementation process. When visualizing the paths defined in the configuration file, a jMonkey class is used. The class has a waypoint system with different types of smoothing available for use. This waypoint system integrates natively with another jMonkey class that is used to animate objects along paths. The animation system lets the program play, pause and reset the animation of objects along the path. Also, the current position and play state of the animation is always available. These functions are essential to the simulator, and saves time in the implementation process. Other examples of less essential functions are: real time camera control, keyboard input mapping and listeners, physics simulation and collision detection.

7 Glossary

GUI Graphical User Interface

Petri net A mathematical and graphical model for the description of distributed systems. It is a directed bipartite graph, in which nodes represent transitions and places. The directed arcs describe which places are pre- and/or postconditions for which transitions. [source wiki]

Synchronization Transition finings are synchronized on the occurrences of external events, such as when animation is finished or user triggers the transition.

Token Petri Net element which moves along the Petri net places through transitions.

Use case A list of steps defining interactions between a role (e.g. "Technical user"), also known as an actor, and a system for achieving a goal. The actor can be a human or an external system

3D Three dimensional.

ePNK Eclipse Petri Net Kernel.

Bendpoint Bendpoint is a point, which functions as a control point for parametric curves allowing "bending" of said curve.

Bézier-curve Bézier-curve is a parametric curve, where shape is defined by blend of different control points.

Catmull-rom Catmull-rom splines are a family of cubic interpolating splines formulated such that tangent at each point of the spline is calculated using the previous and next point on the spline. (http://www.cs.cmu.edu/ 462/projects/assn2/assn2/catmullRom.pdf)

Connector Connector serves as either first or the last point of line.

Geometry object Geometry object can be either point or line.

Input Point Input Point is a point, where user is allowed to add tokens during the simulation.

Javadoc standard Format used by Javadoc is the industry standard for documenting Java classes.

Parametric curve Parametric curve is a mathematical curve defined by point locations.

Point Point is a coordinate location in a two dimensional plane.