System Specification

Extended Petri net 3D Simulator

Group A

December 20, 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 modeling, formal analysis, and design of discrete event systems.
>
> *Zurawski* : 1994

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 modeled 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 visualization from a Petri net, to allow non-Petri net experts to actually 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 visualization possible. For this purpose, we have thought of a software solution that creates a simple link between a Petri net and a 3D visualization.

This document will describe in details how we are going to make this software. The next section will describe the overall purpose of the software and explain further about Petri nets. In section 3 and 4 the features of the system features and requirements will be described. Section 4 will describe how the graphical user interfaces of the software will look like and how will our targeted audience use the software. Lastly, section 6 will provide the reader with information regarding the product architecture and also a description of the models for the different components of the final product.

1.1 Audience

The audience of this document consists of persons affiliated with the company that models and simulates Petri nets, developers of the system and final users of the Extended Petri net 3D Simulator.

2 Overall description

Author: ***Thibaud, Albert***

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 modeling tool for describing concurrent and distributed systems. Some examples of their applications are work flow management, embedded systems or traffic control. The main advantages of Petri nets are their graphical notation, their simple semantics, and their rich theory for analyzing their behavior. However, using the graphical notation of Petri nets 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.

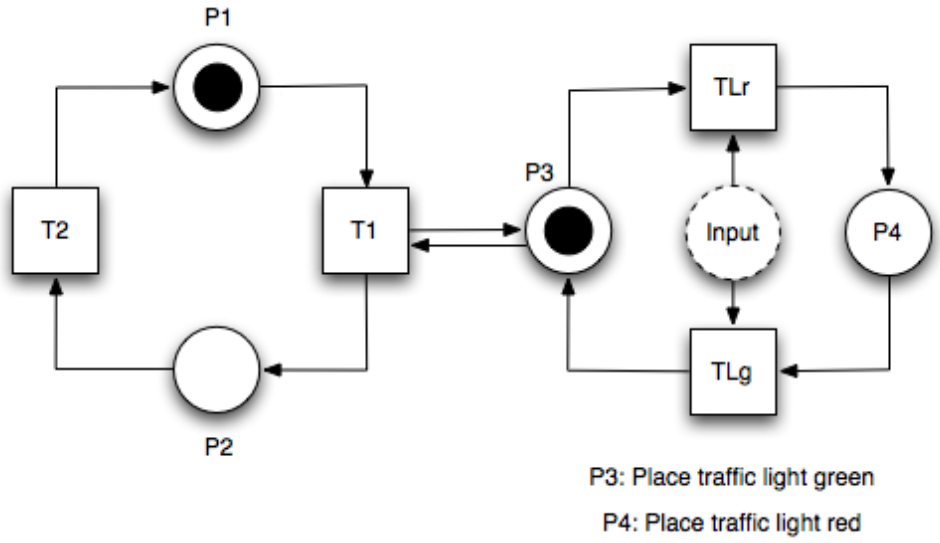

Figure 1: An example of a basic Petri net

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 are created and deleted in the Petri net along the places through the transitions.

As an example we can see 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 T1

will fire if there is also a token in Place P3. 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 met, as the required token is in Place P3. The visualization of this scenario would be a traffic light with a green light. If instead Place P4 had a token on it and no token in P3, it would be a red light and the train would not be able to 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 visualization, 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.

Once a Petri net model is created the user will be able to add a geometry and an appearance to it, in order to specify how the Petri net will look in the 3D visualization.

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 for the Petri net is defined, a shape for each object should be defined. For instance, places represent tracks and the 3D object or a shape and a texture should be linked to that place. The same applies to the transitions and tokens.

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.

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.

2.4 Configuration

Before running the simulation, a definition of how the models discussed above 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

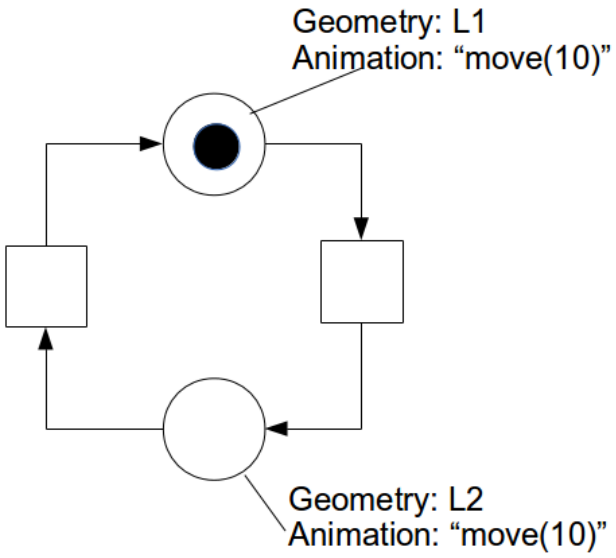

Figure 2: A simple extended Petri net

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.

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).

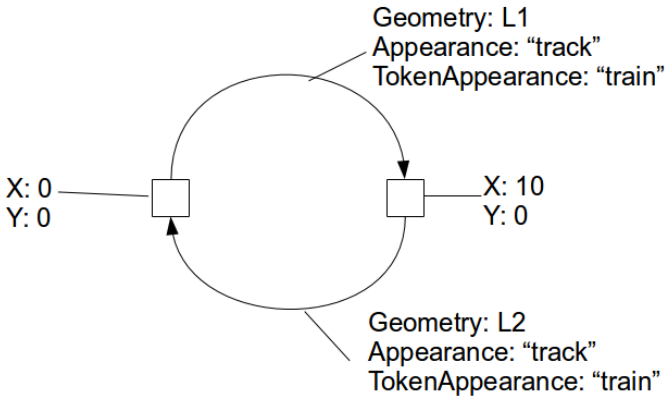

Figure 3: A simple geometry

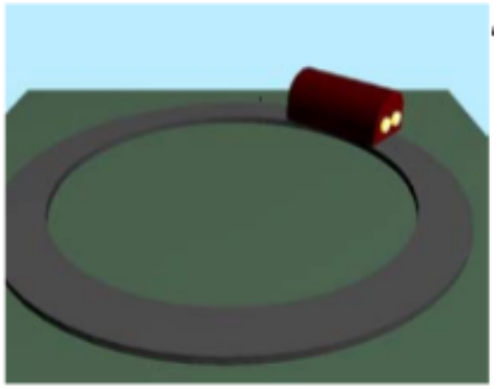

Figure 4: A simple 3D visualization

2.6 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 starts 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 or click on some elements in the 3D visualization.

2.7 General product overview and use

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 is in charge of modeling 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.
  - A Petri net editor to create a Petri net model.
  - A geometry editor to create and edit the geometry of the Petri net.
  - An appearance editor to set the appearance of the objects.
  - A configuration editor to configure the simulation.

  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.

Moreover, this project also introduces new concepts related to Petri nets, because 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 the 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 but also informations about their looks. For example when the user creates a line in the geometry editor, the line will have the same position in the 3D simulation.

- **Appearance Label**: An appearance label gives the Geometry object a label that provides information about the appearance of the object in the 3D simulation. The label is used to specify which 3D object should represent the Geometry object. One can find informations about appearances by reading further in this section.

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.

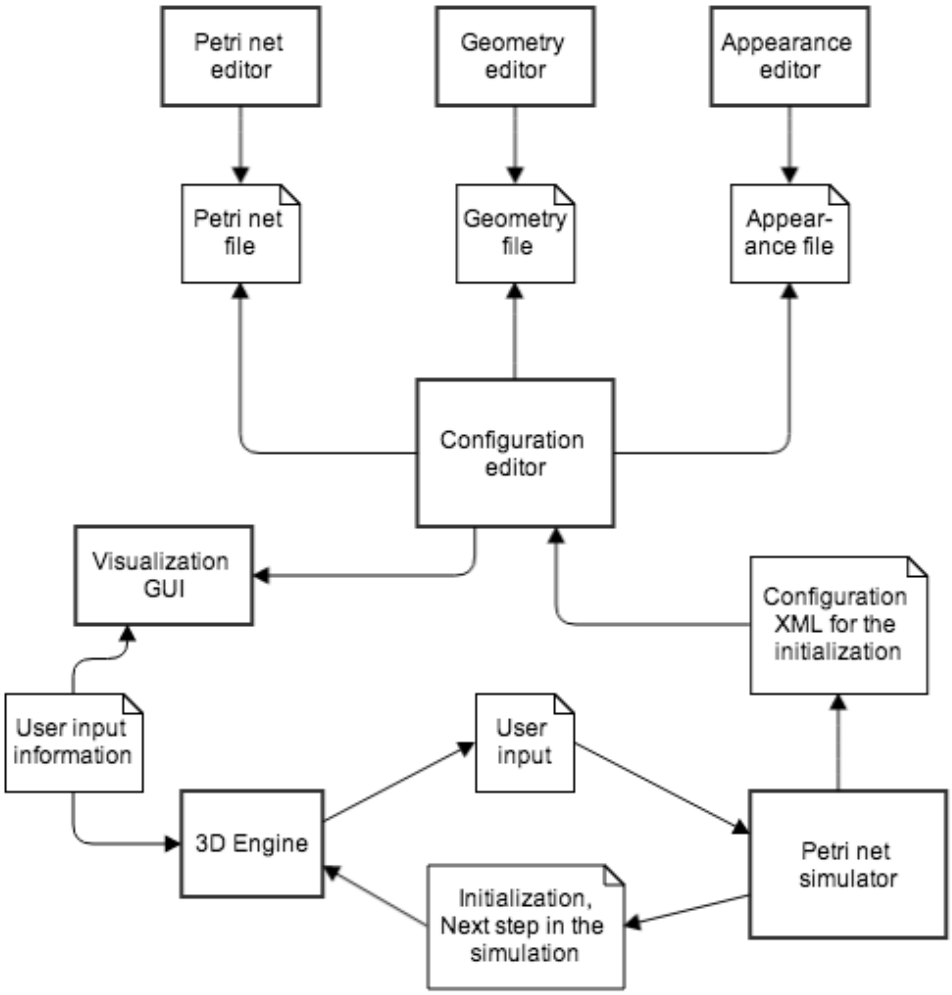

Figure 5: System design

2.7.1 Motivation for Input Places

In the previous section it has been introduced the concept of Input place, an extension to the Petri Net for enabling the user to interact with the 3D simulation. It has been conceived as an easy-to-use concept for modelling elements such as traffic lights, switches or creation of new elements (trains, cars, ...) on run time. These elements correspond to real world inputs which cannot be modelled using a regular Petri Net, and thus this extension can help the end-user to simulate them.

3 System Features

Author: ***Thibaud***

In this section, the main features of the components will be discussed, as well as possible features that could be implemented in future releases of the product. The main functionality needed by each component in order for the project to work will be explained into more detail.
The main use cases of the system components are also provided in each subsection in order to describe how the system will be used by the final user.

3.1 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.

3.2 Petri net editor

Author: ***Albert***

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

3.2.1 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 Input Place label to a Place.
7. The Petri net editor **shall** allow the user to define an Animation label to a Place.

8. The Petri net editor **shall** allow the user to save and load a Petri net model.
9. It **would be nice** that the Petri net editor allowed the user to undo and redo actions.
10. It **would be nice** that the Petri net editor allowed the user to copy and paste.

3.2.2 Use cases

The features described above are shown in the following use case diagram Figure 6.

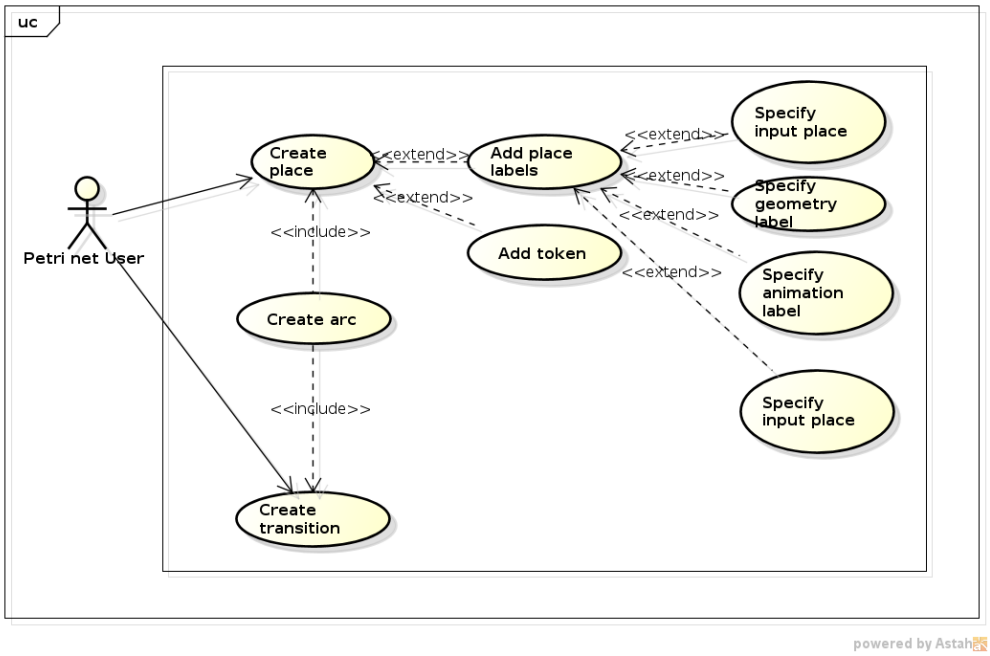

Figure 6: Use cases for the Petri net Editor

3.3 Geometry editor

Author: ***Mikko***

The Geometry Editor is a component that will allow the user to create and edit geometry models. Such models consist of geometry objects, namely *lines* and *points* which are defined as two dimensional representations of the Petri net objects to be simulated. The connection between a geometry and Petri net objects will be done via*geometry labels*.

3.3.1 Functional Requirements

1. The Geometry Editor **shall** allow the user to create, edit and delete points.

2. The Geometry Editor **shall** allow the user to define points as Input Point, Connector or Bend Point.
3. The Geometry Editor **shall** allow the user to create, edit and delete lines.
4. The Geometry Editor **shall** allow the user to load and save geometry files.
5. The Geometry Editor **shall** allow the user to define labels for geometry objects.
6. The Geometry Editor **shall** allow the user to define appearance labels for Lines and Input Points.
7. The Geometry Editor **should** allow the user to create curved lines.
8. The Geometry Editor **should** allow the user to create, edit and delete bend points for the lines.
9. The Geometry Editor **should** allow the user to use undo and redo.
10. The Geometry Editor **should** allow user to use copy, paste and cut geometry objects.
11. The Geometry Editor **should** have drag and drop interface.
12. It **would be nice** if the Geometry Editor allowed the user to create different kinds of parametric curved lines, such as Catmull-rom splines and Bézier-curves.
13. It **would be nice** if the Geometry Editor allowed the user to zoom and pan the geometry canvas.
14. It **would be nice** if the Geometry Editor allowed the user select multiple geometry objects simultaneously.
15. It **would be nice** if the Geometry Editor allowed the user to load multiple geometries on same canvas.
16. It **would be nice** if the Geometry Editor allowed the user to rotate and scale geometry objects.

3.3.2 Use cases

The features described above are shown in the following use case diagram Figure 7.

3.4 Appearance Editor

Author: ***Monica***

The Appearance Editor is a component that will allow the user to add visual information such as 3D objects and textures to the Petri net objects to be simulated. The connection between the appearance files and the Petri net objects will be done via the *appearance labels* defined by the user in the Geometry editor (See section 3.3).

Before using the Appearance Editor, the user has to *create* a new or *load* an existing appearance file he/she wants to edit.

3.4.1 Functional Requirements

1. The Appearance Editor **shall** allow the user to create new appearance objects, *AObject*.
2. The Appearance Editor **shall** allow the user to input the name of an appearance label for an AObject.
3. It **would be nice** to allow the user to load a Geometry file in order to retrieve all the appearance labels.
4. The Appearance Editor **shall** allow the user to load the path of a predefined 3D object file for an AObject.
5. The Appearance Editor **shall** allow the user to load the path of a predefined texture file for an AObject.
6. The Appearance Editor **shall** allow the user to save an Appearance file.

3.4.2 Use cases

The features described above are shown in the following use case diagram Figure 8.

3.5 Configuration Editor

Author: ***Albert, Monica***

The Configuration Editor is a component that will work as a connector between the Petri net editor (Section 3.2), the Geometry editor (Section 3.3) and the Appearance editor (Section 3.4).

The editor will allow the user to load the corresponding files, created in the previous editors, that will make it possible for the simulation to be started. In order to use the Configuration Editor, the user has to *create* a new or *load* an existing appearance file he/she wants to edit.

3.5.1 Functional Requirements

1. The Configuration Editor **shall** allow the user to load a Petri net doc file as a resource.
2. The Configuration Editor **shall** allow the user to assign the loaded Petri net doc file as a value to the Configuration's Petrinet property.
3. The Configuration Editor **shall** allow the user to load a Geometry file as a resource.
4. The Configuration Editor **shall** allow the user to assign the loaded Geometry file as a value to the Configuration's Geometry property.

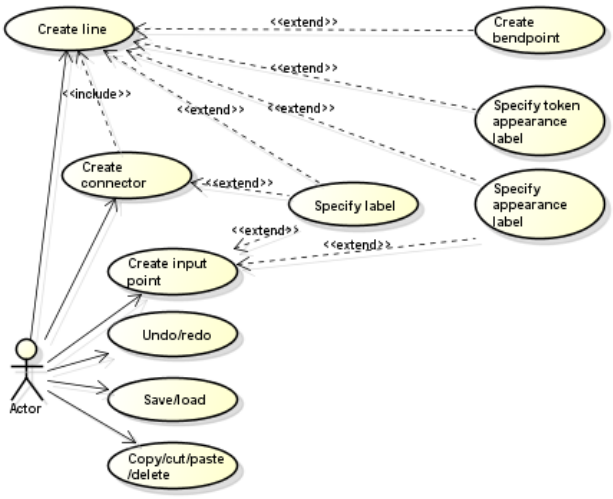

Figure 7: Use cases for the Geometry Editor

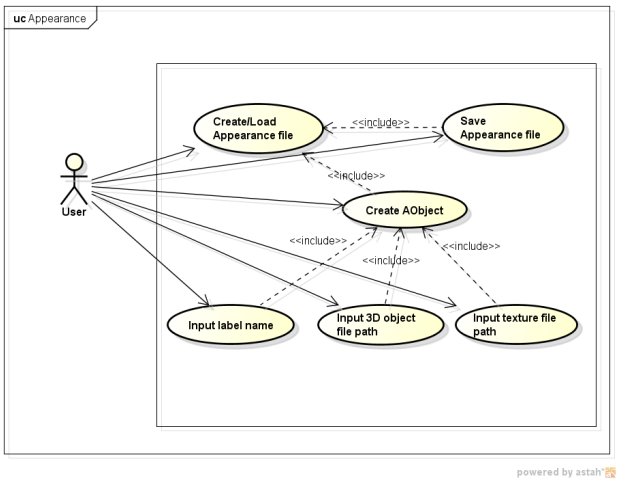

Figure 8: Use cases for the Appearance Editor

5. The Configuration Editor **shall** allow the user to load an Appearance file as a resource.
6. The Configuration Editor **shall** allow the user to assign the loaded Appearance file as a value to the Configuration's Appearance property.
7. The Configuration Editor **shall** allow the user to start a simulation with the referenced Petri net, Geometry and Appearance resources.
8. The Configuration Editor **shall** allow the user to validate that all the needed files have been loaded.
9. The Configuration Editor **shall** allow the user to save a Configuration file.

3.5.2 Use cases

The features described above are shown in the following use case diagram Figure 9.

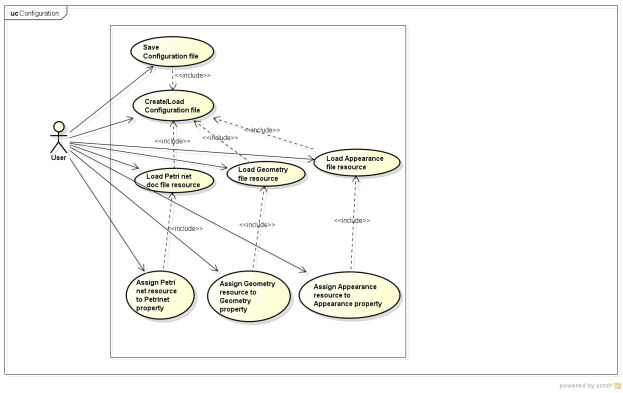

Figure 9: Use cases for the Configuration Editor

3.6 3D Simulator

Author: ***Anders***

The 3D Simulator is a component that will allow the user to see the animated Petri net, while also being able to play/pause the simulation, to change the camera perspective of the simulation and to interact with objects. The functionality of the 3D simulator can be specified using the following statements.

3.6.1 Functional Requirements

1. The Simulator **shall** allow the user to start 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 by interacting with the 3D animation.
5. The Simulator **shall** allow the user to exit the simulation.
6. The Simulator **shall** allow the user to change the perspective of the view.
7. It **would be nice** if the Simulator allowed the user to take screen shots.
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.

3.6.2 Use cases

The features described above are shown in the following use case diagram Figure 10.

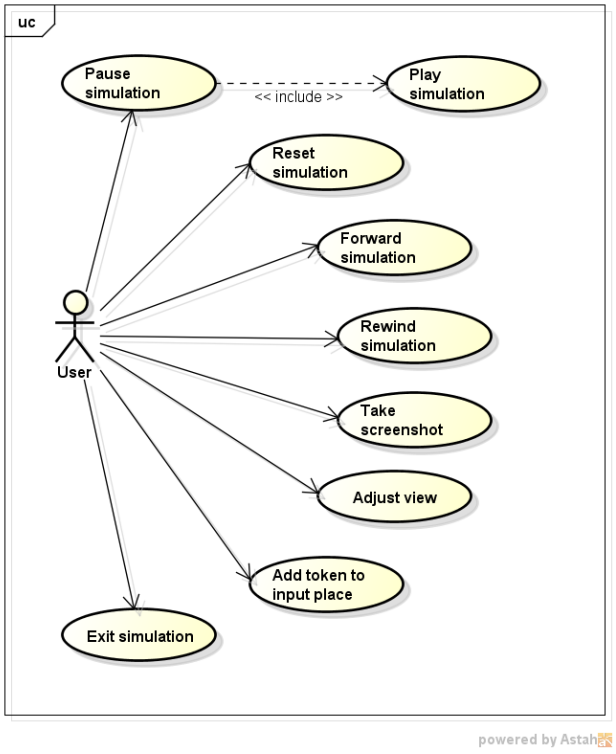

Figure 10: Use cases for the 3D Simulator

4 Non functional requirements

Author: ***Mikko***

In addition to implementing the required functionality of the product, a number of non-functional requirements were also taken into account. This sections describes into more detail the latter.

4.1 Implementation constraints

The software must be implemented as a plug-in to the Eclipse framework and it is based on the 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. ePNK v. 1.0.0
5. JMonkey v. 3.0

4.2 Documentation

Development must be documented and the documents were delivered according to the following deadlines:

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

4.3 Quality Assurance

The procedures that ensure the quality of this software product are described in the following section.

1. Quality of documentation
   (a) Each part of the document is realized by multiple persons in order to decrease the amount of typical errors.

(b) Each part of the documentation is reviewed and feedback is given by each member of the team. Depending on the feedback, appropriate measures are taken and the documentation is edited.

(c) Internal deadlines for documentation are set such that there is enough time allocated for internal reviews before the real deadline.

2. Quality of implementation

(a) Each implementation is reviewed by at least two persons in order to decrease the amount of typical errors.

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

(c) A test strategy shall be developed to test the implementation. It will consists of component and functional testing. Component testing will be performed by doing unit tests on manually created methods. Functional testing will be done by performing common user cases and comparing the result to the excepted results.

4.4 Performance requirements

The software is expected to fulfill the following performance requirements:

1. Satisfactory response times for commands in the geometry and Petri net editors are immediate.
2. Satisfactory response time for commands in the simulator should be immediate for small models. No upper boundary for the size of the model is made thus the response time for larger models will gradually increase with the size of the model.
3. Generally, the amount of user requests as scale of time is small. When the relative amount of requests is large they are simple, such as writing in editors, and when the relative amount of requests is small they are more advanced, such as play command for 3D simulator.
4. The number of users that can perform actions using the software at a time is one. Only one configuration can be run at a time due to the fact that JMonkey does not support multiple simultaneous calls.

4.5 Usability requirements

The software is expected to fulfill the following requirements in terms of usability:

1. The editors and the simulator will have familiar interfaces such that learning to use them is as efficient as possible.
2. The less the user needs to know about software and still be able to use it effectively the less technical oriented the user needs to be. Thus increasing the availability to the software.
3. The software is divided into multiple parts in order to enable users with different levels of expertise to contribute in working with the software.

4. User input should tend towards point-and-click usage style.

5 User Interface

Author: ***Morten***

The following section will describe how the Graphical User Interfaces (GUIs) of the Petri net editor, Geometry editor, Appearance editor, Configuration editor and the Simulation window should look like. How to execute different tasks in the editors, such as creating and editing Geometries and Petri nets, will be described in the handbook.

The different editors will strive towards being intuitive for the targeted user, e.g. the Geometry editor should be easy to use for the non-technical user, while the Petri net editor should be intuitive for the Petri net engineer.

5.1 Technologies

The editors will be implemented as extensions to Eclipse, which allows us to use the different features that come with Eclipse. The functionality listed below is 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 object selected in the editor and 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 features will be further explained in the user handbook.

5.2 Graphical User Interface

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

5.2.1 Introduction to Petri net editor & Geometry editor.

The Petri net and Geometry editors are very much alike. They have the same composition and only the objects that can be drawn on the canvas are different. Both editors allow the use of two kinds of editing views: a *tree editor* and a *diagram editor*. On one hand, the diagram editor consist of a palette, a view with the project files, a view with the properties of the selected object and a canvas. On the other hand, the tree editor has the project explorer, the properties view and a view with the objects tree. Examples of Petri net and Geometry editors can be seen in figure 11 & 12 and figure 13 & 14 respectively.

5.2.2 Petri net editor

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

Canvas (diagram editor)
The canvas is the place where the user can create/draw the Petri net. In the diagram editor the user selects an object from the palette and left-clicks on the canvas to create the object. The user can edit the position of the different objects by dragging them around 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 diagram editor.

Palette
To create and modify the Petri net the user will be able to create objects on the canvas by selecting the desired tool in the palette on the right side. The palette in the Petri net editor has tools to create places, transitions and arcs. Each object will have a dedicated icon in order to increase the usability of the GUI.

Tree editor
In the tree editor the user can create objects by right-clicking and selecting Add object. The tree editor gives the user a quick overview of how the different objects are connected in a hierarchical manner.

Object properties
Each object has different properties that can be defined by the user. The user can modify these properties in the properties view whenever an object is selected. The Petri net objects have the following properties:
Transition

- **Id:** The id of the object used in the simulator.
- **In:** The incoming arcs.
- **Out:** The outgoing arcs.

Place

- **Id:** The id of the object used in the simulator.
- **In:** The incoming arcs.
- **Out:** The outgoing arcs.
- **Tokens:** A list of tokens placed on this place.
- **Input place label:** Determines whether the place is an input place. Can either be true of false.

Arc

- **Id:** The id of the object used in the simulator.
- **Source:** The source object.
- **Target:** The target object.

5.2.3 Geometry editor

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

Canvas (diagram editor)
The canvas is the place where the user can create/draw the geometry. In the diagram editor the user selects an object from the palette and left-clicks on the canvas to create the object. The user can edit the position of the different objects by dragging them around the canvas. Furthermore, the connections between the objects can be edited by dragging lines between two objects. The user can create bend points on these lines to modify the slopes of the lines. The user will be able to see the graphical representation of the Petri net in the Petri net diagram editor.

Palette
To create and modify the geometry, the user will be able to create objects on the canvas by selecting the desired tool in the palette on the right side. The palette in the geometry editor has tools to create input places, connections and lines. Each object will have a dedicated icon in order to increase the usability of the GUI.

Object properties
Each object has different properties that can be defined by the user. The user can modify these properties in the properties view whenever an object is selected. The Geometry objects have the following properties:
Input point

- **Appearance label:** The appearance label is used in the appearance editor where the user specify the appearance in the 3D simulation of the specific label.
- **Label:** Used in the 3D engine for recognition.
- **X and Y location:** The position in the simulation.

Connector

- **Label:** Used in the 3D engine for recognition.
- **In:** The incoming lines.
- **Out:** The outgoing lines.
- **X and Y location:** The position in the simulation.

Line

- **Appearance label:** The appearance label is used in the appearance editor where the user specify the appearance in the 3D simulation of the specific label.
- **Begin:** The source connector.
- **End:** The target connector.
- **Label:** Used in the 3D engine for recognition.
- **Token appearance label:** The appearance label of the tokens on this line. The user specifies the appearance in the appearance editor.

5.2.4 Appearance editor

Figure 15 is an example of the appearance editor. The user has previously specified, in the Geometry editor, the appearance labels for each object. The user creates a new Appearance object (AObject) in the editor and gives it the same label as given in the Geometry editor. The user can specify the texture and model files of the appearance in the texture and Object3D properties.

5.2.5 Configuration editor

The configuration editor (figure 16) helps the user to set up the 3D simulation. In the editor the user has to specify the Petri net, the geometry and the appearance files that he/she wants to use for the 3D simulation. In this way the user can, for example, have different Petri nets but still use the same visual representation for tokens and places in the simulator.

5.2.6 Simulation tool

When the user has successfully set up the Configuration, 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 state of the system; if the simulation state is play the button will be a pause symbol and the other way around. The window also has a **reset** button that allows the user to reset the simulation. Moreover, the user will be able to interact with the simulation, therefore the user will be able to use the mouse to click on different objects. Last but not least, the user will also be able to use the mouse and keyboard to change the camera perspective. An example of the simulation tool can be seen in Figure 17.

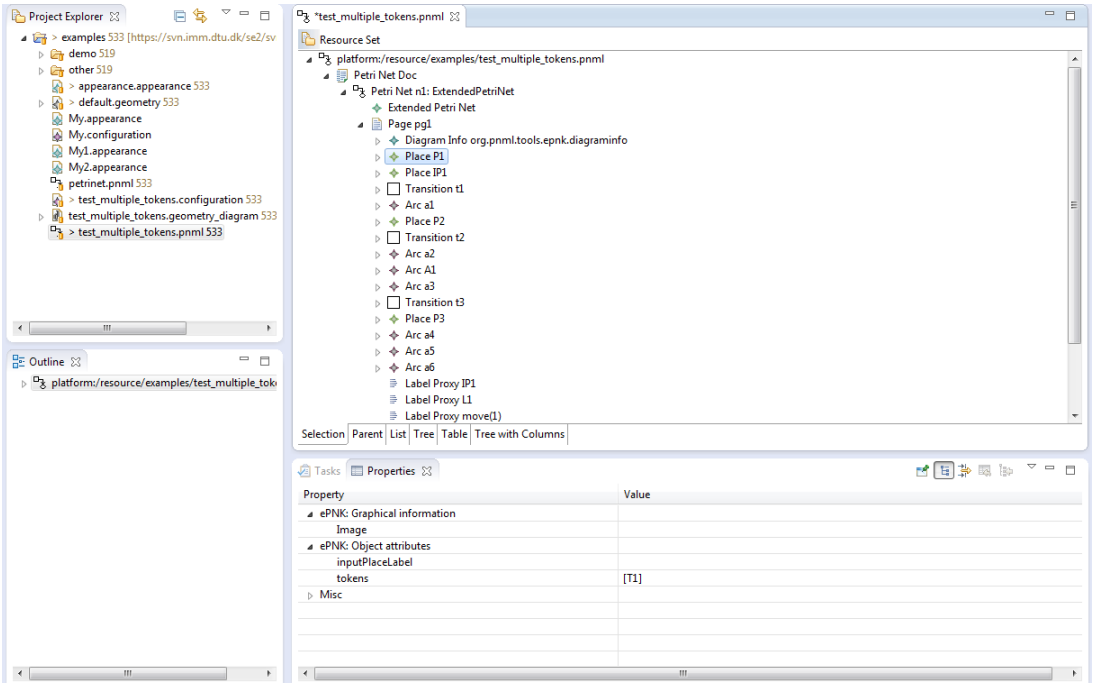

Figure 11: The Petri net tree editor.

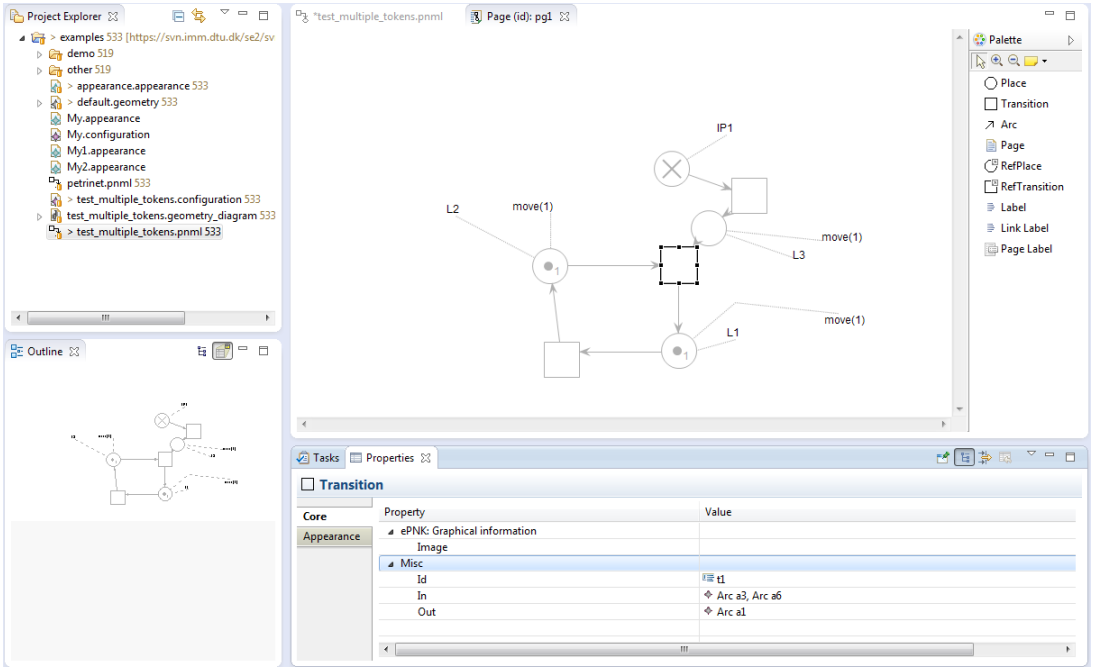

Figure 12: The Petri net diagram editor.

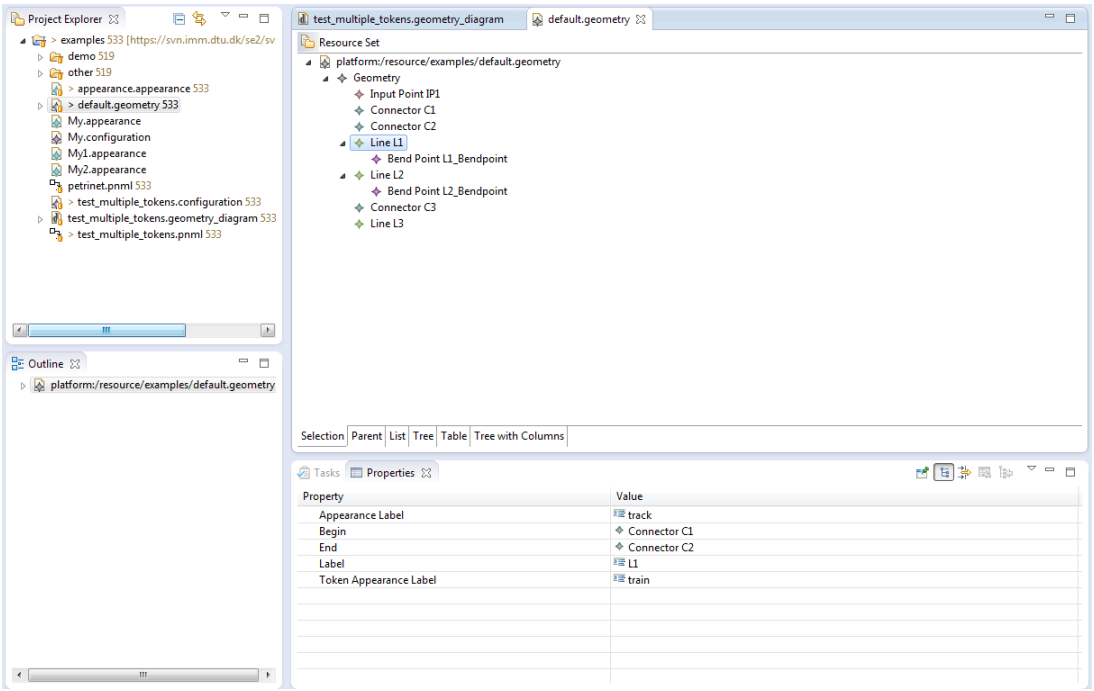

Figure 13: The geometry tree editor.

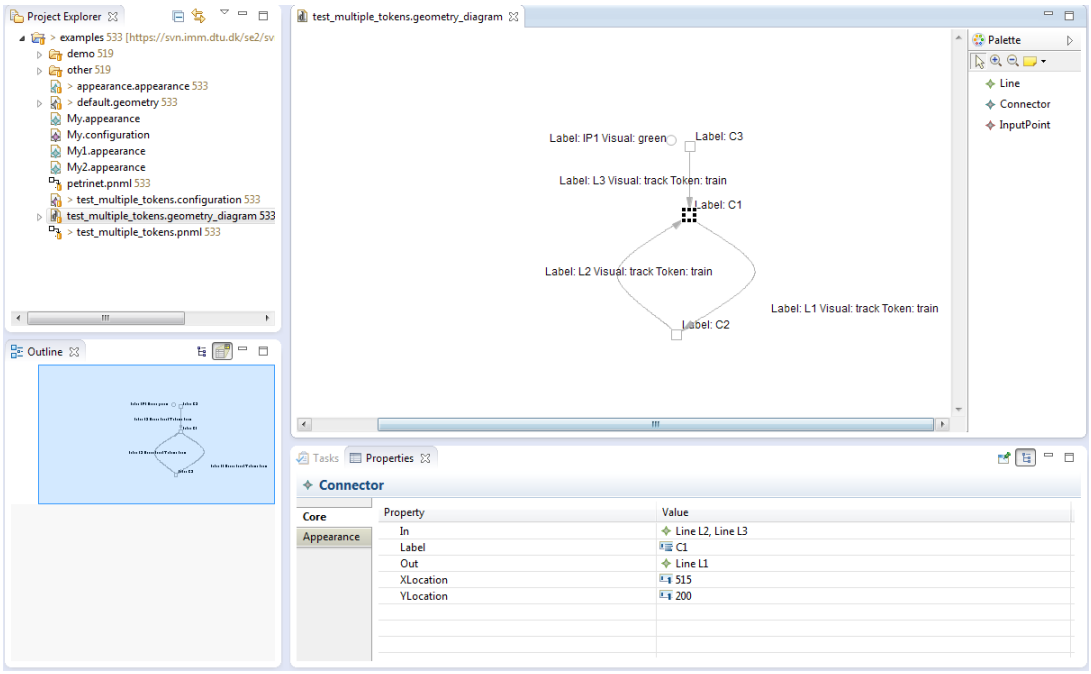

Figure 14: The geometry diagram editor.

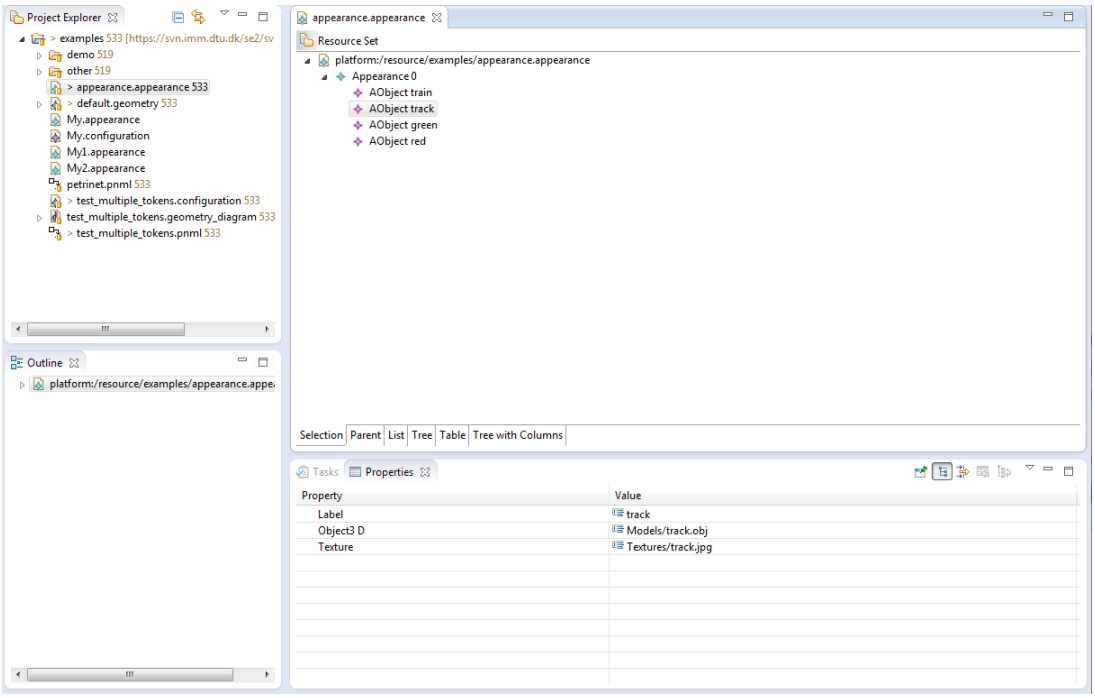

Figure 15: The Appearance editor GUI.

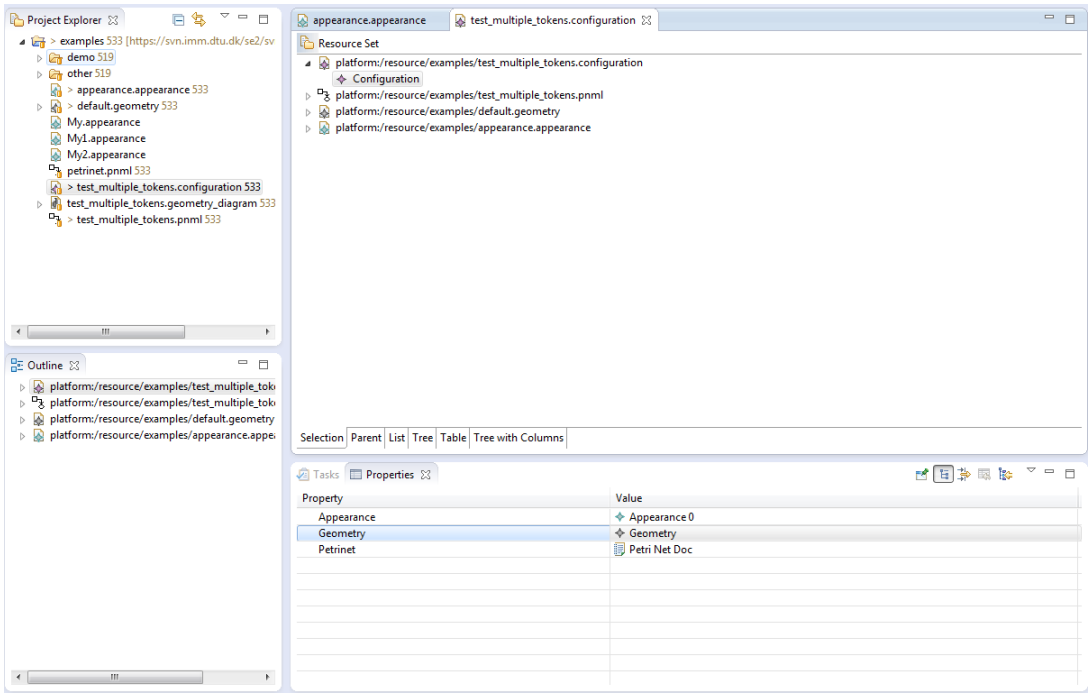

Figure 16: The configuration editor.

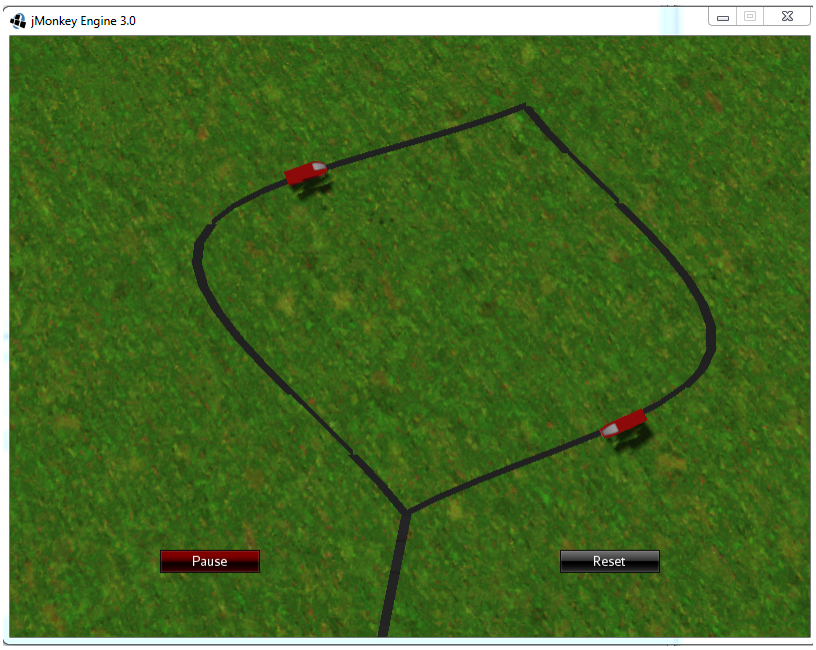

Figure 17: An example of the Simulation tool.

6 Architecture

Author: ***Albert and Monica***

The architecture of this **Extended Petri net 3D 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.

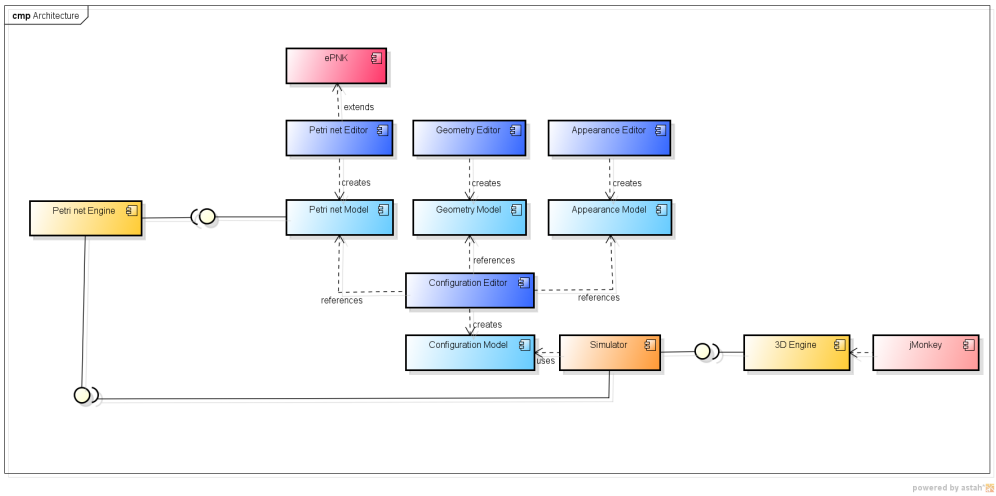

Figure 18: Architecture of the System

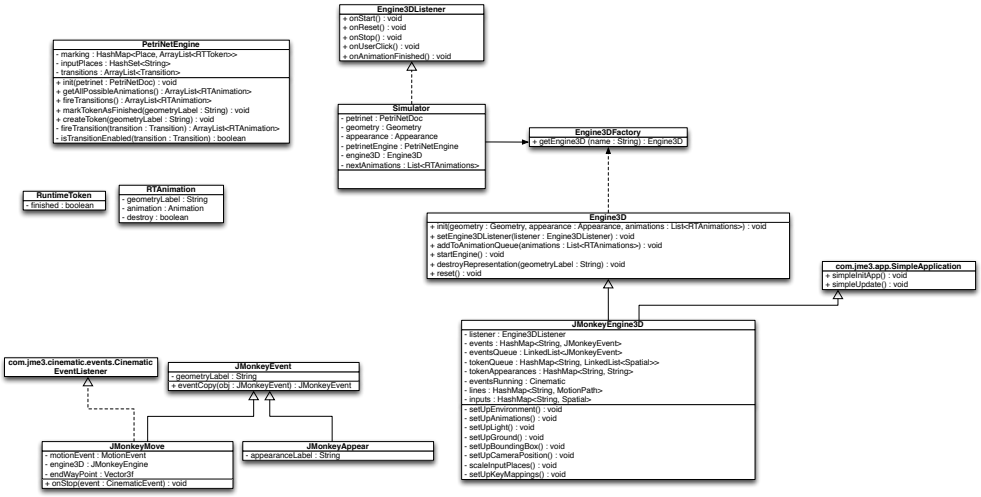

Figure 19: Class diagram of the system

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. The actual implementation of the Petri net editor is shown in Figure 20.

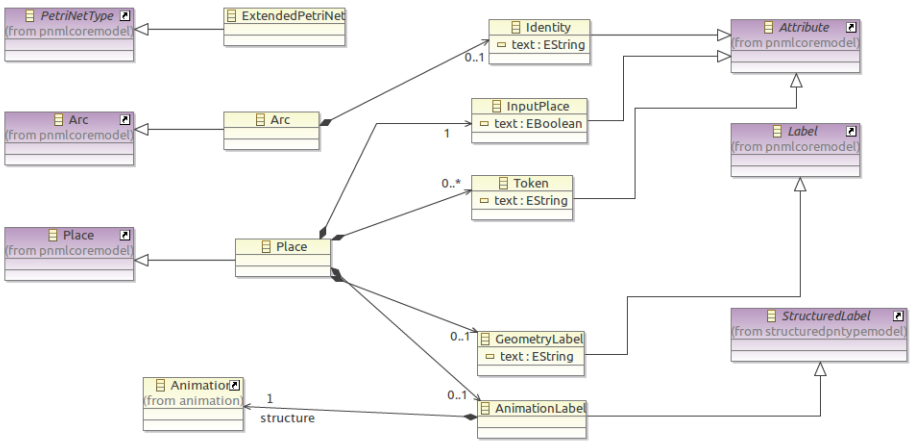

Figure 20: Petri net editor domain model

6.1.1 Petri net editor classes

A description of the classes shown in Figure 20 is provided.

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

- **Token**: an ePNK label to indicate the presence of one or more tokens in this Place.
- **InputPlaceLabel**: an ePNK boolean attribute to indicate if the place is interactive for the user or not (if tokens can be added by the user during runtime).
- **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.

6.1.2 Animation

As shown in Figure 20, Places have a defined animation model, which is the one shown in Figure 21.

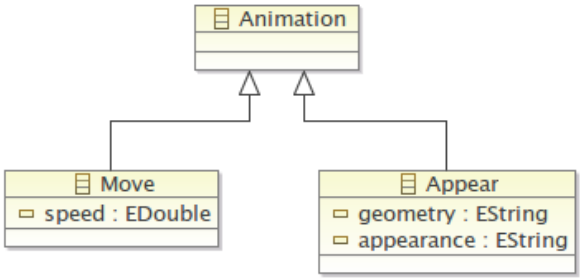

Figure 21: Animation model

There are two possible animations: **move** or **appear**. The first animation type, move, has speed as a parameter, whilst the second type, appear, includes the geometry label of the place whose shape we want to change, and the name of the shape we want that place to have. Using only these two animations it is possible to visualize all the necessary Petri net behavior.

6.1.3 PAL: Petri Net Animation Language

In order to be able to read the animations defined by an animation label in the Petri Net Editor, a language has been implemented using Xtext. The defined syntax is the following:

```
move(2.0)
```

```
appear(L1, red)
```

6.1.4 Graphical Extensions

When extending from ePNK it was also necessary to add two graphical extensions in order to provide more usability to the end-user. An example of such an extension is showing differently in the Petri net diagram which places have tokens and which are input places. Therefore, we implemented the following method in an inherited class of PlaceFigure, which changes the shape of a Place at run time depending on whether the Place has tokens or has been marked as an input place.

```
protected void fillShape(Graphics graphics) {
            super.fillShape(graphics);
            dk.dtu.se2.petrinet.Place p = (dk.dtu.se2.petrinet.Place) this.place;
```

```
            if (!p.getTokens().isEmpty()) {
                    Rectangle rectangle = this.getClientArea();
                    graphics.setBackgroundColor(getForegroundColor());
                    graphics.fillOval((int) (rectangle.x + 0.325 * rectangle.width),
                                    (int) (rectangle.y + 0.375 * rectangle.height),
                                    (int) (0.25 * rectangle.width), (int) (0.25 *
                                        rectangle.height));
                    graphics.drawString(String.valueOf(p.getTokens().size()),
                                    (int) (rectangle.x + 0.6 * rectangle.width),
                                    (int) (rectangle.y + 0.45 * rectangle.height));
            }

            InputPlace inputPlace = p.getInputPlaceLabel();
            if(inputPlace != null && inputPlace.isText()){
                    Rectangle rectangle = this.getClientArea();
                    graphics.setBackgroundColor(getForegroundColor());
                    graphics.setLineWidth(2);
                    graphics.drawLine(rectangle.x + 10, rectangle.y + 10,
                                    rectangle.x + rectangle.width - 10, rectangle.y +
                                        rectangle.height - 10);
                    graphics.drawLine(rectangle.x + 10, rectangle.y + rectangle.height
                        - 10,
                                    rectangle.x + rectangle.width - 10, rectangle.y +
                                        10);
            }
    }
```

6.2 Geometry editor

Author: ***Mikko***

The Geometry editor is the component used to define lines and points in a two dimensional plane which will then define the location and overall shape of the visualized simulation. Figure 22 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 attribute **name**. It is used to name individual geometry objects, so that user can easily tell them apart. Furthermore, it is used as an identifier for all geometry objects.

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

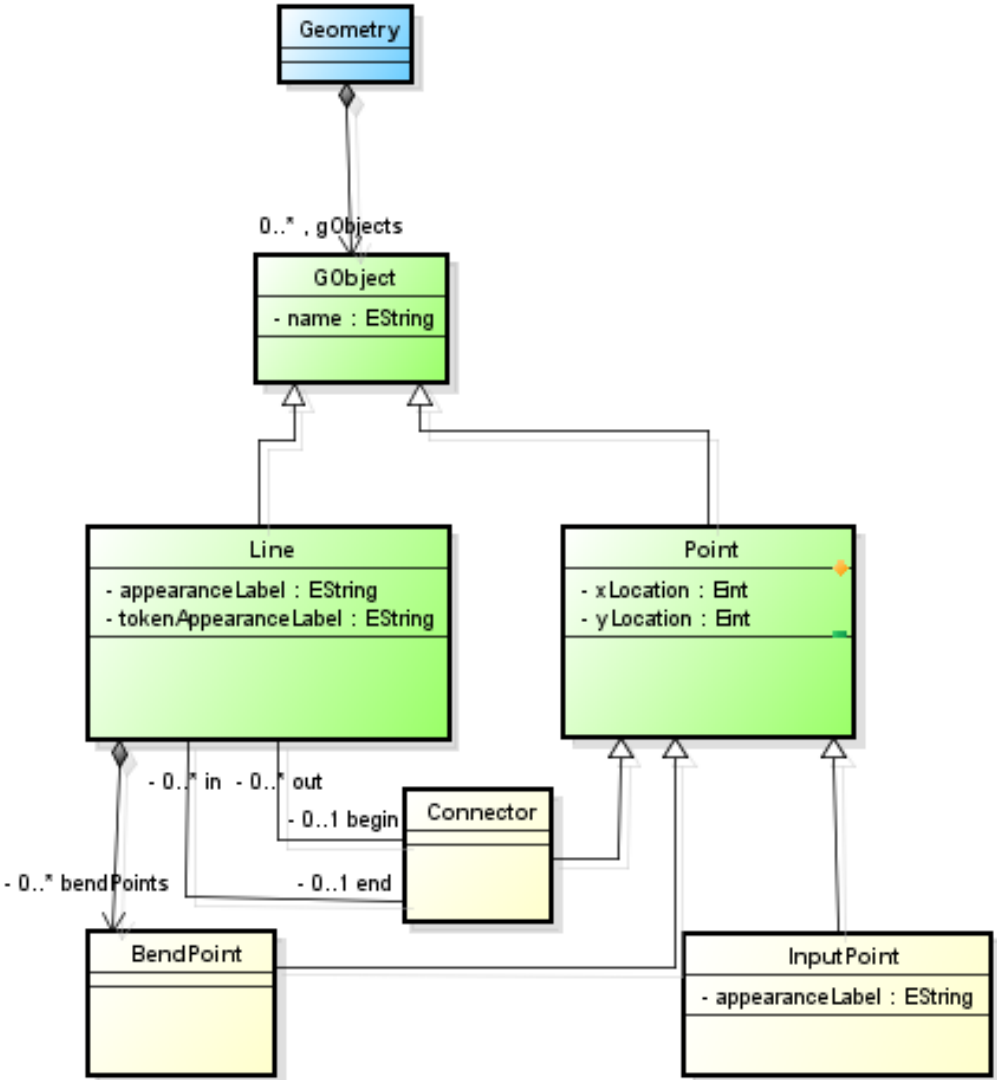

Figure 22: Geometry domain model

InputPoint Class InputPoint refers to objects that will be clickable for the user in the 3D simulation. The**AppearanceLabel** attribute is used to identify the 3D appearance of an InputPoint, such as traffic light, tree, planet etc, within the simulation.

Connector Class Connector is either the beginning or the ending parametric point of a line. Therefore, lines that end at a Connector are **in** and lines that start from a Connector are **out**.

BendPoint Class BendPoint is the type of parametric point that defines the curvature of the line.

Line Class Line is a parametric curve in the two dimensional plane. The line begins at a point location **begin** and it ends at a point location **end**. A **BendPoint** is a point location which defines in a parametric way the curvature of the line, unless it doesn't have one, thus the line would be straight. The**AppearanceLabel** attribute is used to define the 3D shape of the line within the simulation. The appearance will be extrapolated along the length of the line, thus the appearance could for instance be track, road, empty space, etc. On the other hand, the **tokenAppearanceLabel** attribute defines 3D models that are animated along the line. They could be for instance train, bike, space ship, etc.

The geometry editor is initially implemented with Graphical Modeling Framework (GMF). It provides the factory pattern for construction of objects, command pattern for the editing functionality and interfaces for function calls. The implementation is then extended with listeners. The listener notifies the geometry of the changes done to the location of a graphical object in the graphical editor. This information is used to define coordinate locations of geometry objects. Furthermore, the visualization of lines, or as they are called in GMF- connections, in the graphical editor is changed to that of catmull-rom curves.

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 rendered 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 Geometry with predefined 3D and texture files.

Figure 23 shows the model used by the Appearance Editor component.

6.3.1 Appearance Editor classes

Next, the model will be described into more detail.

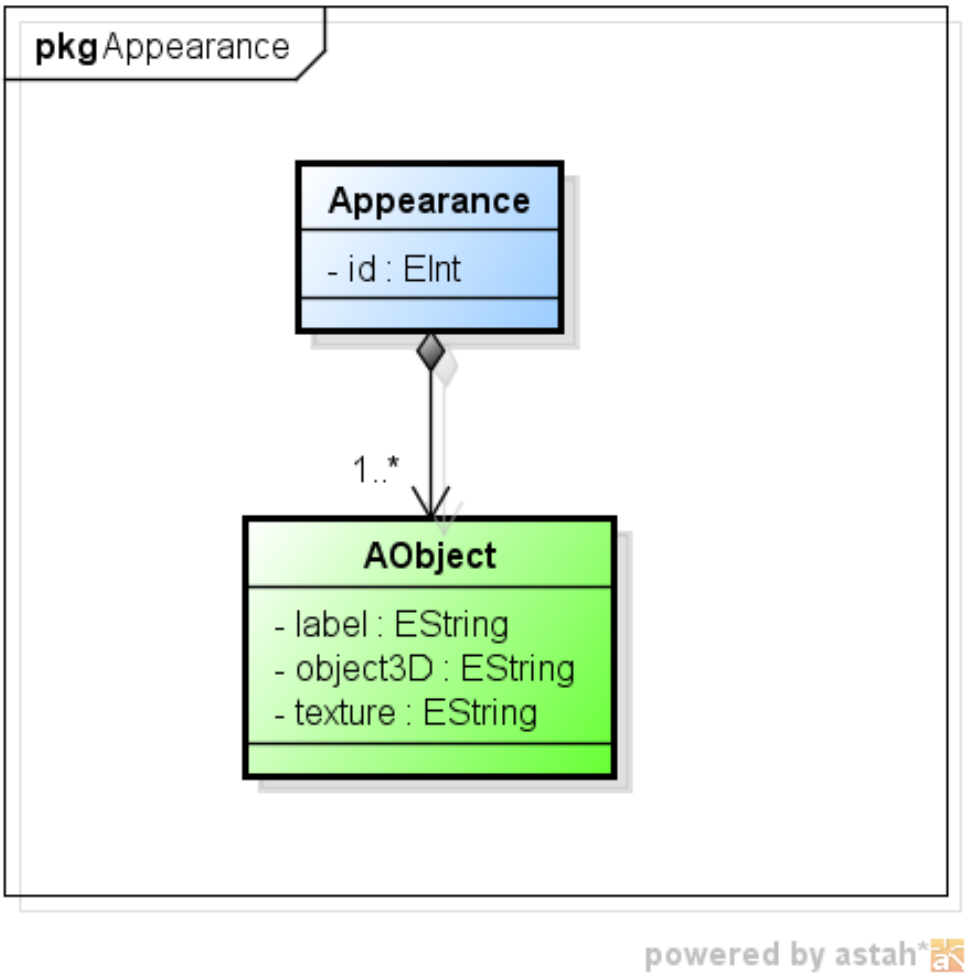

Figure 23: Appearance Domain Model

Appearance The Appearance class is simply an abstract definition of an Appearance object. Its only attribute is the **id**, of type EInt, representing an unique identifier of an Appearance object. Each Appearance object will also contain one or more AObjects.

AObject The AObject class is the class defining all the necessary information for the 3D visual representation of the Petri net. Its attributes are:

1. **label**, of type EString, refers to the appearance label of a Petri net element found in its geometry (e.g. *"train"*).
2. **object3D**, of type EString, refers to the path of a 3D object file that will be used by the simulator during rendering
3. **texture**, of type EString, referring to the path of a texture file that will be used by the simulator during rendering

6.4 Configuration editor

Author: ***Monica***

The Configuration Editor consists of only one class that links Petri net with its Geometry and Appearance. In order to link them, the Configurator class contains references to all the above mentioned classes.

Figure 24 shows the simple model behind the Configuration Editor and how all the information is gathered in order to run the simulation.

After setting up all the references of the above mentioned files, the configuration will combine all of the and start the *Simulator*. This can be done by pressing "Start simulation" in the pop-up menu when right-clicking the configuration object.

6.5 Petri net engine

Author: ***Thibaud, Albert***

Figure 25 shows the model behind the Petri net Engine.

The **Petri net engine** is responsible for handling the behavior of a Petri net at Runtime.

The engine is initialized by the Simulator when a Simulation is launched, using a Petri net model passed as an argument in the Petri net Engine's constructor.

A Petri net engine possesses a **Marking** which is modeled by a HashMap mapping Places to the Tokens present in those places. There is also a specific Token class introduced which is the **RuntimeToken**. This class was introduced to know whether a Token's animation was finished.

The simulator can then interact with the engine by calling one these three methods to simulate the Petri net:

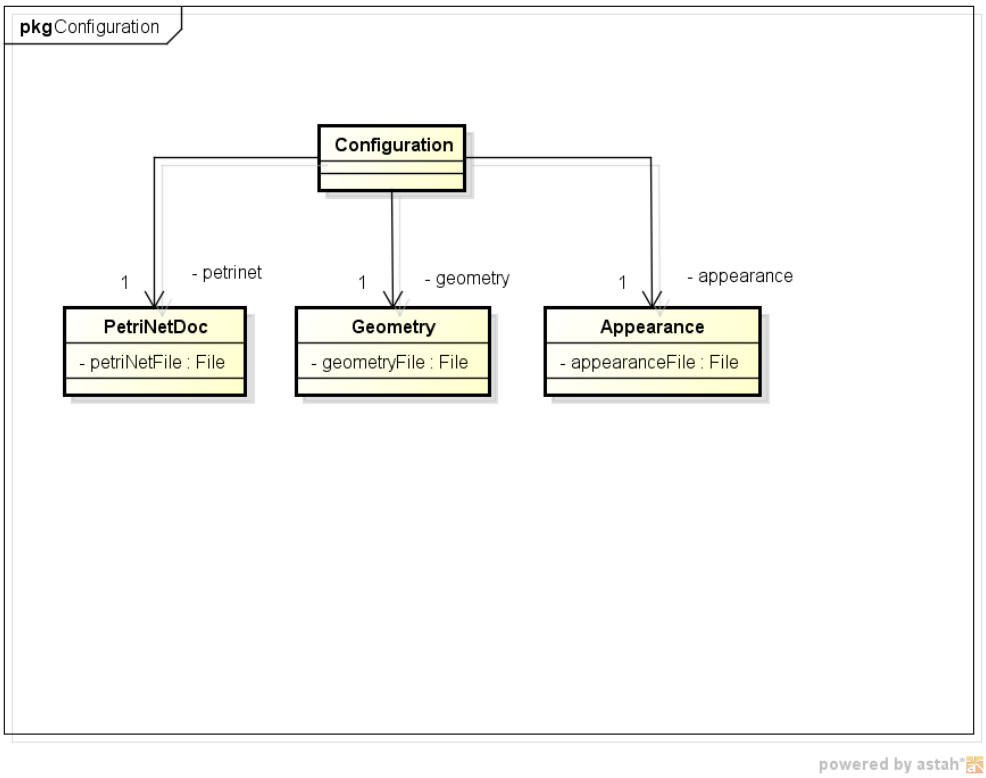

Figure 24: UML for the Configuration Editor

- **getAllPossibleAnimations**: This method is called by the simulator during initialization as the Engine 3D needs the list of animations that should be run at any time.
- **fireTransitions**: This method is called by the simulator whenever an animation is finished. The Petri net engine fires all the transition currently possible and returns a list of **Animation** objects
- **markTokenAsFinished**: This method is called once an animation corresponding to a Token is finished. It is therefore marked as finished so that the method **fireTransitions** can know which transitions are possible to execute right away.
- **createToken**: This method is called by the simulator after an user clicks on an InputPlace. It creates a Token on the Place clicked by the user.

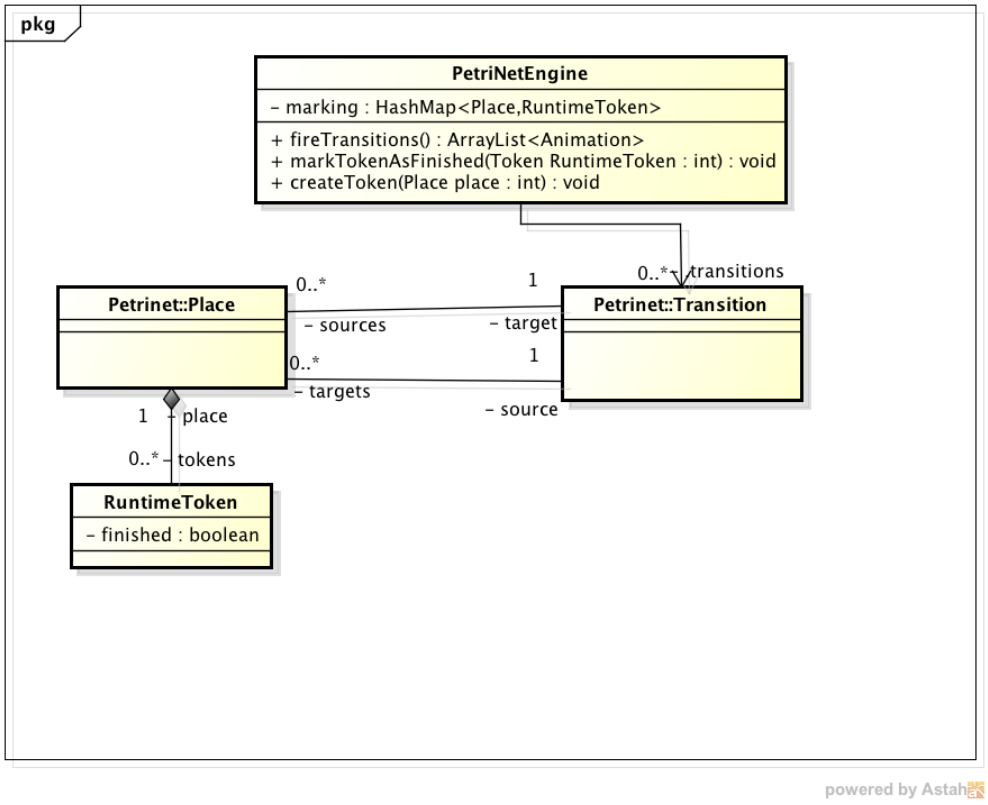

Figure 25: UML for the Petrinet Engine

6.6 Simulator

Author: ***Albert***

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 use the Observer pattern (Figure 26) 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.

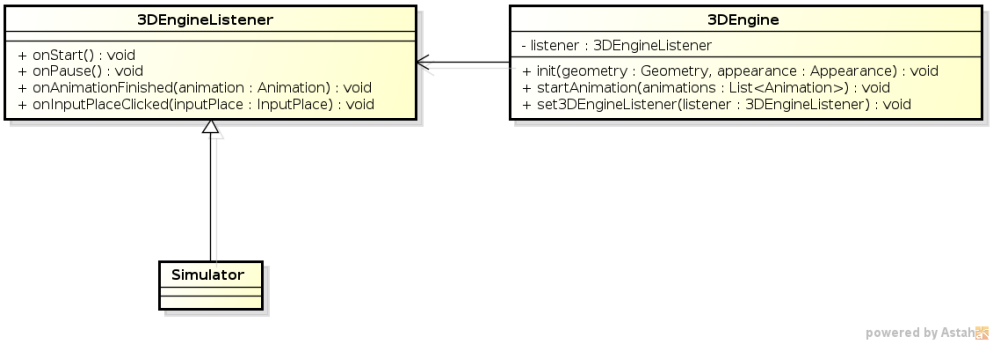

Figure 26: Simulator with the Observer Pattern

Figure 27 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 models.

Then, the 3D engine tells the Simulator that it 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.

6.7 3D Engine

Author: ***Albert, Monica***

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 28 shows the defined classes in the implementation of the Factory pattern.

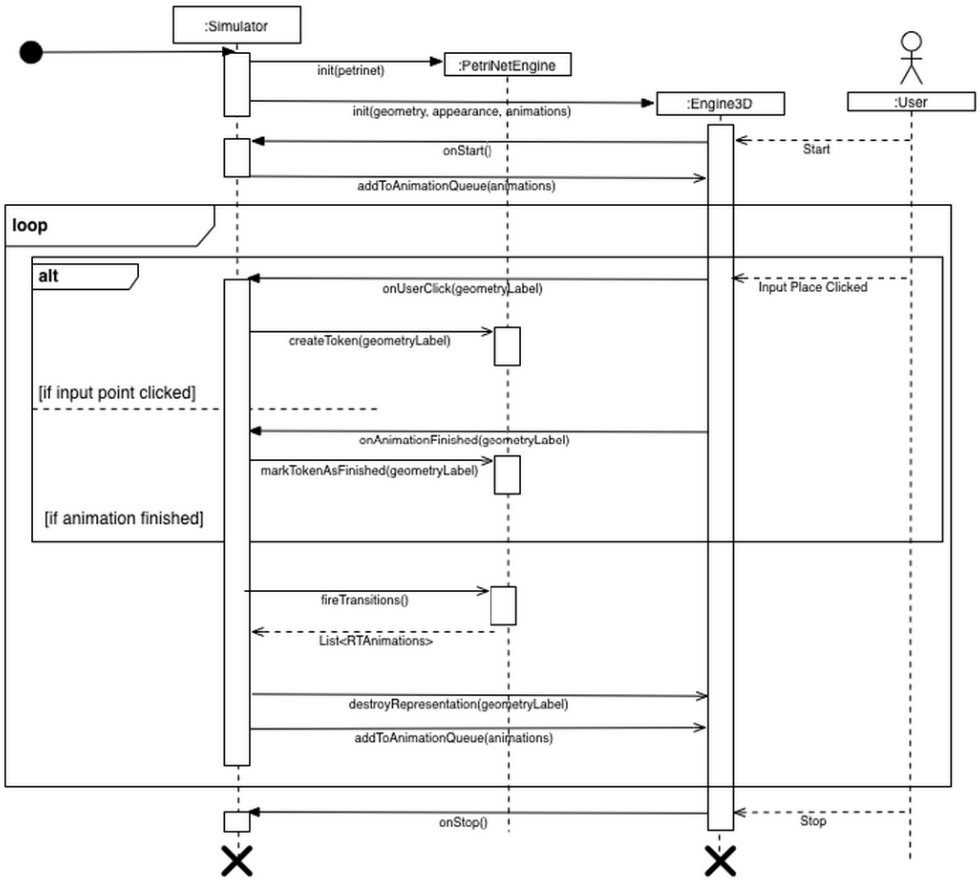

Figure 27: Sequence diagram of the Simulator, Petri net Engine and 3D engine

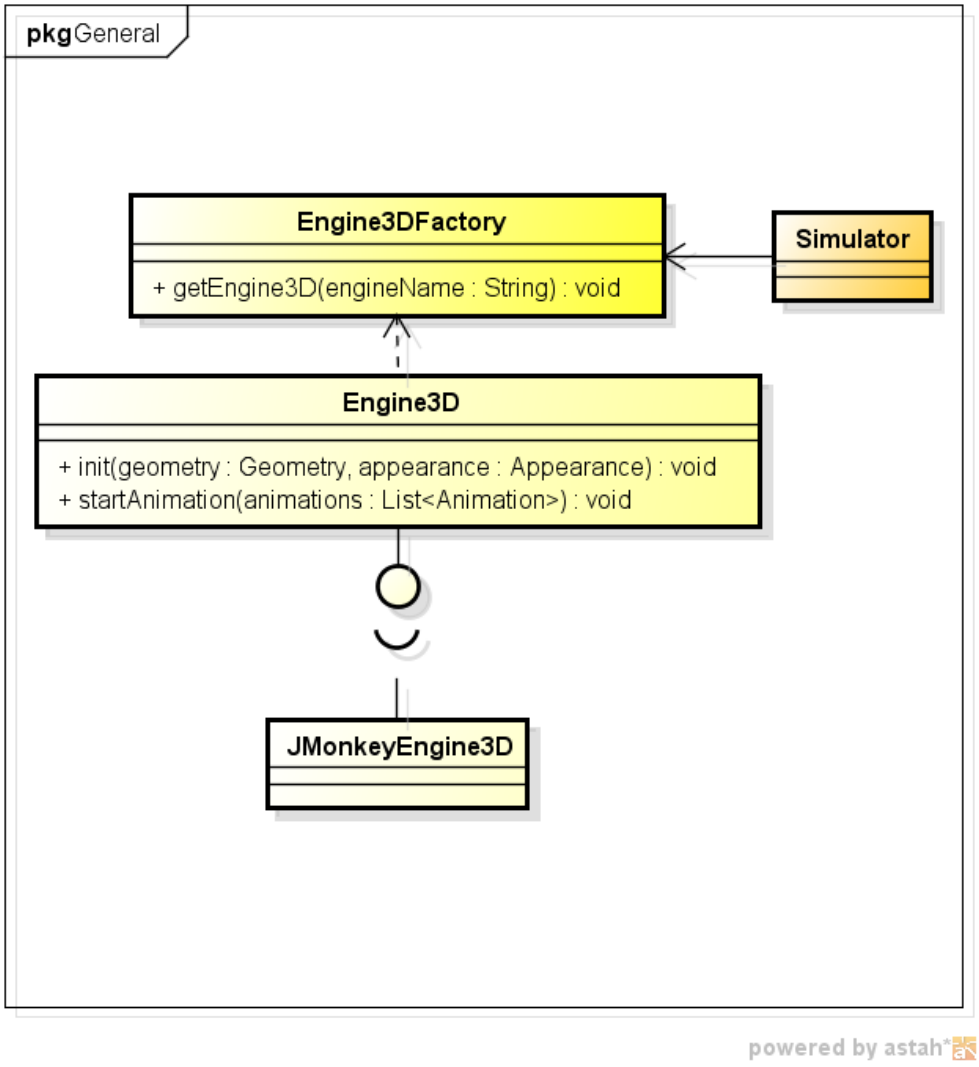

Figure 28: 3D Engine Factory Pattern

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 **JMonkeyEngine**. The following methods will be provided by the Engine3D interface:

- *init(Geometry geometry, Appearance appearance, List¡RTAnimation¿ animations)* - used to initialize the geometry and the appearance models provided by the configuration plus the list of all possible animations;
- *setEngine3DListener listener* - sets a class as the engine's listener
- *addToAnimationQueue(List¡Animation¿ animations* - used by the Simulator to send the list of animations to be performed by the 3D Engine;
- *startEngine()* - used by the Simulator to launch the 3D visualization
- *destroyRepresentation(String id)* - removes a token from the 3D simulation
- *reset()* - used by the Simulator to reset the 3D visualization to the initial configuration *reset()* - restores the simulation to its initial representation

JMonkeyEngine3D As stated before, the JMonkeyEngine3D class will implement all the methods of the Engine3D interface and will perform all the animations it's receiving. This class will be described into more details in the next section.

6.8 The jMonkeyEngine

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

The interface that was described in Section 6.7 will be implemented using the jMonkey library. In the following, the implementation of the engine will be described.

Apart from the methods inherited by implementing the Engine3D interface, jMonkeyEngine also inherits some features from the *SimpleApplication* class, specific to the JMonkey library:

simpleInitApp() - method which sets up general settings of the simulation window such as:

1. the light
2. the environment (rendering of all the 3D objects according to the geometry and appearance files)
3. the bounding box
4. the ground
5. the animations (all possible animations received in the initialization)

6. the position of the camera
7. the key mappings (keys that will trigger a certain behavior when pressed by the user)

simpleUpdate() - called periodically while the simulation is running and verifies the current state of the engine in order to play/pause available animations and check for possible collision.

In order to make the 3D visualization possible, the following JMonkey classes have been used to model the predefined geometries and their corresponding appearances:

- **Curve** - used to draw lines (tracks) in the 3D simulation based on a CatmullRom spline and the control points (connectors and bend points) from the geometry (see Section 6.2.
- **Geometry** - JMonkey object rendered in the simulation and seen by the user
- **Material** - applied to JMonkey geometries for appearance details
- **Texture** - set to the material according to the appearance file
- **MotionPath** - creates a path along a series of control points (the same as for the curve)
- **MotionEvent** - used for *move* animations, requires a MotionPath and a Geometry (such an event will move the given geometry along the given path; e.g. move a train on a track)

All the curves, materials, textures, geometries, paths and all possible event will be prepared in the initialization phase, before the GUI start button is actually pressed, in order to improve the performance of the simulation at run time. This will make the Petri Net visible to the user. It will require stretching and duplication of the shapes and textures. Also, the camera will be positioned and oriented to its default position, and a default background geometry will be created.

All animations are set up in a hash map of ids and MotionEvents and played once the user has pressed the "Start" button. In the beginning, the simulator will provide an initial list of animations to be played. The play and pause actions are dealt with by JMonkey alone but, when an animation has finished, the Simulator is notified and, after receiving the next moves from the Petri net engine, will again provide the list of animations to be run by the 3D engine. This is the main interaction between the simulator and jMonkey.

The buttons in the simulation window are implemented using **Nifty GUI**, a Java library well integrated with JMonkey and used for building interactive graphical user interfaces. Therefore, the buttons screen is overlaid on the JMonkey visualization and the buttons' screen layout is described in an XML file. The buttons actions are caught by the following two methods:

- *onStartButtonPressed()*
- *onResetButtonPressed()*

Moreover, all the other interactions of the user with the simulation window are handled with the implementation of an *ActionListener*.

6.8.1 Collision Detection

Author: *Thibaud*

The collision detection mechanism is straightforward in this simulation. Each time a token is destroyed or created in the Petri net engine, the 3D Engine does the same. Furthermore, each token has a unique ID defined with the help of an atomic counter to prevent concurrent access to the same ID. Since atomic operations take only one instruction of the CPU to execute, we guarantee the thread safety of this property. This ID is then used by the simulator to implement the logic behind the collision detection.

First of all, we defined two hashmaps in our 3D Engine: allRenderedEvents (Key: token ID, Value: JMonkeyMove) and allCollisions (Key: token ID, Value: ArrayList of token IDs which collide with this token). In the allRenderedEvents HashMap, we put each animation running in the simulation.

In the SimpleUpdate loop of the application, we try to collide each spatial with all the other rendered spatials. If they collide, we compare their ids. By design, we know that the token who will need to stop is the one with the biggest ID, since it appeared later on the line. Therefore, we get a list of all tokens who collide with each token, pause all those "collided" tokens, and wait for the collisions to disappear. Once they don't collide anymore, we remove them from the allCollisions HashMap and we resume their animation.

Our 3D Engine doesn't support multi-line collisions yet. If we had a function that could return the progress of the MotionEvent, this would have been really easy (pausing the animations that have the smallest progress when animations collide). However, here, for simplicity, only tokens on the same line will have the opportunity to collide.

7 Glossary

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.

Canvas The graphical view of the editors.

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.

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

Eclipse Eclipse is a integrated development environment, with extensive plug-in system for environment customization.

Editor Is a program used to manipulate information.

ePNK Eclipse Petri Net Kernel.

Factory design pattern Pattern where objects are created, so that the interface that calls the factory will decide which class to create.

Geometry object Geometry object can be either point or line.

GUI Graphical User Interface

Hashmap Hashmap is a data structure where values are mapped to keys.

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

Interface Interface is an abstract type that classes must implement.

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

JMonkey JMonkey is a 3D game engine written in and for java.

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

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 post-conditions for which transitions.

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

Plug-in Plug-in is a software component that adds specific feature to an existing software application.

Simulation Simulation is the imitation of operation of a real world process or system.

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

UML Unified Modeling Language - standardized, general-purpose modeling language in the field of software engineering.

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.

Tool palette A view on the graphical editors that has all the available editing tools for the editors.

Technical/non-technical user User either has or doesn't have prior technical knowledge of the subject.