System Specification
ePNS

Group E

Autumn 2012

Contents

1 Introduction

Author: ***Marius***

This report documents the requirements of **ePNS**, a software system that is being developed in 02162 Software Engineering 2 course.

ePNS aims to create an easy way to visualize and interact with a simulation for a continuous dynamic system (for example, a railway system). The dynamic system will be modeled by a Petri net. To allow the modeling of continuous systems, the classical concepts of Petri nets are extended so that the new version can model more than discrete systems.

1.1 Conventions

The requirements specified in this document will be organized, according to their priority, by the following three keywords: shall, should and would be nice.

shall - describes the basic requirements, which allows the system to work properly as desired

should - describes the requirements which add functionalities that are not mandatory for the system to run, but would be important

would be nice - describes the requirements that are good ideas, but have the lowest priority; they will be implemented only if the resources allow it or in further versions of the software

1.2 Audience

The audience of this document is represented by people who are interested in Petri nets, persons affiliated with companies that model systems using Petri nets and users of **ePNS**.

2 Overall description

2.1 General product description

Author: ***Marius - Anders - Georgios***

This section contains a description of the software, as well as the actors identified and details on how the final system works.

Petri nets are a mathematical modeling language which is used for describing the operational flow of systems with dynamic and discrete behavior. Some examples of domains in which Petri nets have a clear applicability are material flow, transportation or distributed computing. However, this modeling language misses two critical characteristics that would have made it even more useful. Petri Nets do not provide a domain-specific visualization, being therefore challenging even for the most experienced users when trying to understand the full operational flow of a complex system based only on the original graphical notation. Moreover, Petri nets are discrete, implying that systems that indicate a continuous behavior are challenging to be modeled.

Some of these extension might deserve a brief presentation. In figure 1 a semaphore is shown. The dottet circle represents an input place. Input places are places where extra tokens can be placed from an external source, in this case when a user clicks an object in the simulator. In figure 1 the semaphore is green. Tracks are places with a special identifier, in this example track places are colored red as well as the arcs connecting them. Whenever a "train token" appears on the left track place L, the transition S will fire. The token in the semaphore will be consumed by T and produced again. The train token will be produced in R. If anybody clicks the object representing

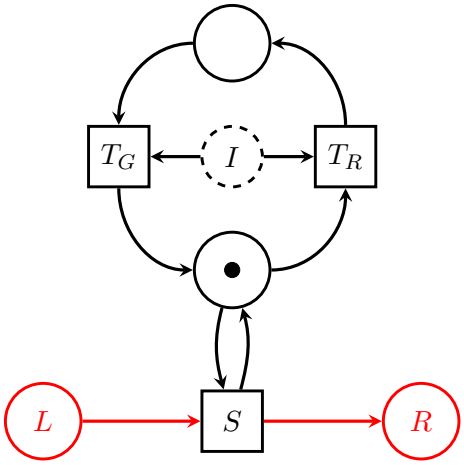

Figure 1: Petri net for a semaphore

the semaphore in the simulator, a token will be placed in I, and the T_R (Turn Red) transition will fire. Both the new token in I and the token in the lower semaphore place will be consumed. A new token will produced in the upper semaphore place. S will not be able to fire in this state of the Petri net - the semaphore is red.

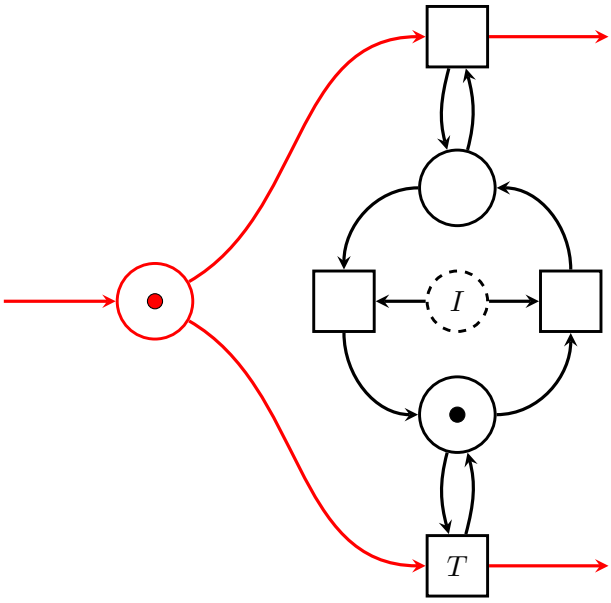

Figure 2: Petri net for a switch

The next example in figure 2 illustrates a switch I will change the switch state. As it is on the example the train token will choose the lower track. But it will wait until its animation is finished. The token is not marked as ready, illustrated by red filling (marked tokens are filled black). When the animation, probably one moving the traing from one end of the track to the other, is finished, the token will be marked (turn black) and the transition T can fire.

As a basis for our subsequent explanations and descriptions, we consider the Petri net example in Figure 3. It is a Petri net which models a railway with three signals, one switch and a train. The places, transitions and tokens are based on the ones in a classic Petri net.

For the purpose of creating a 3D visualization of systems that can be modelled as a Petri net, the application's user has to configure where the objects should be represented in the 3D space (and this is what we called "geometry") and what the objects look like ("appearance"). Continuing our example of the railway, for a proper visualization, the shape of the track has to be defined and the positions of other elements (e.g. signals, switches) have to be fixed. An example of such a configuration can be seen in Figure 4.

The ultimate goal of **ePNS** is to visualize and interact with a dynamic system by using an extended Petri net model enriched with features that will allow the user to correlate the model with the physical world. Animation and visualization of an underlying Petri net model would constitute an excellent feature that could potentially assist users in efficiently examining and analyzing the model.

The following actors have been identified:

the technical user - has knowledge of Petri nets and uses the software to design the networks according to the desired specifications.

the non-technical user -doesn't know the specifics of Petri nets, but needs to see the simulation

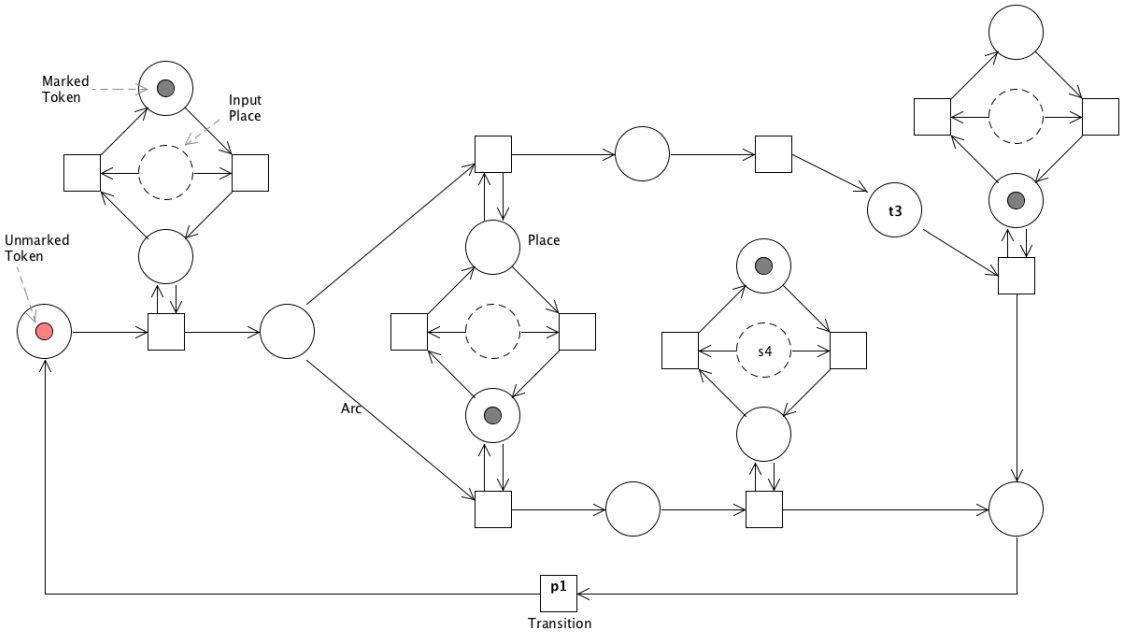

Figure 3: Petri net example

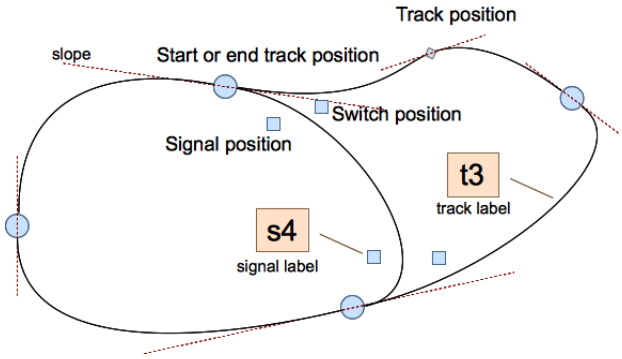

Figure 4: Geometry example

of a discrete system, so he can have a better understanding of the real-life system. This type of user will only use pre-existing Petri nets (could be Petri nets created by the first type of user), with the required additional configurations, and run the simulation.

The following case is used to prove the utility of the project in a practical scenario. An engineer could use this tool to design the extended Petri net, assigning any possible attributes to it (such as shape and identity), so as to include any possible characteristics of the physical 3D objects he might want to visualize. After the completion of the Petri net configuration, this technical user could couple the underlying model with a 3D-animation to present it to his manager. The manager does not need to have any understanding of Petri nets, but he can still evaluate the model and get a clear view of its internal functionality.

From the user's point of view, the system can be divided in three parts:

the Petri net and Animation Editor - where a user (the technical user) sets the design of the Petri net, the animation associated with each place and the appearance of a token.

the Geometry Editor - where a (technical) user sets the geometry of the system; for the railway example, in the geometry editor, the user sets the shape, length and appearance of the railway.

the Simulator - where both types of users can see a 3D representation of the modeled system.

All the components and the way they interact with one another will be thoroughly described in the following sections.

2.2 Basic functionality

Author: ***Cosmin***

In order to use **ePNS**, the application user starts by using the **Petri net Editor** to create and configure the Petri net and the **Animations** that are executed during the simulation. These will be later loaded by the **Graphical Simulator**, which, after being started, will simulate the movement of tokens through the Petri net and will execute the configured animations, resulting a visual representation of the Petri net's simulation on the screen.

Then, using the **GeometryEditor**, the **Geometry** would be created. It would be later used by the **Graphical Simulator** in order to know how (on which paths) to move objects/token representations and where to place different objects in the 3D space[1].

The **Graphical Simulator** also requires information about the **Appearance**, in order to know how to represent tokens, tracks and other objects during the simulation. It is a simple editor that connects labels (keys) with 3D Models(vrml, png, jpg ...), textures or just plain data (Colors, Shapes).

The last step is to create a **Configurator** that connects the previously created configurations and allows the user to start the graphical simulation. When started, the **Graphical Simulator** reads the state of the simulation from the *Petri net*. This read state does not include exact positions

[1]Even though the geometry is specified in a 2D space, during the simulation, all the object representations will be drawn as 3D objects moving on a plane.

of tokens in space, this information being loaded from the *Geometry*, or appearance information, loaded from the *Appearance*. After initialization, the Graphical Simulator displays the state of the simulation and handles all the users' interaction as specified in the rest of this document.

For further details of what exactly each of the components allows the users to do please check the following section or, in order to get more details regarding how to use **ePNS**, please read the *Handbook*.

2.3 General concepts

Author: ***Cosmin***

This subsection will introduce the general concepts used in the **ePNS** system. More details will be provided in the Architecture and System Features sections, however the most important concepts are presented below.

First of all, the classical concepts of **Petri nets** have been extended to accommodate the required information for the graphical visualization of the simulation:

Input Places - in order to provide the users with more power and customizability, some of the Places in the Petri net can be configured to allow users, during the graphical simulation, to drop (create) tokens. These are called *Input Places* and act as normal Places in all other respects, except for that they permit the possibility of a token being created there. For example, in a train track simmulation, it allows the creation of simulation features such as a Traffic lights or switches with which the users can interact during the graphical simulation.

Animations - can be associated by the user to a particular Place and are run when a Token is added on that Place, either by result of executing a transition or by being dropped there (after a user interaction). Even though the token is removed from the source Places of the fired Transition, they are not available for firing a new Transition until the animations associated with a place are finished. More details will be specified later, but the supported animation types include: moving an object on a path, showing or hiding objects, wait a fixed amount of time.

Place Appearance - each Place can have an associated appearance, describing how it must look like in the Graphical Simulator.

Token Appearance - each Token can have an associated appearance, describing how it must look like in the Graphical Simulator. Thus, the appearance of a Token will not change based on a place. This will allow multiple tokens, with different representations, to be on the same place/track.

Arc Identities - each arc can have attached an identity used to control the flow of tokens (or, more precisely, of token representations) in the simulation. For e.g., if we have a Transition with one input Arc and two output Arcs and we take the case of simulating a train running on a track, using the same identity on the input Arc and on one of the ouput Arcs will tell the Graphical Simulator to move the Token representation (a train), which came on the input Arc, on the corresponding output Arc. This allows a token representation to move continously in the direction the user wants, without being destroyed or unnecessarily recreated.

Regarding the **Geometry**, as defined, it allow the users to specify the positions of objects and the paths on which they move in the simulation space. The most important related concepts that need to be presented at this point are:

Track - defines how a curve (or line), on which an animation can take place, looks like. It can also be connected with information about what the surface of this track looks like and usually are used as graphical representations of Places.

Simple Position - defines just a position in the simulation space and can be connected to the an appearance it has. Can be used for completing the specification of some animations, for representing an Input Place or just for displaying simple objects during the simulation.

Referring to the **Appearance**, it allow the users to easily define how objects look like during the simulation. In **ePNS**, there are mainly two big types of appearances that can be configured:

Shape - defines how a 3D Object displayed in the simulation should look like. For example, it can be a reference to a file storing a 3D Model, which can then be loaded in the application or it can simply be a 3D Object, such as a Cube or Sphere.

Surface - defines how a surface displayed in the simulation should look like. It can be applied, for example, to a train track, and it could be either just a Color or a reference to a file containing a texture that can be applied on the surface.

3 System Features

Author: ***Juan***

In this section we will start by presenting a general overview of the software being developed, showing its main components and how they will interact with the user, as well as with each other. Afterwards, we will go into more detail in order to find out what part each component will play during the execution of the program, describing the various functionalities each one will or may provide. Specific use cases showing the interaction between each component and the users are also provided, showing how they can be used and what is offered to the users.

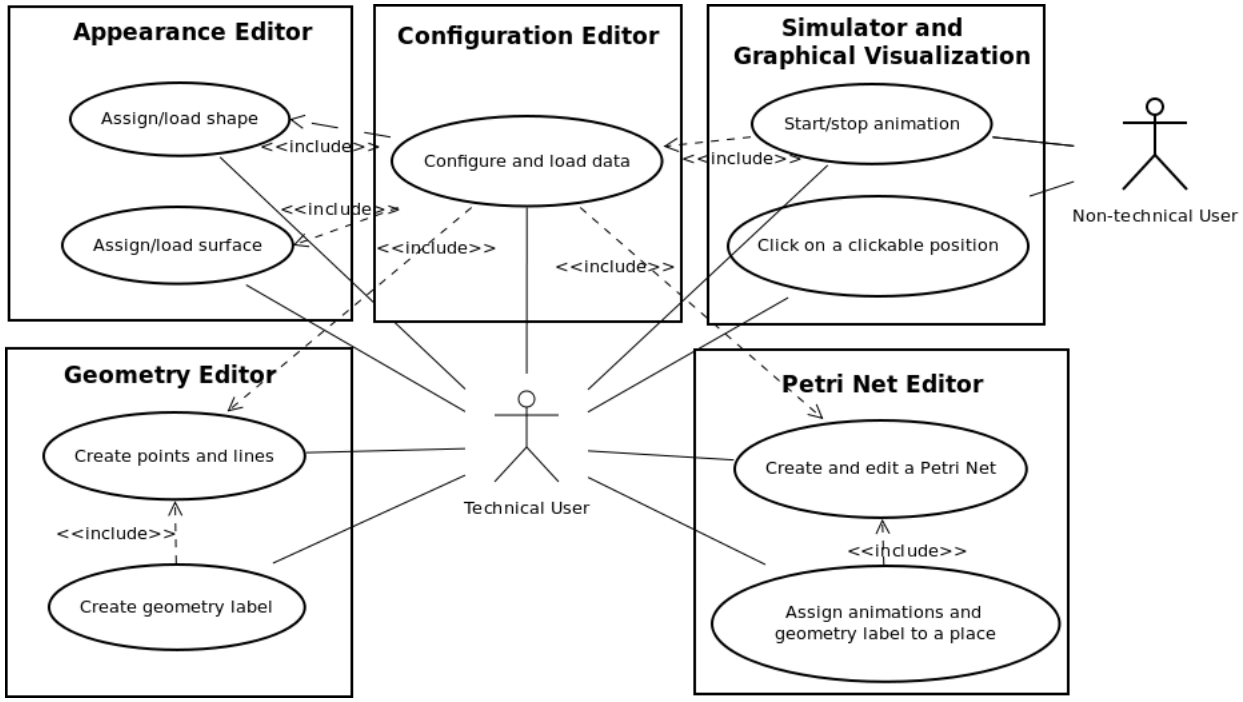

Figure 5: System Use Cases

In Figure 5 we can see the manner in which the different components of the software in development interact with each other and in what way they use one another. External to the software are the technical and non-technical user, who are given a set of functionalities to work with. As we can see, the technical user will be making use of the editors to set up everything that is needed for the Petri net simulation/visualization. He/She (or any non-technical user) will then be able to view the simulation thanks to the Simulator and Graphical Visualization component.

3.1 Petri net Editor

Author: ***Pablo***

This component will extend the ePNK tool to provide new features to the user, such as Input Places. It will also include the Animation Editor, which provides the user with a way of creating

and configurating the animations for the Petri net that will be simulated. The user will be able to assign different Animations to be executed when Tokens reach different Places of the Petri net with this functionality.

3.1.1 Functional requirements

1. The Petri net editor **shall** allow the user to create and edit Places.
2. The Petri net editor **shall** allow the user to create and edit Transitions.
3. The Petri net editor **shall** allow the user to create and edit Arcs to connect Places and Transitions.
4. The Petri net editor **shall** allow the user to create and edit Tokens inside Places.
5. The Petri net editor **shall** allow the user to create and edit Input Places.
6. The Petri net editor **shall** allow the user to assign and edit Labels to the objects created.
7. The Petri net editor **shall** allow the user to assign Animations to Places of the Petri net.
8. The Petri net editor **shall** create a Petri net and Animation Configuration file that can be read by the Simulator.
9. The Petri net editor **shall** allow the user to define a Sequence of Animations.
10. The Petri net editor **shall** establish a relationship between a defined Geometry and a Place in the Petri net.
11. The Petri net editor **shall** allow the user to save Petri net models.
12. The Petri net editor **shall** allow the user to load existing models.
13. It **would be nice** to allow the user to save a Sequence of Animations specifying a name for it to use it whenever he or she wants.
14. It **would be nice** to give the user a preview of a defined Animation.

3.1.2 Use Cases

See Figure 6 for the Petri net Editor's use cases. The main features of this editor are shown, specifying the functionalities provided by our system to allow 3D visualization.

3.2 Geometry editor

Author: ***Juan***

This component allows users to create a Geometry for the designed Petri net. It will allow him/her to define a Geometry for the Places of said Petri net.

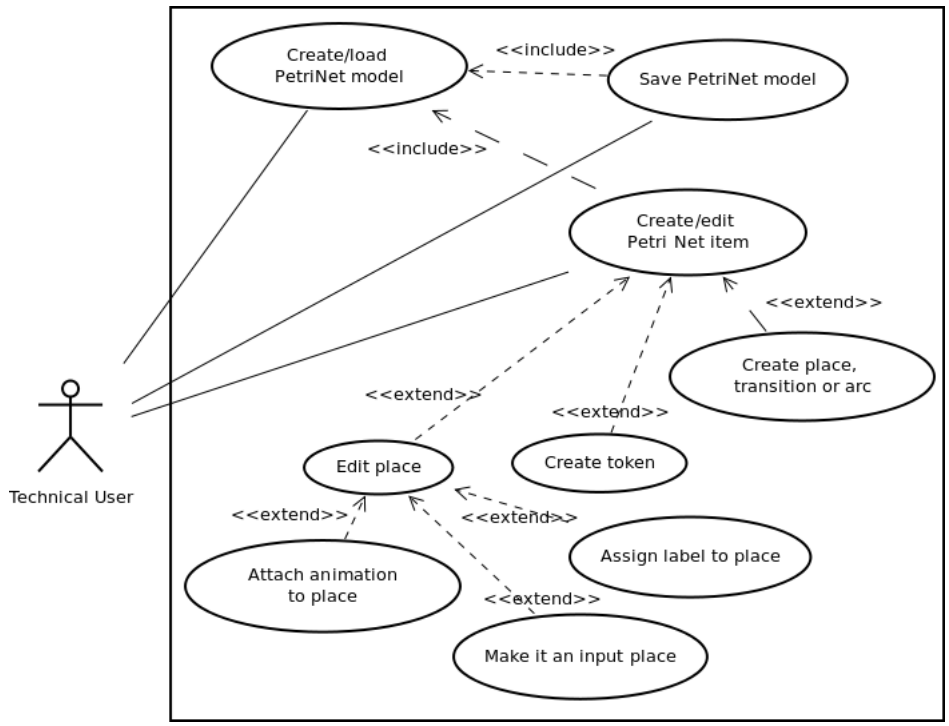

Figure 6: Petri net Use Cases

3.2.1 Functional requirements

1. The Geometry Editor **shall** allow the user to create points.
2. The Geometry Editor **shall** allow the user to create lines connecting two points.
3. The Geometry Editor **shall** create a Geometry Configuration file that can be read by the configurator.
4. The Geometry Editor **shall** allow the user to establish a relationship between a Geometry item and its Appearance from the Appearance Configuration.
5. The Geometry Editor **should** allow the user to create curves defined by a limited number of intermediate points.
6. The Geometry Editor **shall** allow the user to load a Geometry Configuration.
7. The Geometry Editor **shall** allow the user to save a Geometry Configuration.
8. It **would be nice** to allow the user to create curves defined by any number of intermediate points.
9. It **would be nice** to provide a history of the previous actions performed during the current editing process (for Redos, Undos...).
10. It **would be nice** to provide the user a set of predefined figures like circunferences or squares.

11. It **would be nice** to allow the user to save figures.
12. It **would be nice** to allow the user to assign different lengths to the figures added in order to represent real proportions.

3.2.2 Use Cases

See Figure 7 for the Geometry Editor's use cases.

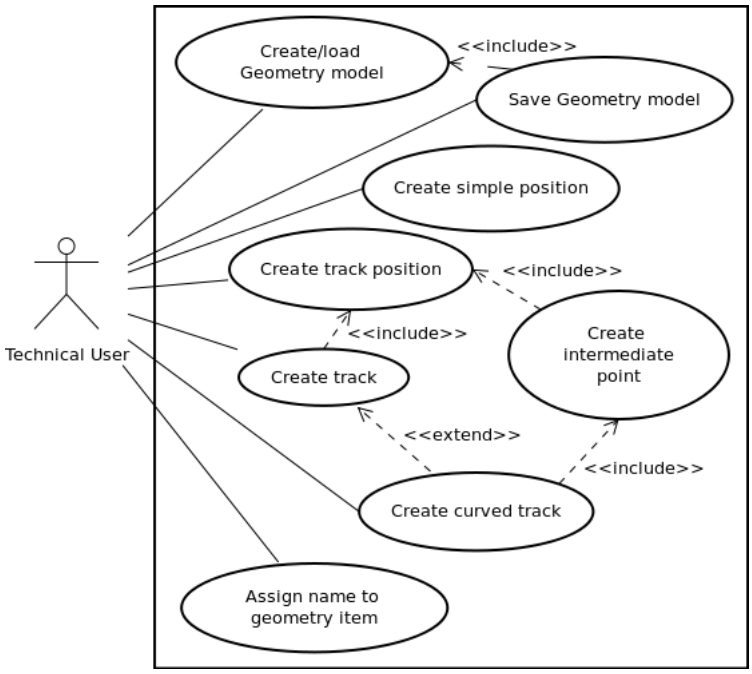

Figure 7: Geometry Use Cases

3.3 Appearance editor

Author: ***Juan***

This component allows the user to set the shape and surface of the objects belonging to the visual representation of the Petri net to be simulated.

3.3.1 Functional requirements

1. The Appearance Editor **shall** allow the user to create 3D objects by choosing a predefined simple shape.
2. The Appearance Editor **shall** allow the user to create surfaces by choosing a predefined color.

3. The Appearance Editor **shall** create an Appearance Configuration file that can be read by the Configurator.
4. The Appearance Editor **shall** allow the user load an Appearance Configuration.
5. The Appearance Editor **shall** allow the user to save an Appearance Configuration.
6. The Appearance Editor **should** allow the user to create 3D objects by referencing an existing 3D model.
7. The Appearance Editor **should** allow the user to create surfaces by referencing an existing texture.
8. The Appearance Editor **should** allow the user to assign a surface to a 3D object.
9. It **would be nice** to provide the user with a collection of pre-defined complex shapes.
10. It **would be nice** to allow the user to choose the colour from a palette.

3.3.2 Use Cases

See Figure 8 for the Appearance Editor's use cases.

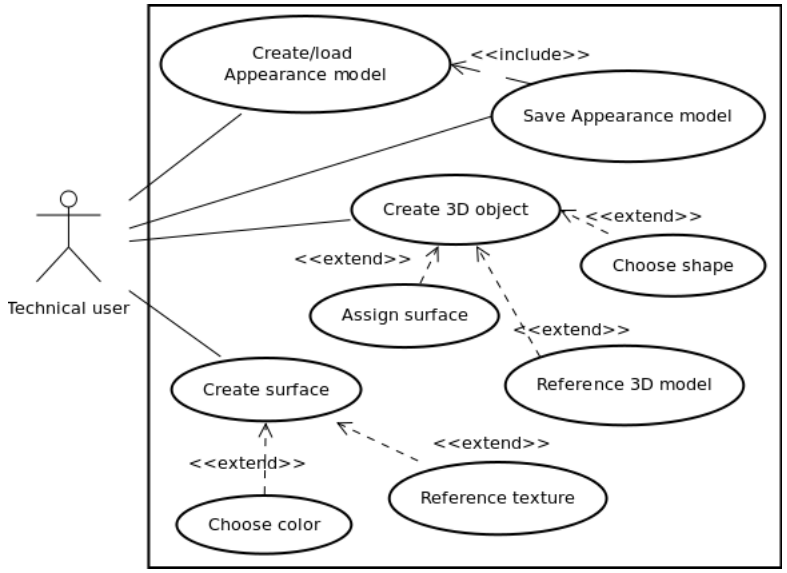

Figure 8: Appearance Use Cases

3.4 Configuration Editor

Author: ***Juan***

This component allows the user to configure a simulation, by connecting the Petri Net, Geometry and Appearance files used in the simulation and visualization.

3.4.1 Functional requirements

1. The Configuration Editor **shall** allow the user to set a reference to the file containing the Petri net and Animations Configuration.
2. The Configuration Editor **shall** allow the user to set a reference to the file containing the Geometry.
3. The Configuration Editor **shall** allow the user to set a reference to the file containing the Appearance.
4. The Configuration Editor **shall** allow the user to start the visualization of the simulation with the configured data.
5. It **would be nice** to allow the user to set some aditional attributes that are not related to a concrete editor (e.g. default track width or background color).
6. It **would be nice** if, once the Petri net is loaded, the Geometry and Appearance Configurations are detected automatically based on name and extension.

3.4.2 Use Cases

See Figure 9 for the Configuration Editor's use cases.

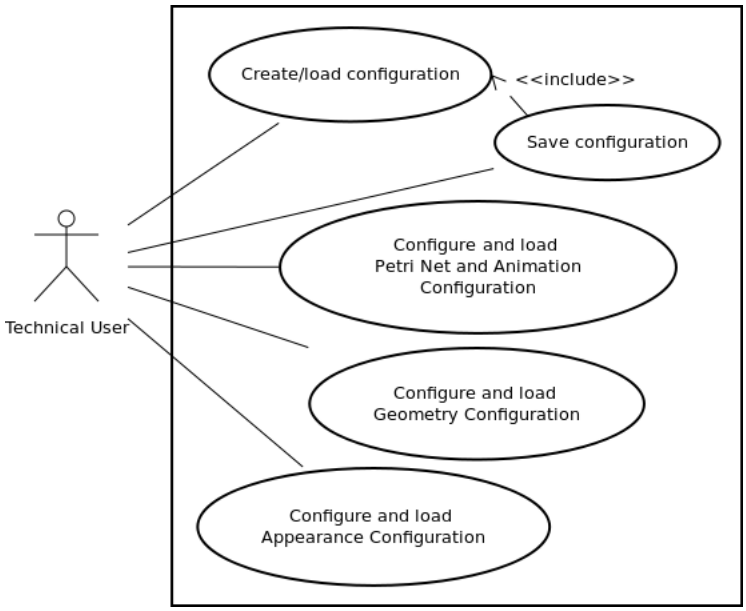

Figure 9: Configuration Use Cases

3.5 Simulation and Graphical visualization tool

Author: ***María***

This component simulates the behaviour of the Petri net and provides a 3D visualization of the objects defined for it, as well as the corresponding animations.

3.5.1 Functional requirements

1. The graphical visualization functionality **shall** show a 3D visualization of the Geometry defined in the Geometry Configuration.
2. The graphical visualization functionality **shall** show a 3D visualization of the Animations.
3. The graphical visualization functionality **shall** detect when a user interacts with the visualization, i.e. clicks on a clickable position.
4. The graphical visualization functionality **shall** allow the user to start/stop the animation.
5. The graphical visualization functionality **should** allow the user to pause/restart the animation.
6. The simulation functionality **shall** simulate the given Petri net.
7. The simulation functionality **shall** allow the user to start as many simulations of the same Petri net as he or she wants.
8. It **would be nice** to load a Petri net at initialization and allow the user to edit it in parallel without interfering with the simulation.
9. It **would be nice** to offer a visualization of the changes in the Petri net during the simulation.

3.5.2 Use Cases

See Figure 10 for the Simulation & Graphical Visualization tool's use cases.

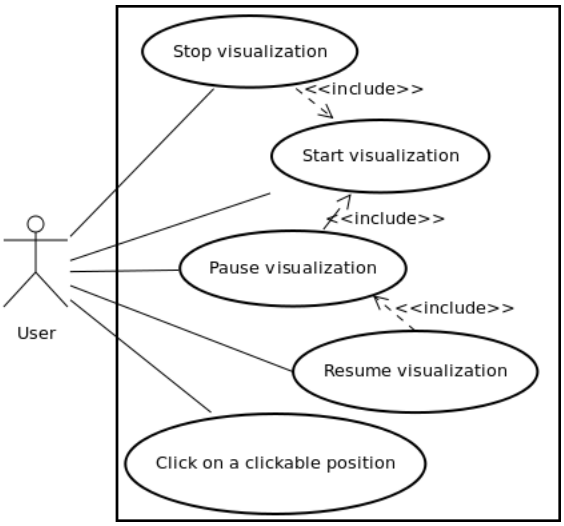

Figure 10: Simulation and Graphical Visualization Use Cases

4 Non-functional requirements

Author: ***María***

The purpose of this section is to specify a number of requirements that are not related with the domain and analysis part. These requirements are divided in the three following subsections.

4.1 Implementation constraints

The system shall be implemented as a plug-in for the Eclipse framework version Juno 4.2. It is a requirement to build the system using the following technologies/tools:

- EMF - (Eclipse Modeling Framework) v. 2.3.0
- GMF - (Graphical Modeling Framework) v. 2.0.1
- ePNK v. 0.9.4
- Java 3D v. 1.5.1

4.2 Documentation

During the development process the following documents shall be delivered before the specified deadline:

- Project definition - 05/10/2012
- UML diagrams - 12/10/2012
- System specification - 02/11/2012
- Draft version of handbook - 23/11/2012
- Final documentation - 21/12/2012

4.3 Code deliverables

Some intermediate prototypes and experiments are required to be delivered:

- Technology experiment - 26/10/2012
- First prototype - 16/11/2012
- Feature complete prototype - 30/11/2012
- Final software - 21/12/2012

4.4 Quality

In order to ensure the quality of the product, a set of procedures have been established. These procedures are different depending on whether a document or a piece of code is being developed.

4.4.1 Quality of documentation measures

1. Before starting to write a new document, some members of the group shall define the main structure and the number of people working on each part. Afterwards, a brief explanation shall be done to the rest of the group.
2. A first draft of the document should be ready at least a week before the deadline, so that all the members of the group can read it and propose improvements.
3. A final version of the document should be prepared at least two days before the deadline, so that all the members of the group can check writing and consistencies.
4. Once the document is approved, one member of the group shall be in charge of formatting it.
5. One member of the group shall be in charge of delivering it before the specified deadline.
6. The document shall be changed and checked as described above based on the feedback received by the professor.

4.4.2 Quality of implementation measures

1. All implementations should be reviewed by at least two members of the group, so that errors can be minimized.
2. Every version of the code shall be committed to the SVN repository, with a comment about the changes made.
3. Each component should be tested by a member of the group that has not taken part in the implementation, to ensure that every possible scenario is covered. Unit testing should be used for this purpose when possible and acceptance testing when not.
4. Integration tests shall be carried out in order to check the correct behaviour of each of the pieces connected.
5. System tests shall also be used to check the correct behaviour of the whole product.
6. Every piece of code shall have comments explaining the steps, in order to allow the rest of the group to understand it.
7. It would be nice to create a programming style document before starting the implementation process.
8. Javadoc shall be used to provided thorough documentation of the code.

5 User interface

Author: *Jerome*

This chapter describes the main aspects regarding how the User Interface of our Software product looks.

5.1 Technologies

The GUI shall be implemented using Eclipse plugins. These plugins are of various types and together work to obtain the desired functionality. The used types are:

Editor - used for the creation and edition of the different resources (Resource Editor) to design the networks according to the desired specifications.

View - show the different properties of the resource currently being edited (Diagram Overview).

Pop-up - used to add extra functionalities (Right-click Menu).

5.2 GUI parts

In this section, we will talk in more details about all the different parts of the global editor running in Eclipse that will be provided with the software. Throughout this document we will refer to this as the Workbench. The Workbench is composed of five main parts as shown in Figure 11: the Project Explorer, the Resource Editor, the Properties View, the Diagram Overview and finally the ToolBar.

The most important components are the Project Explorer, the Resource Editor and the Properties View because they allow the user to interact with the software, creating new parts or editing them. In the following sections a short description of all the different parts will be provided.

5.2.1 Project Explorer

The Project Explorer is a standard Eclipse view. It provides the user with an overview of the project showing the different folders and files composing it, with their extensions. By selecting a file and pressing Enter, the view will determine what type of file the user has selected and open the Editor associated with this type of file (see Figure 12).

5.2.2 Resource Editor

This part of the Workbench is an Editor, more precisely a set of Editors. In the following figure you will see a representation of the Petri net Editor, which allows the user to create, design and modify Petri nets. (See Figure 13).

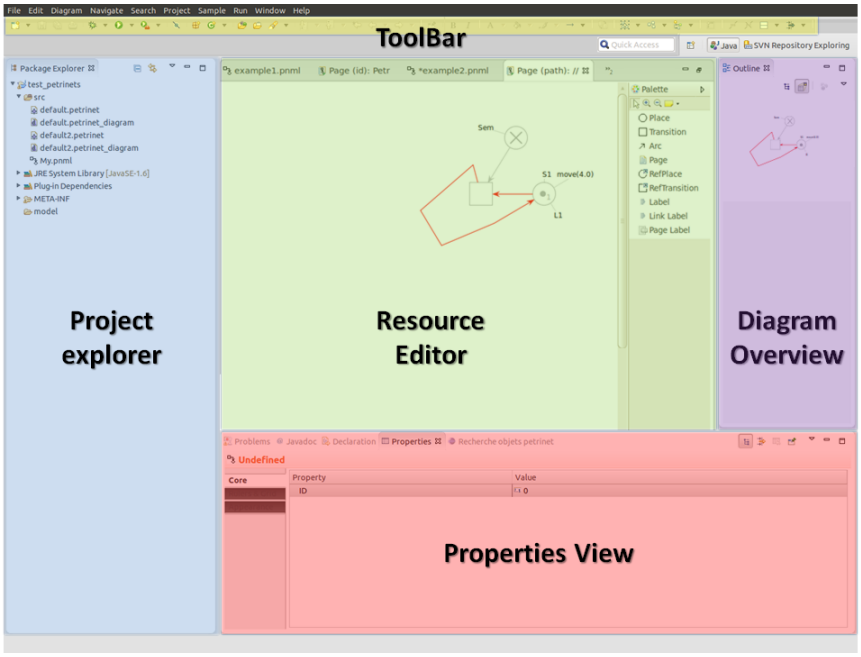

Figure 11: Workbench

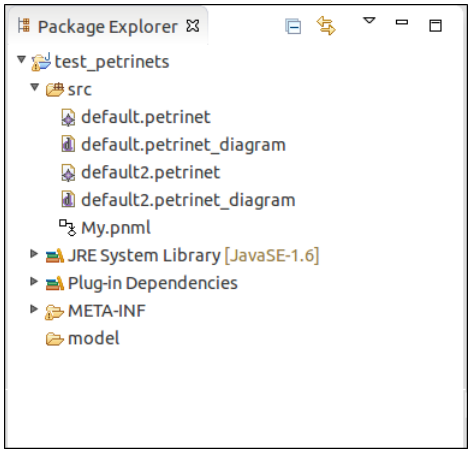

Figure 12: Eclipse Project Explorer

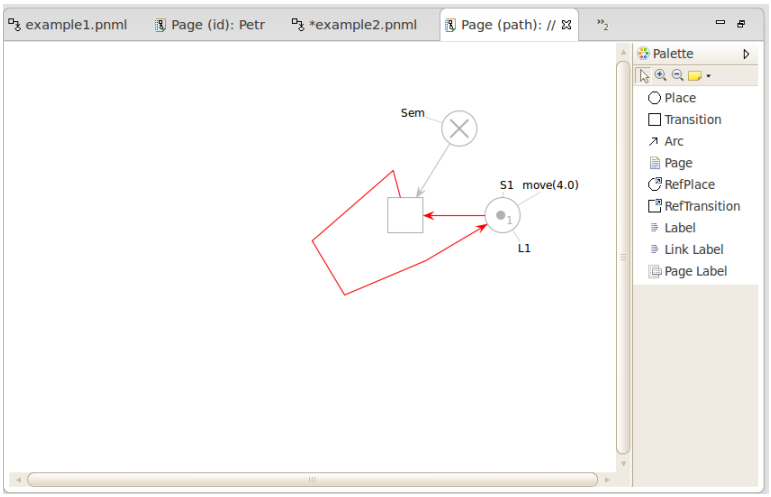

Figure 13: Petri net Editor

The Resource Editor is split into two parts : the left area contains a graphical representation of the resource the user is editing while the right area displays a Palette of elements that the user can drag and drop to modify the graphical part.

In the graphical representation, the user is allowed to move elements, resize them and edit their name by double-clicking on them. The Palette contains some other useful tools such as zooming and adding notes in the graphical representation so the user can explain to another user some parts of his representation. In addition to the Petri net Editor, we also have three other Editors provided with our software: the Geometry Editor that allows the user to draw a shape (for example, the shape of the track) that will be associated with the Petri net for the simulation (See Figure 14); the Appearance Editor that provides a way to edit the appearance of Places and Tokens; and the Configuration Editor that keeps a reference to the Petri net Model and the Geometry Model to configure the simulation.

5.2.3 Properties View

The Properties View is another one Eclipse built-in plugin displayed as a view. This view is connected to the Resource Editor and displays some additional properties depending on the selected object. The goal of the Properties View is to edit the properties of different objects of the diagram. The Figure 15 below will for example provide you the Properties View associated with the selection of a Transition in our Petri net editor.

5.2.4 Diagram Overview

One more useful Eclipse plugin is the Diagram Overview, also implemented as a view. The particularity of this view is that its goal is to show a global view of the Diagram we are editing with a box symbolizing the part we are actually seeing in the Resource Editor. As a diagram can be a lot bigger than the user's screen this plugin is used to navigate through the Diagram as the Resource Editor

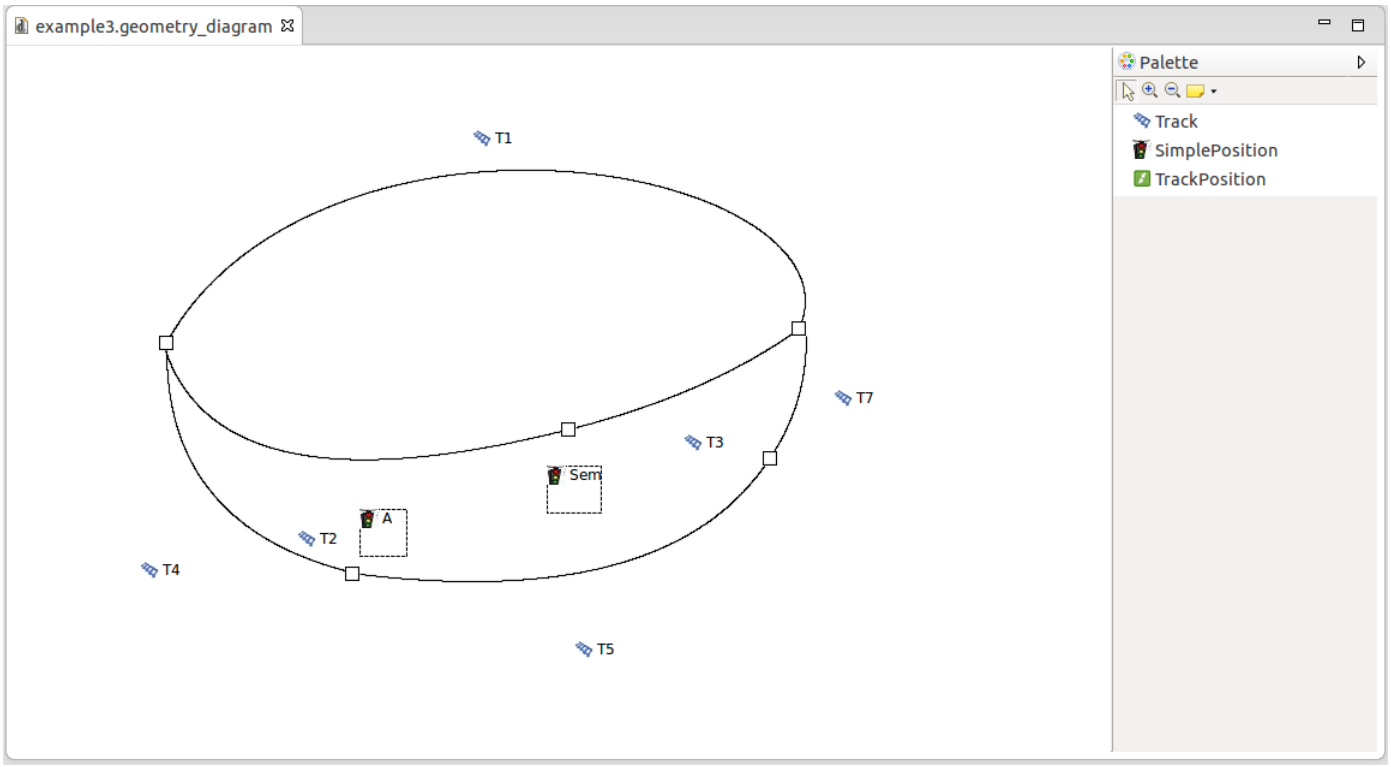

Figure 14: Geometry Editor

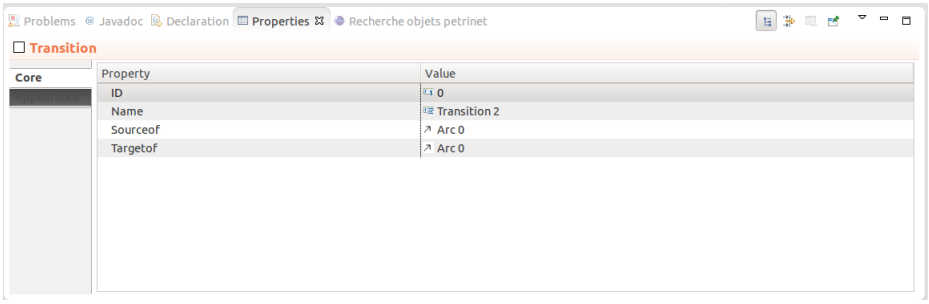

Figure 15: Property View

will display what is inside the box and we can move the “seeing box” dragging it in the Diagram Overview (see Figure 16).

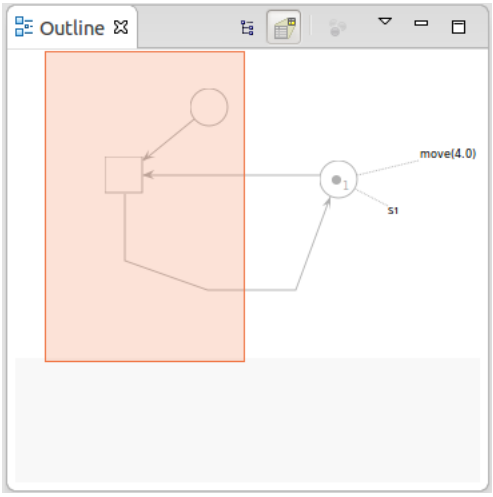

Figure 16: Seeing box

5.2.5 ToolBar

The ToolBar is the area in Eclipse where all the general functions are placed. This includes commands like creating, saving and loading, next and previous page, etc (see Figure 17).

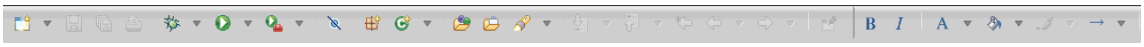

Figure 17: Toolbar

5.2.6 Right-click Menu

This global menu (in the sense it can be accessed everywhere) is not shown in our Figure 11. It can be triggered by the user pressing the right button of his mouse. It will show some relevant options such as delete object, or start some actions. Indeed each content of the menu will depend of the type of selected component. As an example, we can show the "Start Simulation" action that has been implemented in **ePNS**. This function is only available when a right-click is performed on a “Configurator” object and allows launching of the Simulation and the Visualization for the Petri net (see Figure 18).

5.2.7 Simulator and Graphical Visualization

This last component is also not shown in Figure 11 as it is a window detached from Eclipse. It provides a 3D visualisation of a chosen Petri net using the Geometry Model and the Appearance Model that have been set up earlier. It has two main buttons: Start/Stop and Pause/Resume

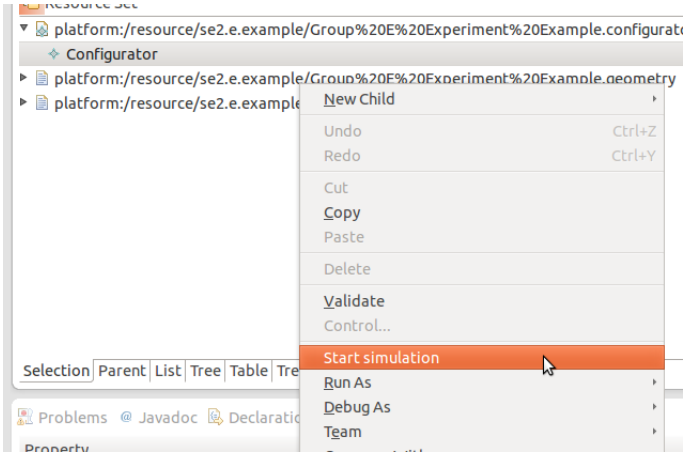

Figure 18: Right-click Menu

allowing the user to control the simulation & visualization. The user may also interact with the tool by clicking on the interactive input places in the simulator to trigger events, for example to change the status of a a traffic light from the red to the green state (see Figure 19).

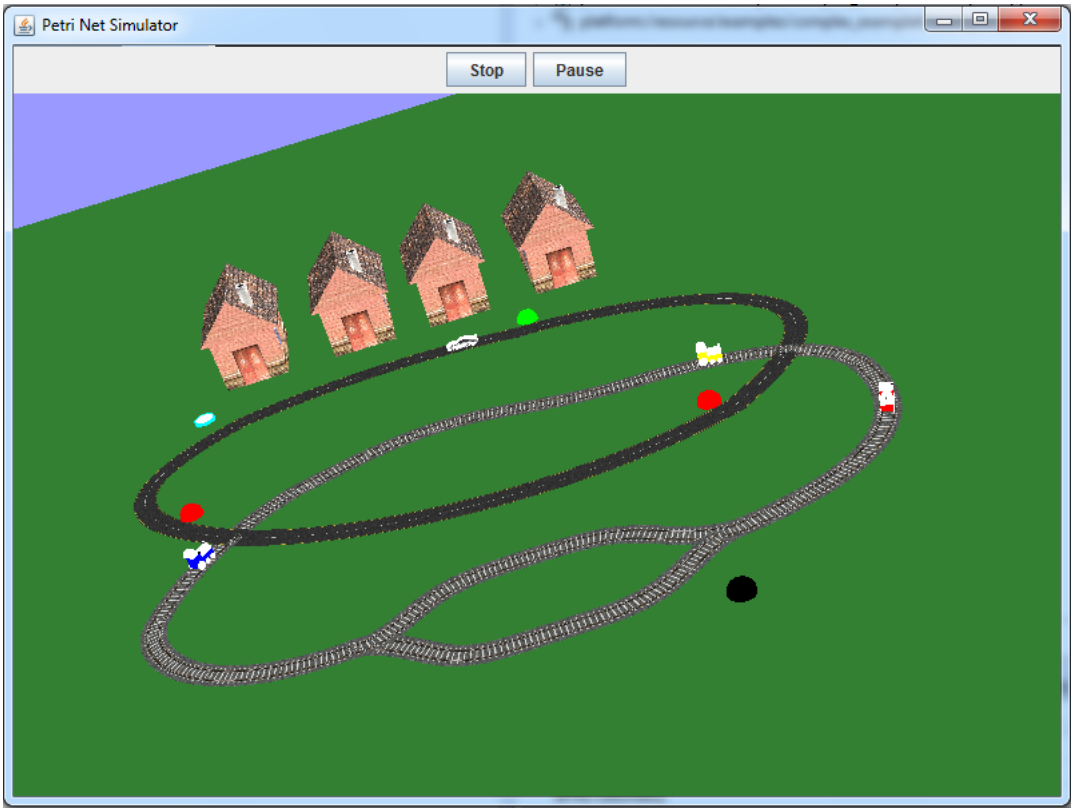

Figure 19: The Simulator

6 Architecture

Author: ***Cosmin, Ruxandra, Anders - Intro Part***

The overall architecture is based on the principle of modularity. The system is separated in several components, with well defined interfaces. This approach has several advantages:

- The development of components can be easily delegated to different group members. As long as the interfaces remain the same, changes can be made to the component behind that interface with minimum influence on other components.
- Components can be easily tested
- Due to the interfaces, components can easily be mocked, therefore making it possible to test the overall system in a structured manner.

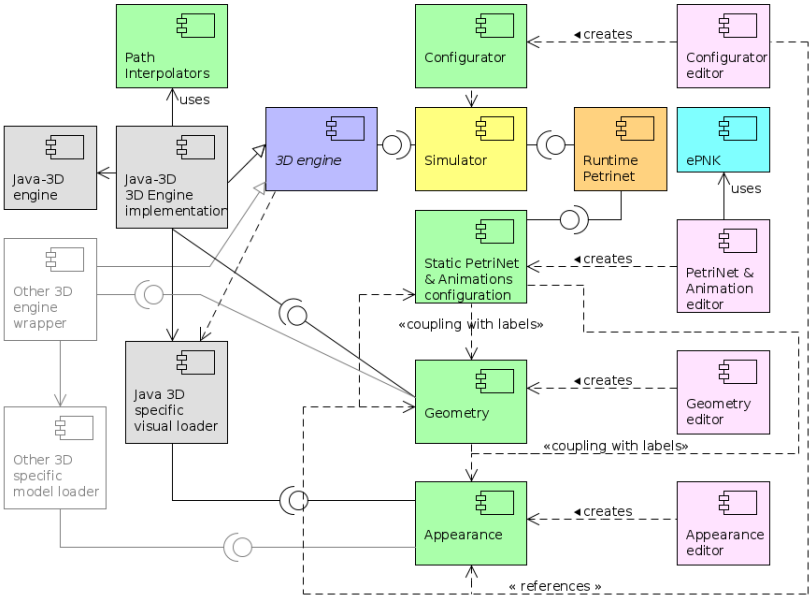

Figure 20: Components diagram

A general view of the domain model is presented in figure 20. A short description of the most important components will be provided in the next paragraphs. More details are available in the sections dedicated to each component.

In order to use our system, the application user begin by using the **Petri net Editor** to create the **Static Petri net**. When starting the **Simulator**, it would load the **Static Petri net** and create a **Runtime Petri net**. This runtime Petri net contains the information about Places, Arcs and Transitions, but also contains up-to-date information regarding the tokens present in each Place(a hash-map).

The way the tracks look like is defined in the **Geometry editor**. The settings from the geometry editor will be used by the 3D Engine in order to know how (on which paths) to move objects/tokens

when instructed by the Simulator. This **Geometry editor** allows the usee to create paths (curves) in the 3D space, which will be used when animating objects during the simulation. Moreover, it allows the user to add positions on which to place various objects(like traffic lights, trees, etc.) in the 3D space.

The 3D Engine also requires information about how the objects will look like, in order to know how to represent tokens, tracks and other objects during the simulation. This information is set up in the **Appearance editor**.It is a simple editor that connects labels with 3D Models(.3ds and .obj), 3D Shapes, Textures or Colors.

For setting up animations for the objects in the simulation we will use an **Animation Editor**, which, in our project is integrated in the Petri net editor to create an easier and more intuitive interface for the user. The user can set up a single animation or a sequence of animations which is loaded by the Simator and is then used by the 3D engine to display the objects.

The **Simulator** is a component that supports the simulation of a Petri net. In order to do so, the Simulator needs a **Petri net** to simulate, a **3D engine** to display the simulation on and the settings provided by the **Geometry editor**, **Appearance editor** and the integrated **Animation Editor** to know how to map Petri net concepts to 3D coordinates.

The **3D engine** is an abstraction of 3D engines with focus of Petri net simulations of objects on tracks (see project definition). The idea is to facilitate various 3D engines: Java3D, OpenGL, HTML5 without having to reimplement everything. Our system will make use of an abstract factory for the wrapper implementations and their 3D models.

In Figure 21, we represented the interaction between the main components active during a running simulation & visualization. The Simulator is the main component, which is started by the user and has access to all the configuration files necessary. Then, the 3D engine is started by sending the Geometry and Appearance configurations and the Input Places. Next, the Runtime Petri net is initialized based on the static Petri net created by the technical user. The actual simulation is started by the user, by clicking the Start button, event which is captured by the 3D Engine. The latter notifies the Simulator that the simulation was started, which, in turn, starts the initial animations.

The simulation runs in a loop where animations are started for tokens on places. Whenever an animation ends, the 3D Engine notifies the Simulator and the corresponding token in the Petri net is marked as finished. If an interactive input Place has been activated (clicked upon the associated 3D Object), then a token is created in it. In either case, the RuntimePetriNet is asked to fire any waiting transitions [2]. A collection of moved tokens and the animations that have to be run is returned. Using this, the Simulator might need to start new animations or to destroy representation for tokens that do not exist in the simulation anymore[3]. Then the loop starts over.

The simulation stops when a special place (e.g. a Stop button) is pressed in the 3DEngine, which, in turn, notifies the 3D Engine accordingly.

In the rest of this chapter, more details are provided for each of the components used in **ePNS** and general implementation details are discussed.

[2]Considerations should be made on infinite loops by the technical user configuring the network.

[3]More details will be provided in subsequent sections.

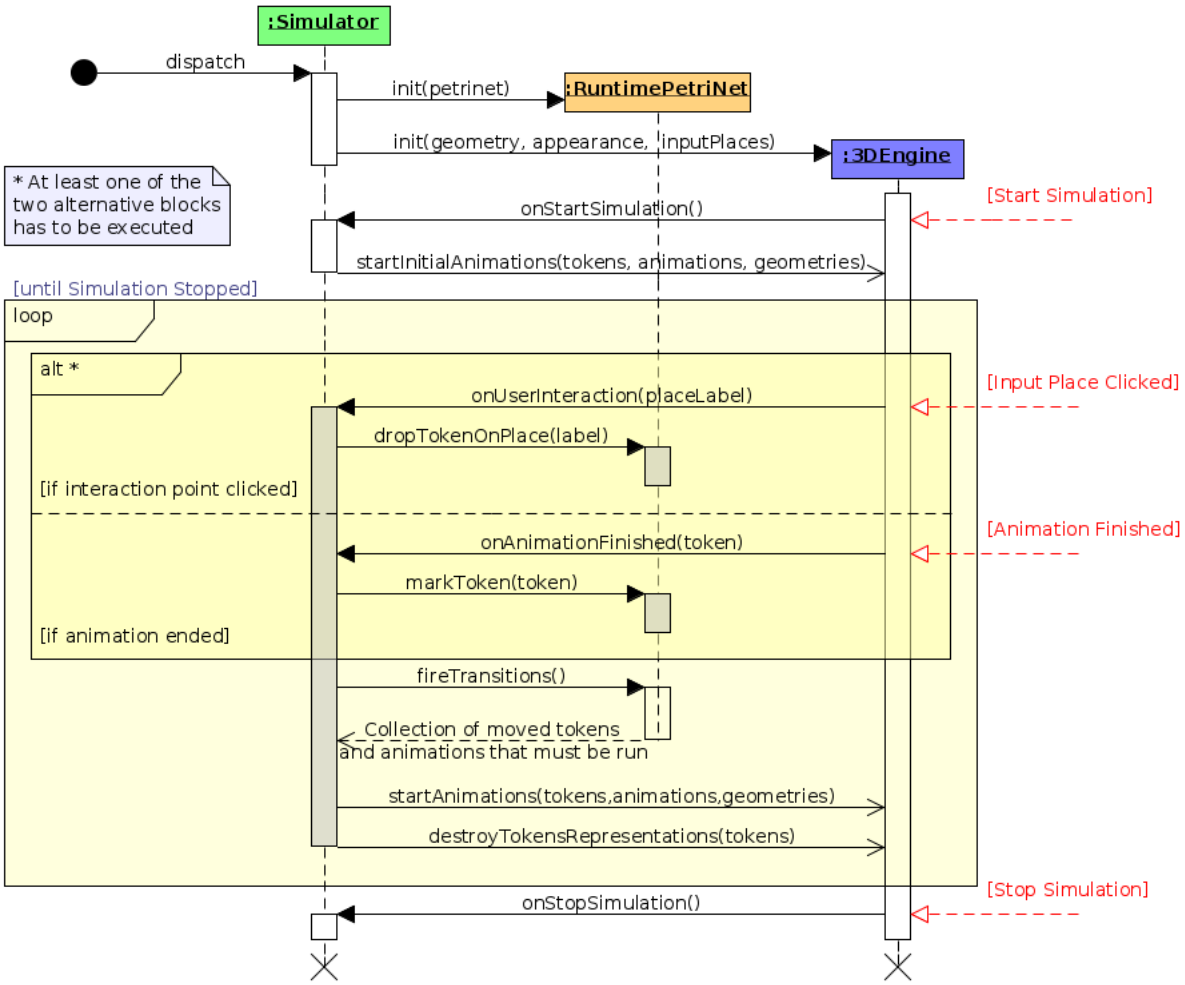

Figure 21: System Sequence Diagram

6.1 Petri net Editor

Author: ***Pablo***

As previously stated, this component is logically separated into two parts: the Petri net Editor itself (see figure 22) and the Animation Editor (see figure 23). Therefore, we will describe them separately.

6.1.1 ePNS Petri net classes

Here are described the classes that define the components of an ePNS Petri net according to the model used for the software in development (the model is shown in figure 22).

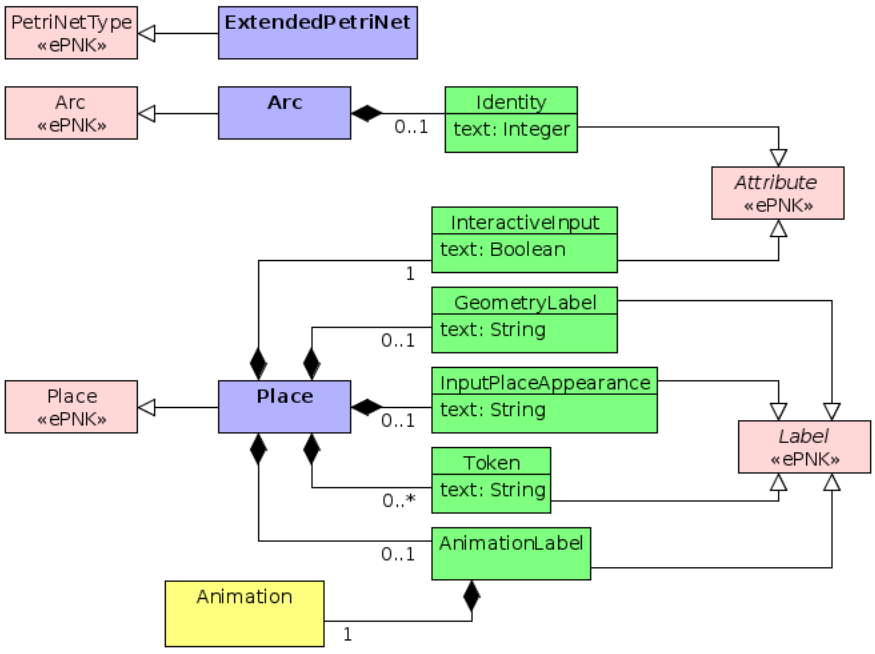

Figure 22: Petri net Domain Model

Arc class
The Arc class has the following attributes:

> **Identity:** an ePNK Attribute with an integer that indicates the Arc's identity (used for Token flow, for more information see "Arc Identities" in 2.3).

An Arc connects Places to Transitions and vice versa, as well as determining (thanks to their Identity attribute) which way Tokens must go when "going through" Transitions.

Place class
The Place class has the following attributes:

> **InteractiveInput:** an ePNK Attribute with a boolean value indicating whether Tokens can be inserted into the Place (for more information see "Input Place" in 2.3).
>
> **GeometryLabel:** an ePNK Label with a string containing the name of the Geometry object that will represent the Place (this will typically be a line or a point in the geometrical description; for more information see "Track" and "SimplePosition" in 6.2).
>
> **InputPlaceAppearance:** an ePNK Label with a string containing the name of theAappearance of the Place (only used if the Place is an Input Place, for more information see "Place Appearance" in 2.3).
>
> **Token:** an ePNK Label representing a token with a string indicating the name of the Appearance that will be associated to said token (for more information, see "Token Appearance" in 2.3). As in classic Petri nets, a Place may contain any number of Tokens.

AnimationLabel: an ePNK Label containing an Animation object (this will indicate the Animation or Animation Sequence to be executed on a Token that is inserted into the Place, whether this is through a Transition or through external means; for more information on Animations, see "Animations" in 2.3).

6.1.2 Animation classes

Here are described the classes that define the animations of the components of an ePNS Petri net for the later visualization.

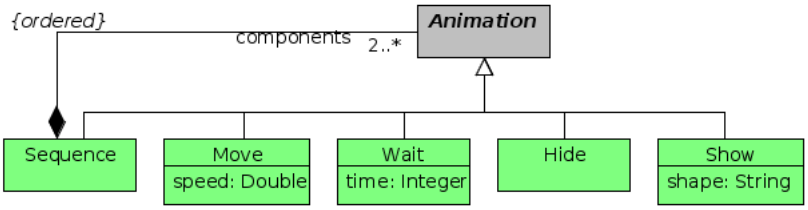

Figure 23: Animations Domain Model

Animation class

The Animation class is a definition of an animation, which will be indirectly associated to different elements of the Geometry of the Petri net and the Appearance of its components. An Animation in itself isn't anything, the classes described below are the ones that specify an Animation's behaviour.

Sequence class

The Sequence class is an Animation object containing:

components : a set of 2 or more Animation objects

This class is used to model an aggregation of Animations.

Move class

The Move class is an Animation object containing:

speed : Real

A Move object represents an animation of an object moving at a certain speed along a Track that represents the Place to which the Animation is associated.

Wait class

The Wait class is an Animation object containing:

time : Integer

A Wait object models an animation in which an object waits for a certain amount of time (e.g., a car waits in a crossroads for 3000 miliseconds before entering it).

Hide class

This object models an "animation" where any object currently associated to the Place to which the Animation is applied is hidden from the screen.

Show class

The Show class is an Animation object containing:

shape : String

This class represents the "animation" of showing an object (described by its shape) in the position of the Place associated to this Animation (if the Place is represented by a Track the object will be shown at the beginning of the Track). This could be used to, for example, put a traffic light into the simulation in the following way:

1. Two places are created, both with the same geometrical position. One of them has attached a show(green_light) and the other a show(red_light), where green_light and red_light are the appearances of a green traffic light and a red traffic light respectively.
2. When a Token moves into one of the Places, a traffic light (a green light in one of them and a red light in the other one) would be shown.
3. An Input Place or a Wait Animation could be used to change from a green to a red traffic light.

6.2 Geometry editor

Author: *Georgios*

The Geometry Editor is the component with which the technical user will draw the geometry on which the Petri net behavior will be simulated. This will be achieved by providing a Geometry Model that basically describes what a Geometry object is comprised of, or in other words, what the components that make up a Geometry object are.

At this point the reader should recall that the simulation is track-bound. Thus, through the Geometry Editor the technical user will draw the paths (straight or curved lines-tracks) along which 3D objects will move. Furthermore, the technical user will have the possibility to set control points, such as a signals, with which the animation of the 3D objects could be controlled. Other than that, the Geometry Editor will allow for naming different geometry objects, such as tracks and signal positions. This could be achieved by a label attribute associated with every Geometry object. It constitutes a unique identifier for such an object. In this way the label attribute plays the role of one-to-one link between Places in the Perti Net and the Geometry objects.

In particular, for visualising systems modeled as Petri nets in a track-bound manner, the Geometry model in Figure 24 has been created.

A detailed description of each and every class in the model will be presented in that point.

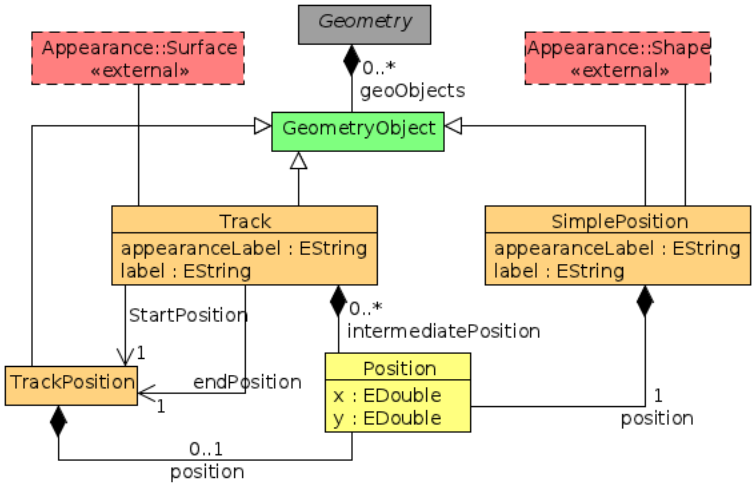

Figure 24: Geometry Domain Model

Geometry

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

GeometryObject

The GeometryObject class is an empty class from which different geometry objects, such as Track, TrackPosition and SimplePosition, inherit.

Track

A Track class consists of :

appearanceLabel : EString

label : EString

A Track object resembles a line in the geometry. This line can be drawn between two TrackPosition objects. A Track can be either a straight or a curved line. A Track label constitutes a reference to that particular Track object while the appearanceLabel defines the appearance of the Track on hte 3D visualization space.

TrackPosition

A TrackPosition is a GeometryObject that simply represents a point on a geometry. Two TrackPosition objects are needed for a Track to be created.

Position

A Position class contains :

x : EDouble

y : EDouble

A Position object resembles a point in the two dimension space in a geometry.

SimplePosition

A SimplePosition is a GeometryObject containing :

appearanceLabel : EString

label : EString

A SimplePosition object is correlated to one Position object. Its appearanceLabel defines the appearance of the SimplePosition object on the 3D visualaization space, while the label is a reference to it.

6.3 Appearance Editor

Author: ***Jerome***

The Appearance Editor is another component we will provide to the user, allowing him to associate a visual representation to certain elements of the Petri net so each element can have a representation in the Simulator. This Editor should be used by the technical user and will be displayed as a tree. Each element of the tree will have a Label to identify it and some special properties to customize it.

We can see in Figure 25, the domain model associated to that component.

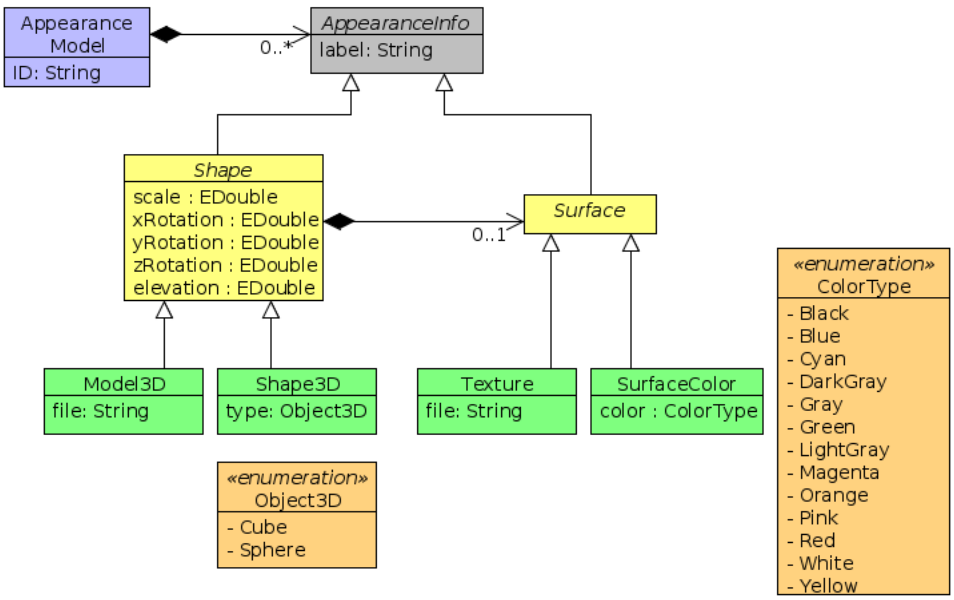

Figure 25: Appearance Domain Model

We will now describe in details each class of this domain model.

Appearance Model

The AppearanceModel class is simply an abstract definition of an appearance object.

AppearanceInfo

The AppearanceInfo class is an empty class from which different appearance objects, such as Shape or Surface, inherit.

Shape

A Shape class consists of :

scale : EDouble

xRotation : EDouble

yRotation : EDouble

zRotation : EDouble

elevation : EDouble

A Shape object that will be used for 3D elements. It has a lot of attributes such as scale that represents the scale the 3D object should be represented with, (x,y,z)Rotation that represent the rotation on each axe we have to apply for this object and elevation that represents the "high" the object has to be from the ground. This is also an abstract class and it can be inherited by the Model3D and Shape3D classes.

Model3D

A Model3D class consists of :

file : String

A Model3D object contains a single proper attribute : file that represents the place of the 3D model file on the hard disk. It inherits from Shape.

Shape3D

A Shape3D class consists of :

type : Object3D

A Shape3D object contains a single proper attribute : type that represents the type of 3D shape we want to load. These types are already configured internally and don't use any external 3D shape. This class inherits from Shape.

Object3D

An Object3D enumeration consists of :

Cube

Sphere

An Object3D is a pre-configured 3D shape that can be used for our software. If the user wants to have another 3D shape he needs to use the Model3D object.

Surface

A Surface class is simply an abstract definition of a 2D object. It can be inherited from the Texture and SurfaceColor classes.

Texture

A Texture class consists of :

file : String

A texture object contains a single proper attribute : file that represents the place of the Texture of the Surface on the hard disk. It inherits from Surface.

SurfaceColor

A SurfaceColor class consists of :

color : ColorType

A SurfaceColor object contains a single proper attribute : color that represents the color of the Surface. This color will be one color we have pre-loadedin our software. The list of all different colors is the enumeration ColorType. This class inherits from Surface.

ColorType

A ColorType enumeration consists of :

Black

Blue

Cyan

DarkGray

Gray

Green

LightGray

Magenta

Orange

Pink

Red

White

Yellow

A ColorType is a pre-configured color we can use for coloring a Surface.

6.4 Configuration Editor

Author: ***Ruxandra***

The Configuration Editor consists of only one class that links the Petri net, its Geometry and its Appearance. In order to link them, the Configurator class contains references to each of the previously mentioned classes. Also, the Configurator contains an attribute specifying the width of all the tracks in the scene. A simple diagram that describes how the configurator gathers all the information needed for the running simulation is shown in Figure 26.

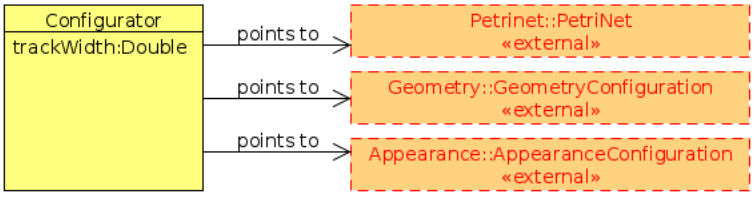

Figure 26: Configuration Dependencies

After editing all the files, linking them in the Configurator file and setting the track width, the simulation can be started by pressing "Start simulator" in the pop-up menu when right-clicking the configurator object.

6.5 Simulator

Author: ***Ruxandra, Marius***

The Simulator is the mediator between the RuntimePetriNet and the 3DEngine. At the initialization, it receives all the necessary data to start the simulation and the 3D visualization: the Petri net model, its geometry, its appearance and the animations associated. The simulator processes the information received, creates a RuntimePetriNet and initializes the 3D engine with the geometry, the appearance and the animations. It then waits for the user's input (press start button) to start the animations for the initial tokens. The way the 3D engine and the RuntimePetriNet work will be discussed in the next sections.

As it can be seen in figure 27, the simulator implements the Engine3DListener interface which means that it listens to the 3Dengine: it is notified when the user start the simulation, when an animation is finished, when the user clicks on a interactive input point or when the user stops the simulation. In each case, it takes the appropriate measures to produce the required response.

6.6 RuntimePetriNet

Author: ***Ruxandra***

The RuntimePetriNet is the component that knows where a Token is located in the Petri net at each step of the simulation. It can be noticed that the Token from the static Petri net is replaced in the

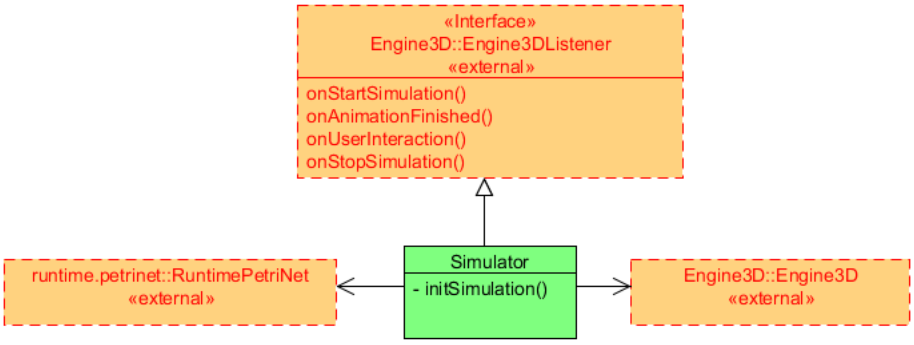

Figure 27: Simulator with listener

RuntimePetriNet by a RuntimeToken, while the Places and Transitions classes are the same. The reason for this decision is that the places and transitions are neither created nor deleted during the simulation(i.e. they are static components), unlike tokens, which are created and deleted at each execution of a transition. Therefore, tokens need to be small objects, containing just the relevant information, not all the irrelevant information (irrelevant for the actual simulation) inherited from EMF objects.

Figure 28 shows the internal structure of the RuntimePetriNet. It provides five public methods that can be called from other classes (in our case from the Simulator): init(), which initializes the RuntimePetriNet's structures and returns the movements(defined by a token and the place where the token is created) for the inital tokens; fireTransitions(), which iterates over all the transitions from the Petri net and executes all the ones that can be executed; fireTransition(), which executes only one transition and returns information about the new places of tokens after executing thay transition; markToken(), which is called whenever an animation finishes and a token needs to be marked as finished (in order to allow previous animations to finish); getInputPlaces() which returns all the geometry labels for the places that are defined as input places(see 6.1.1)

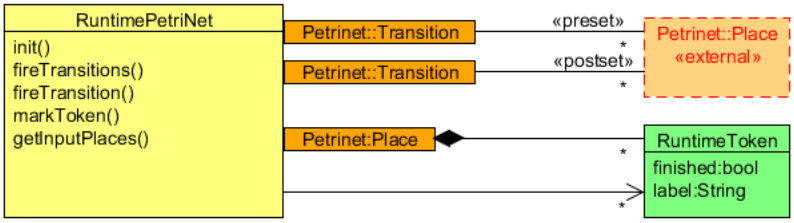

Figure 28: Runtime Petri net

Structurally, the RuntimePetriNet has a dictionary that associates a place with all the RuntimeTokens it contains, whether they are marked or not. It can be seen in figure 28 that a RuntimeToken has two main attributes : one is for checking if the corresponding animation has finished and one for indicating the way the RuntimeToken will look in the 3D visualization. The RuntimePetriNet

contains also two optimizations for rapidly retrieving the presets and postset of a transition : a dictionary that associates each transition with all the places in its preset and a dictionary that associates a transition with all the places in its postset.

6.7 3D Engine

Author: ***Cosmin***

The 3D Engine is the component of the application that handles the drawing and animation of the objects from the simulation, on the screen. It's main responsibilities include:

- the handling of the Window in which the graphical simulation takes place
- handling of the buttons through which the user can interact with the simulation (Play, Pause, Stop)
- drawing the representations for the objects in the graphical visualization
- maintaining an association between the 3D Objects that represent the elements from the Petri Net and the actual Petri net objects (Tokens, Places)
- updating the animations according to the configurations
- handling user input in the case of Input Places

In order to build a system that can be easily modified, the interaction between the Simulator and the 3D Engine is done by employing Interfaces and the Factory and Observer patterns. This interaction can be seen in the Figure 29.

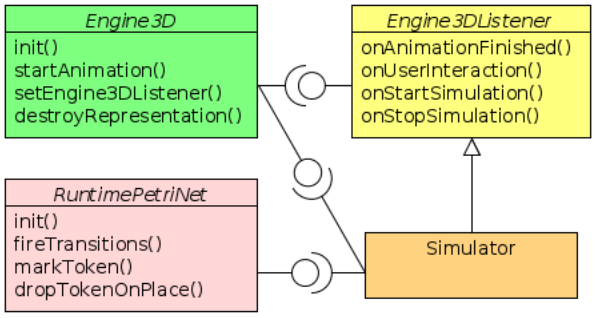

Figure 29: 3D Engine Interactions

The Simulator is the one that uses the Engine3DFactory to get a reference to an implementation of the Engine3D. Then, it initializes the Engine by providing it the configurations for the Geometry and the Appearance and registers itself as an Engine3DListener. The Engine3D keeps a reference to an Engine3DListener (in our implementation this is the Simulator) and for every event that can occur (e.g. an animation is finished, an user interaction has been detected, the animation was started or stopped) it notifies it. During a normal run, when an animation is finished or an user

interaction has been detected the Engine3D notifies the Simulator using the Listener's interface and, in return, the Simulator uses the Engine3D interface to start required animations.

Figure 30 shows a detailed view of **ePNS**'s implementation for the Engine3D Interface presented above. Java 3D was chosen as the supporting 3D framework. We will now shortly describe each of the components in the diagram and their interaction.

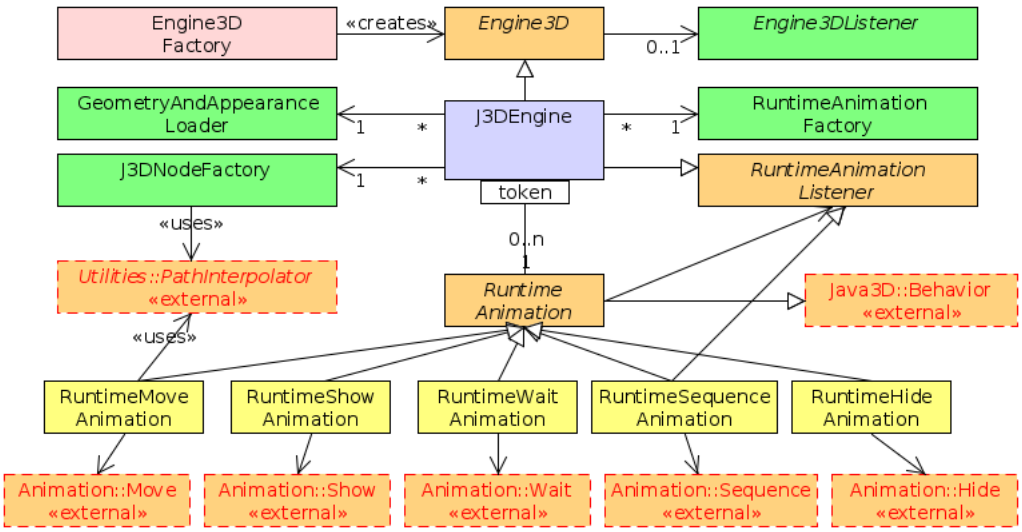

Figure 30: 3D Engine

J3DEngine - This class is the core implementation for our Engine3D and is handling the main logic for the graphical representation of the simulation. Besides this, it interacts with the Simulator, either by receiving calls (through the methods defined in the Engine3D interface) or by sending notifications to it (through the Engine3DListener interface). Also, this class takes care of initializing the Java 3D environment and the Window and handles user interaction.

Engine3DListener - As mentioned before, this is used to allow a proper mechanism for the Engine3D to notify interested components of events regarding the simulation. As it can be seen in Figure 29, such a listener has to implement 4 methods which allows it to be notified when the user starts the simulation, when an animation is finished, when the user clicks on a interactive input place or when the user stops the simulation. Using it allows the 3D Engine to work independently of other components. For example, the J3DEngine could also be used by something else besides the Simulator, for the purpose of animating objects.

GeometryAndAppearanceLoader - This helper class is initialized by the J3DEngine with the Geometry and Appearance configurations. It loads the necessary data (e.g. textures, models etc.) and, when required by the engine, serves geometry or appearance information.

J3DNodeFactory - The node factory is a Java3D dependent class, that handles the building of the various Java3D nodes and branches, for different representations of the objects during the visualisation. Basically, given the geometry or appearance label, it creates a new Java3D node (or

alters an existing one, if required) used for displaying objects according to the user's preferences. For example, these nodes are used to display Token representations, visualization for Geometry *Tracks* or *Simple Positions*. For the purpose of creating the tracks' representations, the *PathInterpolator* is used to compute where(position) / how(rotation) quads should be placed to obtain a proper display result.

RuntimeAnimation - This abstract class is used by the J3DEngine to manage the animations that are running. It holds all the required information (e.g. the RuntimeToken it is animating, references to Java3D Nodes in the scene graph etc.). It is built as a unit that completely self-manages its life-cycle: after it is initialized and started, all the required actions / updates are performed automatically by it and the finalization actions & notifications are executed accordingly, without any other operations performed by other component (e.g. J3DEngine, Simulator). In order to properly implement this, the Java3D **Behaviors** are used, the RuntimeAnimation abstract class actually being an extension of the *Behavior* class. What it does it that it attaches itself to the Java3D scene graph and, based on the implementing classes (one for each Animation - see below), it sets up wakeup conditions in order to execute code exactly when needed (e.g. at every X frames, after a time interval has passed etc.)

RuntimeAnimationListener - This interface allows the RuntimeAnimations to communicate with other classes interested in the progress of **RuntimeAnimation**s. To exemplify, for the simple Animations (Move, Wait, Show and Hide), the J3DEngine is used as a listener, so it is informed, for example, when the animations are finished. However, the Sequence Animation also implements this interface and, when executed, starts the 'child' animations by itself and registers itself as a listener for them. Thus, when one of the animations in the sequence is finished, the next one is started. The Sequence Animation also has to handle the proper forwarding of wakeup conditions between the child animations and the Java 3D Engine.

RuntimeMoveAnimation, RuntimeSequenceAnimation, RuntimeWaitAnimation, RuntimeShowAnimation, RuntimeHideAnimation - These 5 classes are implementations of the **RuntimeAnimation** interface presented above and each of them takes into consideration the particularities of the Animation type it refers to. They employ Java3D features to display the required animation or to have the desired effects. Considering that most of the control logic is implemented in the **RuntimeAnimation** class, these implementation mostly have to focus on the actions needed to be performed and the next moment when they should execute actions again (see Table 1).

Implementation	**Actions**	**Next Wakeup**
Move Animation	Move the token representations	Fixed framerate
Wait Animation	Just wait	Set up time interval
Show Animation	Show the object	Never
Hide Animation	Hide the object	Never
Sequence Animation	Initialize animations and forward wakeups	When child animations require

Table 1: Table showing Runtime Animations actions and wakeups

RuntimeAnimationFactory - This class is used as a Factory class in order to initialize the implementation for the **RuntimeAnimation** corresponding to a given **Animation**.

6.8 Path Interpolators

Author: ***Anders***

In order to make paths in the geometry editor and their counterparts in the simulator look the same, a utility component has been developed. As it is an architectural choice, that the simulator should be open for new implementations of the 3D engine, it was crucial that the path interpolator component should be independent of both GMF, EMF, and of course the chosen 3D engine. Also, not knowing the choice of data types in alternative 3D engines, it was chosen use `double` as the base type for coordinates, Carthesian and polar. Another choice could have been `BigDecimal`, but knowing that even **Open GL** only uses float, and that BigDecimal is inefficient because of its need for garbage collection, `double` was chosen as a fair compromise.

The path interpolators shall support the geometry editor with a list of points, that can easily be converted to a `PointList` without actually knowing that class. They also shall support 3D engines, with exact position and orientation of an object at any distance from the start of a path. The two representations shall reflect each other, even though the geometry editor does not need orientations.

Consequently two utility classes have been created: A vector (`Vector2D`) class that holds coordinates and methods for calculating with vectors, and another class holding a vector for position and a vector for orientation. The name of the latter class is `Where`, a name that might be changed in a later version.

The facade classes and interfaces of the path interpolator component are:

`PathInterpolator` This is the interface for interpolators in this component. It defines methods for getting the length of the underlying path, finding the position and orientation at any distance from the path start. These methods are ment for the 3D engine to calculate where objects should be and what way they should point. I also defines a method for getting intermediate point, a pixel apart, for the geometry editor.

`Vector2D` This is the presentation of a 2D vector. It can be used to define a position or an orientation. It supports Carthesian and polar coordinates. It has been developed with efficiency in mind. Many methods aim to produce as few new objects as possible, when making calculations.

`Where` Is mainly a position and an orientation, implemented with two `Vector2D` fields. Several convenience methods are provided.

The concrete classes that implements the `PathInterpolator` interface are:

`LinearPathInterpolator` This interpolator works with straight lines, and is mainly provided as the first prototype, and for testing purposes. It has also worked as a proof of concept to the `PathInterpolator` interface and the other facade classes.

`QuadraticBezierPathInterpolator` This interpolator works with quadratic Bézier curves, quadratic Bézier curves works with two end points and an intermediate point defining the tangents to the two other points. This interpolator assumes that any other point is a tangent target, all other points are intermediate start/end points. In the case with an even number of points the

last line will be straight. The code for this interpolator is **highly** inspired from the code in **ePNK** and uses a recursive approach.

`BezierPathInterpolator` This interpolator uses any number of intermediate points and uses Bernstein polinominals to calculate actual positions, see 6.8.1 Bézier Curves.

`BezierCurvePathInterpolator` This is a cleaned up version of `BezierPathInterpolator`. It also provides extra points showing the tangents of the end points for the geometry editor.

The component includes additional helper classes, that can be used for other other purposes, not only in the scope of Petri net simulations and Bézier curves. The most important are:

`BezierCurve` The `BezierCurve` class is the foundation of the `BezierCurvePathInterpolator`. It incapsulates a list of `Vector2D` points and calculates the value for any t $0 \leq tt \leq 1$ of the Bézier function. It also finds the distance between two such values, and the `Where` of a distance from start $t = 0$.

`ToolBox` The tool box is a class with static methods as the `java.lang.Math` class. It has methods for calculating:

- the power of a real (`double`) number to an integer exponent. A recursive algorith is used here to increase efficiency.
- the binominal coefficient of n over k. This is used by ...
- the value of a Bernstein base polynonimal.

6.8.1 Bézier Curves

Author: ***Anders***

The most commonly known Bézier curves are quadratic and cubic Bézier curves. Quadratic Bézier curves uses three points, a start and an end point and one intermediate point. The intermediate point determines the outgoing angle of the curve from the starting point and the ingoing angle to the end point. The cubic Bézier curve has two intermediate points, each of them determining the outgoing angle of the start point and the ingoing angle of the end point, further more the distance of the intermediate points determine the detour of the curve from start to end. Even more intermediate points can determine the detour more and more precisely. The number of points besides the starting point, determines the order n of the curve. An order $n = 1$ is a straight line, $n = 2$ is a quadratic curve, and $n = 3$ is a cubic Bézier curve.

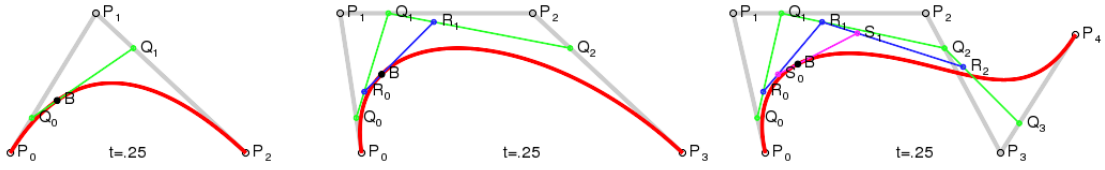

Figure 31: Bézier curves of order 2, 3, and 4

Figure 31 shows Bézier curves of order two, three, and four[4]. Most graphical programs doesn't use more than third order (cubic) Bézier curves, and it is not recommended to add too many intermediate points in the geometry editor, the curve will not be much nicer, but harder to calculate.

In the component we use:

$$B(t) = \sum_{i=0}^{n} b_{i,n}(t) P_i \tag{1}$$

wher $B(t)$ is the value of the Bézier function and $t \in [0, 1]$. $b_{i,n}$ is the i^{th} Bernstein polinomial of degree n. P_i is the i^{th} intermediate point. The Bernstein polinomial is defined as follows:

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{2}$$

The binomial coefficient is calculated using this formula:

$$\binom{n}{k} = \prod_{i=1}^{k} \frac{n-k+i}{i} \tag{3}$$

The implementations can be found in the `se2.e.utilities.ToolBox` class.

[4]The images has been borrowed from Wikipedia: http://en.wikipedia.org/wiki/Bézier_curve

7 Glossary

Author: *Georgios*

7.1 Technical Glossary

Animation The dynamic behavior of objects on 3D Canvas that are associated with Places of the Petri Net model.

Appearance The look and feel of the objects visualized by the 3DEngine .

Appearance Editor A graphical editor provided EMF for configuring appearance.

Control GUI A simple Graphical User Interface with appropriate actions (play/pause/stop buttons) so as the end user to be able to interact with the 3DEngine and consequently with underlying Petri Net execution .

Geometry the track (3D line/curve) on which the 3D-visualization of the Petri Net simulation takes place. The geometry consists of predefined objects such as, semicircles, points and lines.

Geometry Editor A graphical editor provided by ePNK and GMF for designing the area on which the simulation will be reflected.

Identity The attribute of Arcs in the Petri Net model that defines the trajectory of the Tokens.

IgnoreAnimation The attribute of Arcs in the Petri Net model that allows the target transition to fire without the associated Place Animation having finished.

Input place a place in which tokens can be insterted externally, i.e. by clicking an interactive control point.

Interactive control point A point that allows the user to interact with the system (e.g. by a click)

Petri Net Editor A graphical editor provided by ePNK and GMF for designing the Petri Net of the system a user wants to visualize.

Physical object A graphical object that is used by the 3DEngine for visualizing its behaviour during the Petri Net simulation.

Simulator The software component that will provide the underlying Petri Net execution information to the 3DEngine so as to be able to visualize the Petri Net execution with 3D objects.

Shape The visual traits of a physical object being part of the simulated system visualization.

Token The Petri Net element that moves along Petri Net places through transitions.

7.2 Technology Terms

Ecore Tool editor - The EMF editor that provides all the necessary elements for realizing or drawing the model in question.

EMF - The Eclipse Modeling Framework. It auto-generates code that represents the corresponding model (usually described in UML).

EMF Validation Framework OCL Integration - Helps for expressing constraints on ambiguous graphical models such as a class diagram. Tightly used with UML users can define constraints on their models.

ePNK - The eclipse Petri Net Kernel, a modern equivalent of PNVis missing the visualization part though. Built following the model-based software engineering paradigm. ePNK Petri Net Types to be extended to provide new functionality.

GMF - The Eclipse Graphical Modeling Framework provides the appropriate infrastructure for developing graphical editors based on EMF.

Xtext - Xtext is a framework for development of programming languages and domain specific languages.

Index