

System Specification

Extended Petri net Simulator

Group A

October 27, 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	3
1.1	Conventions	3
1.2	Audience	3
2	Overall description	4
2.1	What are Petri nets?	4
2.2	Adding geometry to Petri nets	5
2.3	Adding appearance to Petri net objects	5
2.4	Configuration	5
2.5	Simulating Petri nets	5
2.6	General product description	6
2.7	Basic functionality	8
3	System Features	9
3.1	Petri net editor	9
3.1.1	Functional Requirements	9
3.1.2	Use cases	9
3.2	Geometry editor	9
3.2.1	Functional Requirements	10
3.2.2	Use cases	11
3.3	Appearance Editor	11
3.3.1	Functional Requirements	11
3.3.2	Use cases	13
3.4	Configuration Editor	13
3.4.1	Functional Requirements	13
3.4.2	Use cases	14
3.5	3D Simulator	14

3.5.1	Functional Requirements	14
3.5.2	Use cases	15
4	Non functional requirements	15
4.1	Implementation constraints	15
4.2	Documentation	15
4.3	Quality Assurance	17
5	User Interface	17
5.1	Technology	17
5.2	GUI parts	17
5.3	Handbook	17
6	Architecture	17
6.1	Petri net editor	17
6.1.1	Petri net editor classes	18
6.2	Geometry editor	20
6.3	Appearance editor	20
6.4	Configuration editor	20
6.5	Petri net engine	20
6.6	Simulator	20
6.7	3D Engine	20
6.8	jMonkey	20
7	Glossary	20

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.

¹Petri 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.

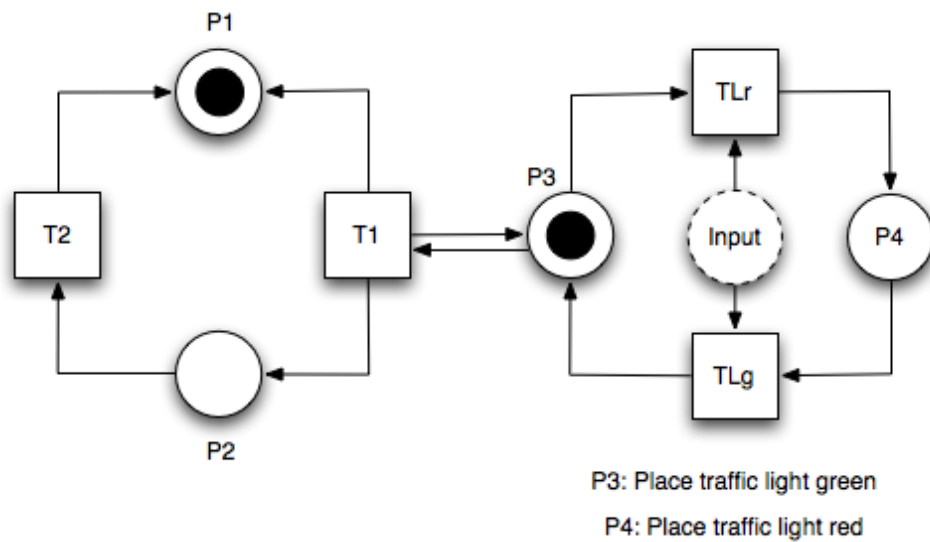


Figure 1: An example of a basic Petri net

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.

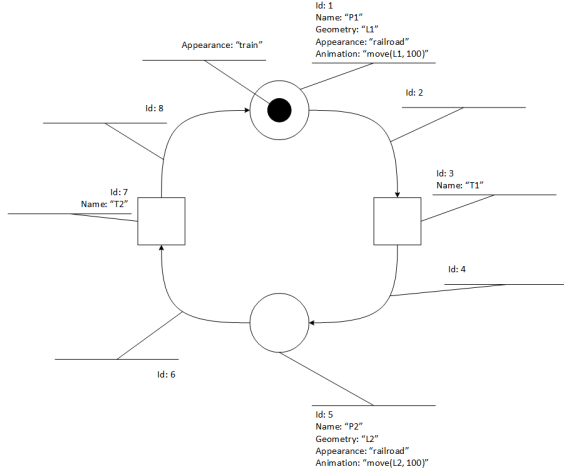


Figure 2: A simple extended Petri net

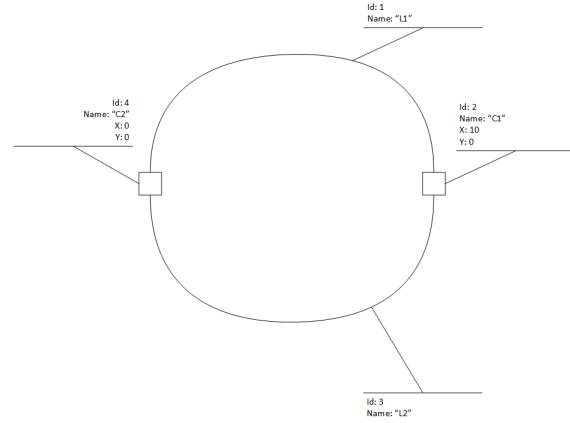


Figure 3: A simple geometry

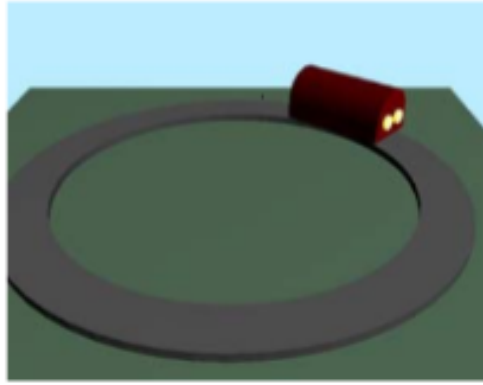


Figure 4: A simple 3D visualization

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.

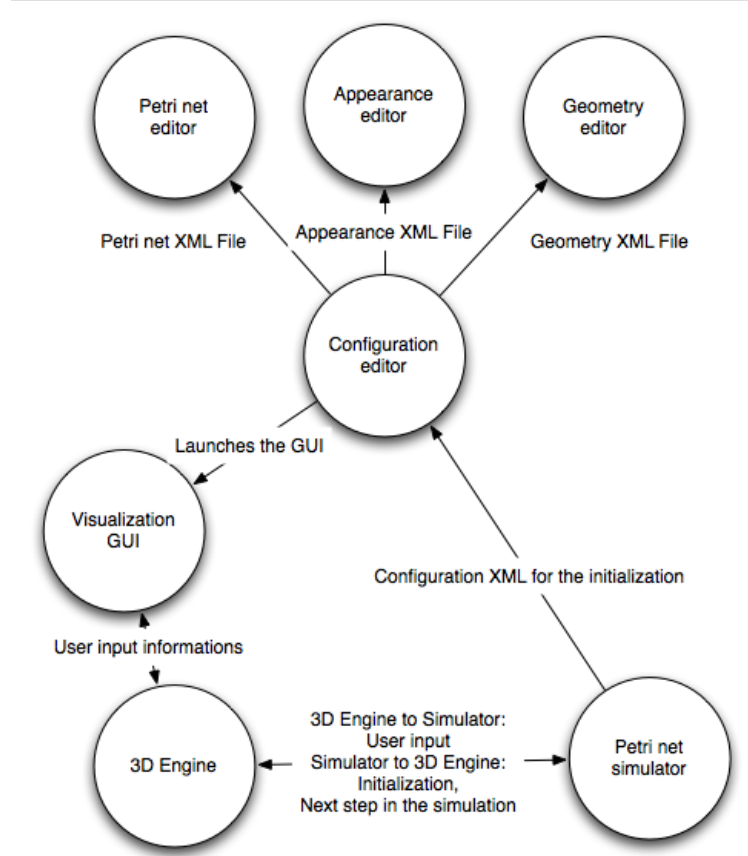


Figure 5: System design

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.

Once the files are linked together in the configuration editor, a 3D Visualization can be launched in the workbench.

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

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.

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.

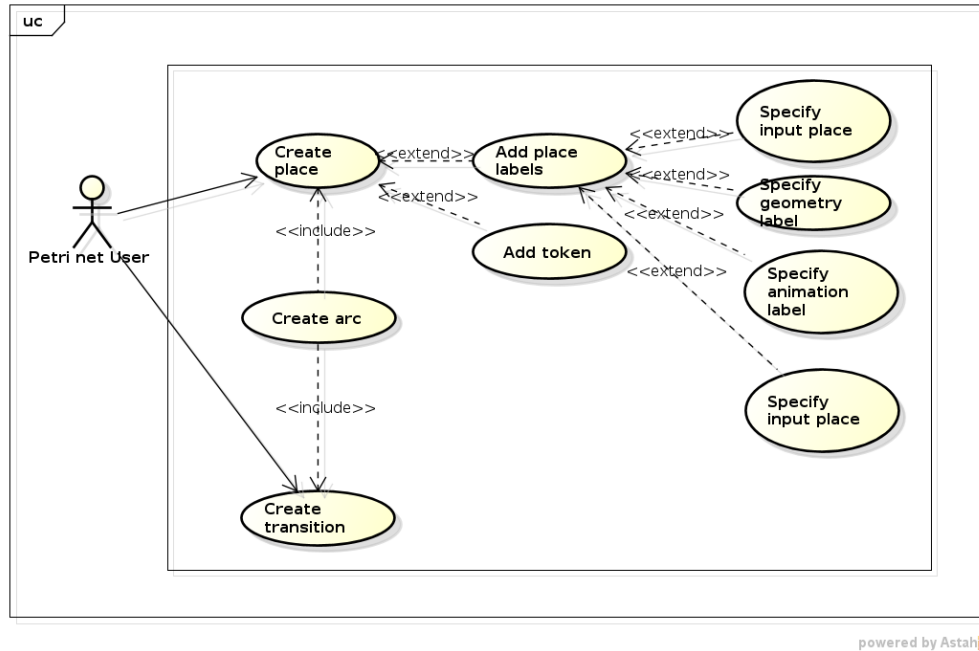


Figure 6: Use cases for the Petri net Editor

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 net 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.

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.

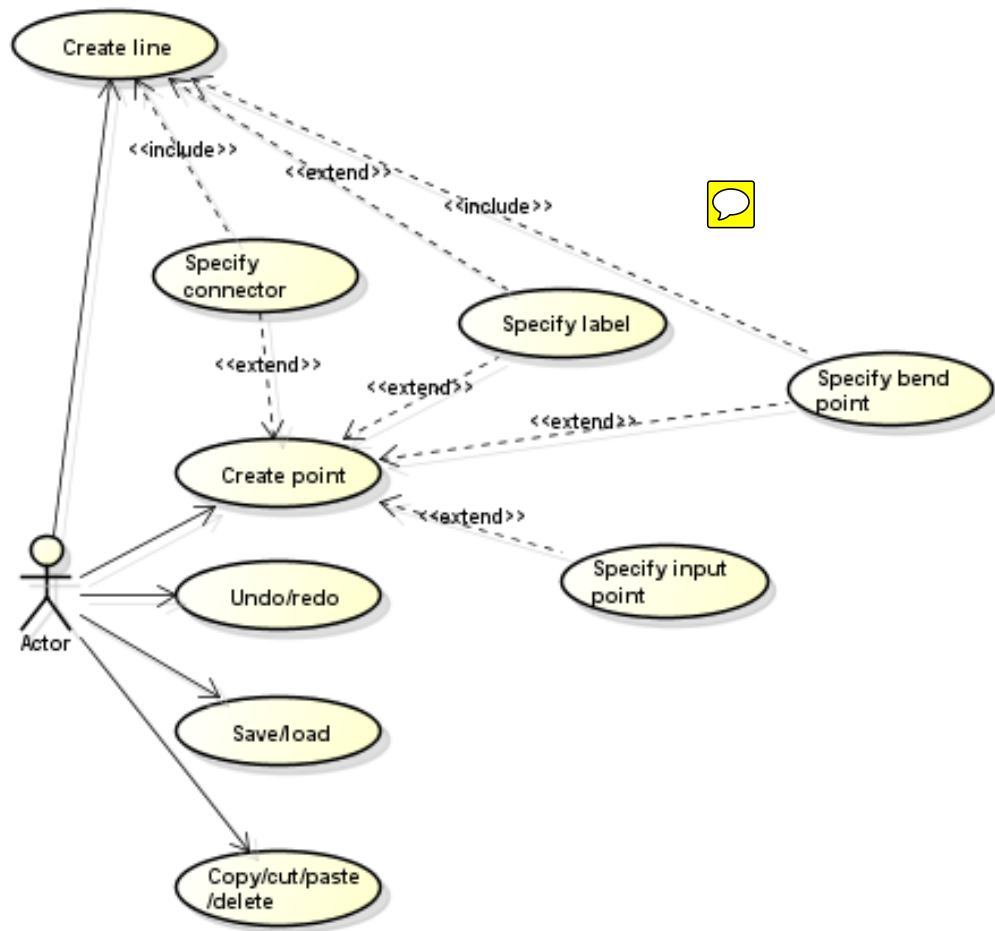


Figure 7: Use cases for the Geometry editor

3.3.2 Use cases

The features described above are shown in Figure 8.

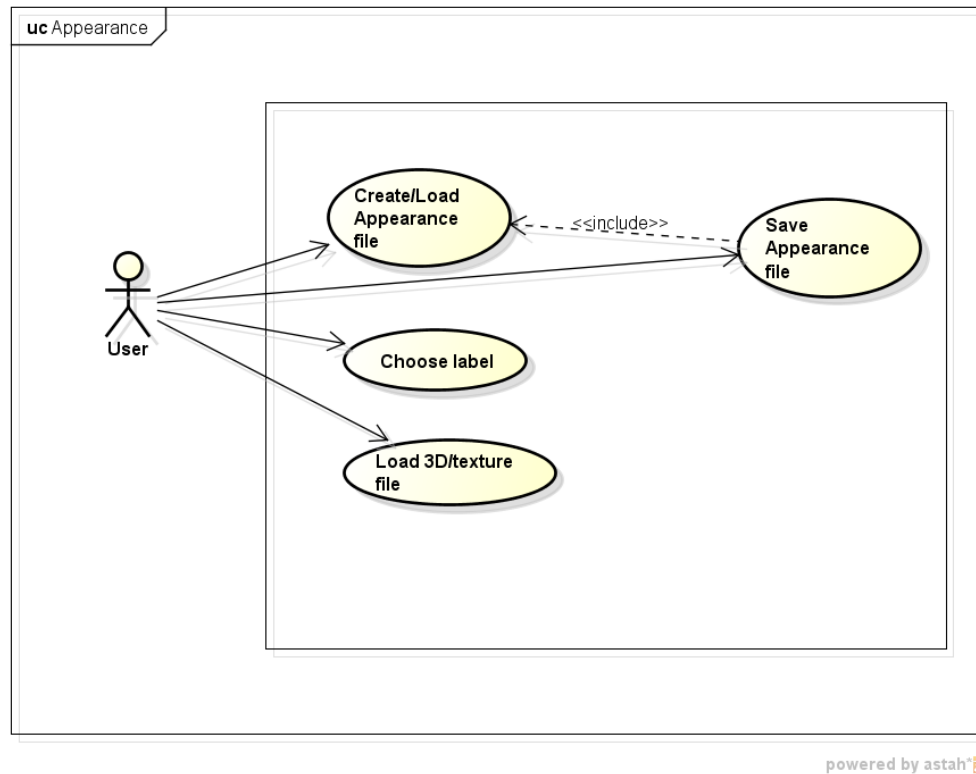


Figure 8: Use cases for the Appearance Editor

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.

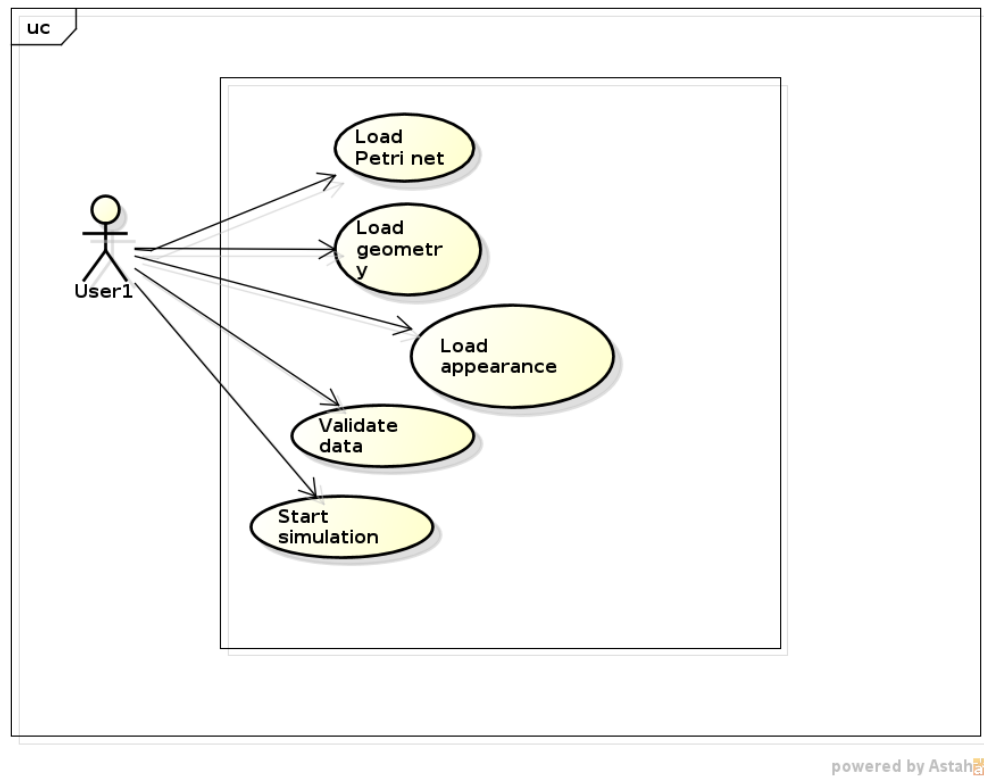


Figure 9: Use cases for the Configuration Editor

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.

4 Non functional requirements

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

4.1 Implementation constraints

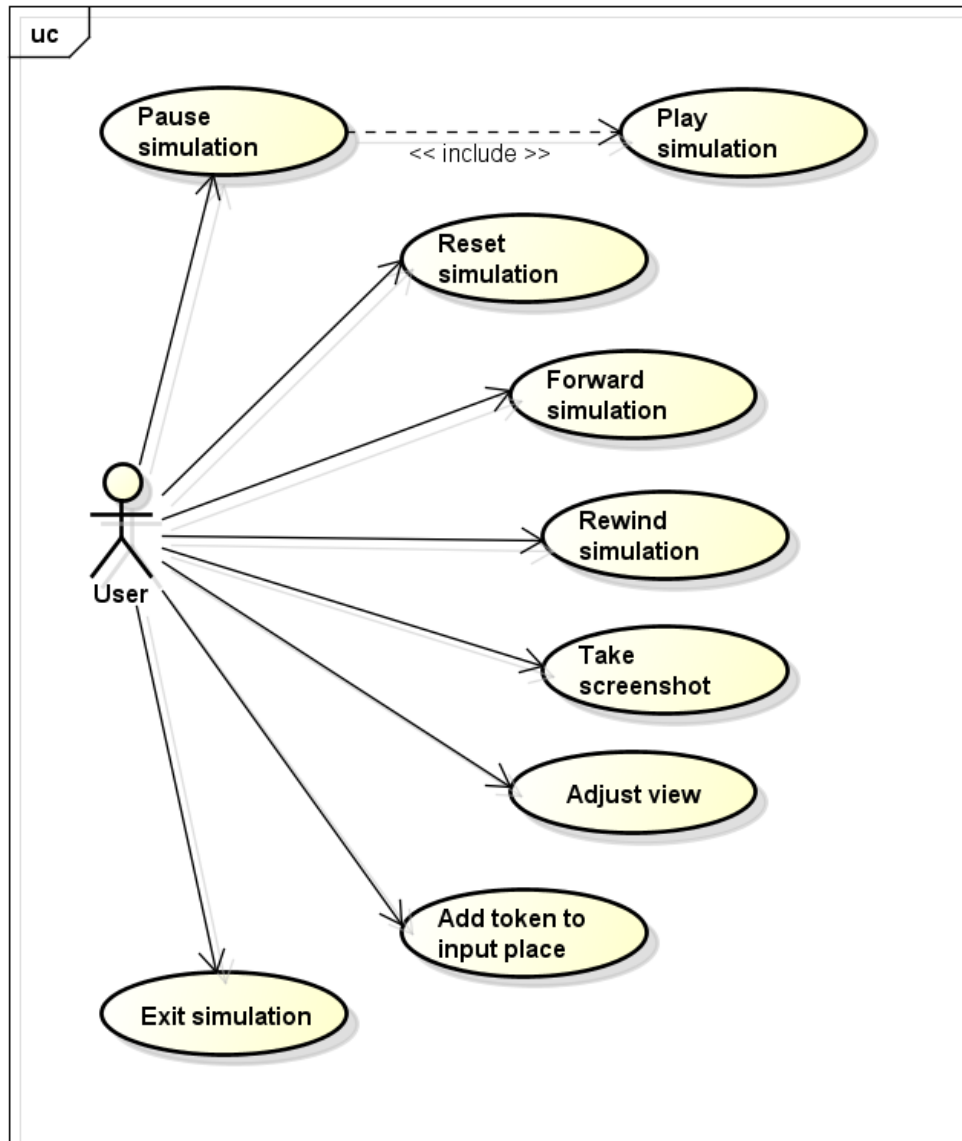
The software must be implemented as plug-in to the Eclipse framework. The software is based on following technologies:

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



powered by Astah

Figure 10: Use cases for the 3D Simulator

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 documentation is reviewed for feedback. Depending on feedback appropriate measures are taken to edit the documentation.
 - (c) Internal deadlines for documentation are set couple days in advance to the real deadline to give time to make feedback.
2. Quality of implementation
 - (a) Each implementation is produced 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. Manually made code should have good results with these tests.

5 User Interface

5.1 Technology

5.2 GUI parts

5.3 Handbook

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.

6.1 Petri net editor

Author: *Albert*

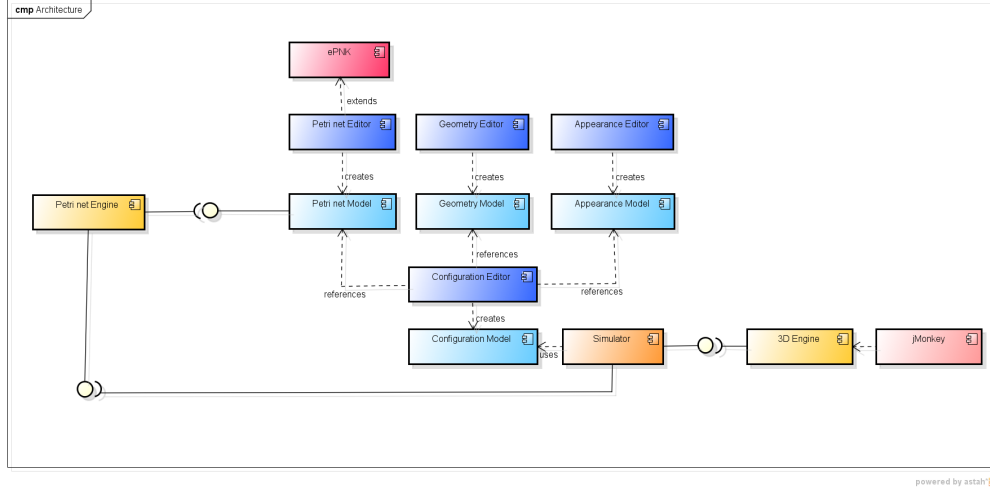



Figure 11: Architecture of the System

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 12. Its actual implementation is shown in Figure 13.

6.1.1 Petri net editor classes

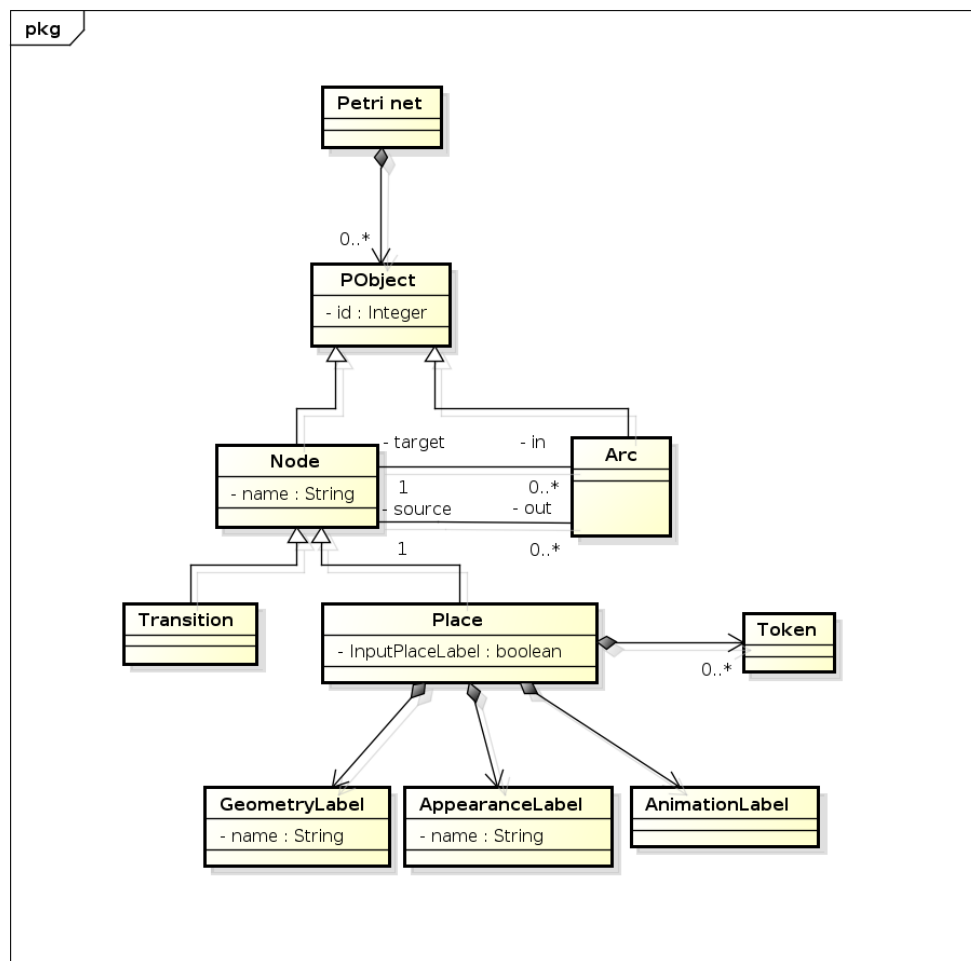
A description of the classes shown in Figure 13 is provided.

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.
- **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.



powered by Astah

Figure 12: UML for the Petri net Editor

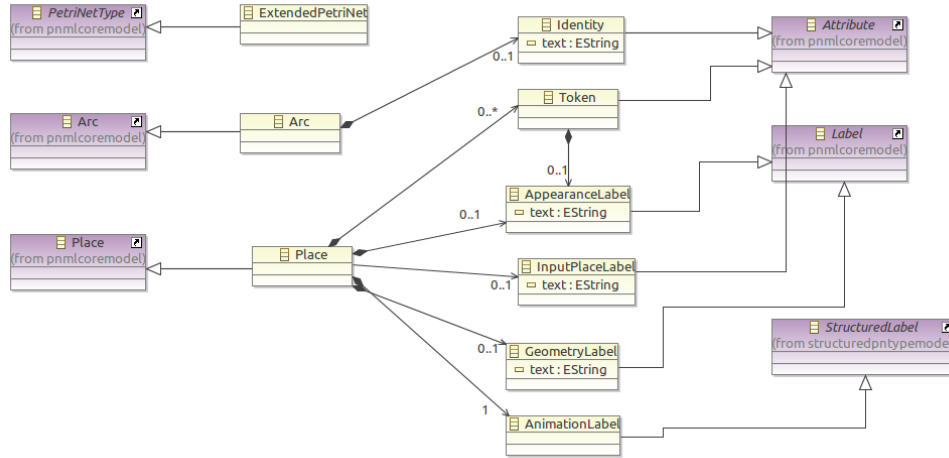


Figure 13: Petri net editor domain model

6.2 Geometry editor

6.3 Appearance editor

Author: *Monica*

The Appearance Editor is the component that 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 ?? shows the model used by the Appearance Editor component.

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.

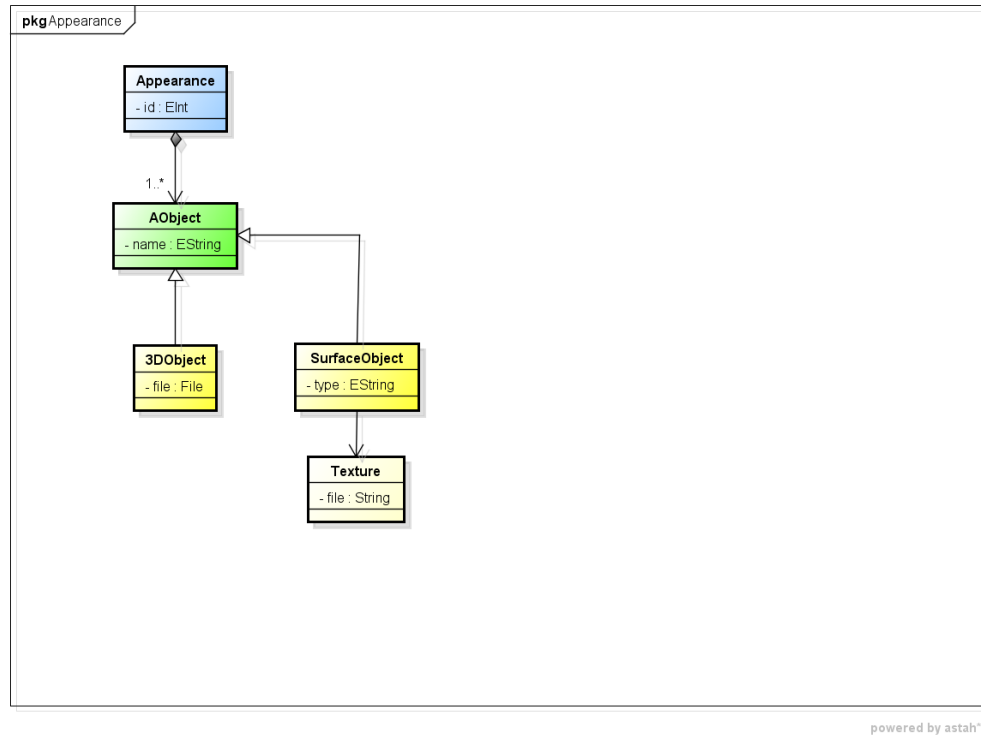


Figure 14: Appearance Domain Model

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

6.5 Petri net engine

6.6 Simulator

6.7 3D Engine

6.8 jMonkey

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 timings 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.