DTU - Denmarks Technical University

Software Engineering 2 (02162)

System Specification - Group C

s131343 - Diego González
s113440 - Jacob Gjerstrup
s131342 - Miguel Gordo
s132440 - MIkkel Abrahamsen
s083162 - Niklas Joensen
s131686 - Simon Päusch

21. October 2013

CONTENTS

1 Introduction

1.1 Description - Miguel

When working with abstract mathematical models, we make an abstraction of reality to better understand, improve and perfect a given system. This is done at a price: Usually, using an abstract mathmatical model implies a high level abstraction that can make us lose the connection to what we were modeling in the first place.

The software system developed by Group C for the course 02162 "Software Engineering 2" at DTU [1], hereafter refered as "our software system" intends to lessen this disadvantage for people working with abstract mathematical models, specifically Petri Nets, by providing an interactive 3D visualization in which the end user can see the results of the simulation and interact with it in real time.

Though Petri nets can be used in many different application areas, the focus of this project is for companies working with transport or material flow systems. Thus, this software could help speed up the process of designing such systems by rapidly identifying problems within our visual 3D representation.

1.2 Notation for the document - Diego

This document must be read following these rules:

The document have several references to external documets or web pages. These references will appear with these notation: *Reference* [$number$].

We have also included a glossary with some useful key words that the reader may want to consult. These references will appear with this notation: *Glossary term* ($number$).

The interpretation of some key words must be done in the way the RFC 2119 [1] does. Here, a part of the document has been included:

The key words "**must**", "**must not**", "**required**", "**shall**", "**shall not**", "**should**", "**should not**", "**recommended**", "**may**", and "**optional**" that appear in bold must be interpreted as follows:

- **must** This word, or the terms "**required**" or "**shall**", mean that the definition is an absolute requirement of the specification.
- **must not** This phrase, or the phrase "**shall not**", mean that the definition is an absolute prohibition of the specification.
- **should** This word, or the adjective "**recommended**", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
- **should not** This phrase, or the phrase "**not recommended**" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable

or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

- **may** This word, or the adjective "**optional**", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option **must** be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option **must** be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

The authorship of the document is defined the following way:

- Each section or subsection has a reponsible. The author will be defined as $< Section'sname >$ $- < author >$. When someone is resposible for one section, he is also responsible for all its subsections (for example, Jacob is responsible for section 3.2 and its subsections). This will hold UNLESS there's a specific author for one subsection (for example, Diego is responsible of section 6.4 but Nikklas is responsible for subsections 6.4.1 and 6.4.2).

- If there are more than one responsible for a section (for example section 1.4 with authors Diego and Miguel) all of them are equally resposible for the content.

1.3 Introduction to Petri Nets - Miguel

"Petri nets are are a graphical notation for modelling all kinds of discrete dynamic systems. Once a system is modelled as a Petri net, it can be simulated, analysed, and checked for correctness, which helps making sure that the system under development does what it is supposed to do"[2]. More formally, but without giving a mathematical definition, a Petri net is essentially a directed bipartite graph.

Petri Nets consist of three basic elements, *places*, *transitions* and *arcs*:

- *Arcs* connect *transitions* and *places*, but never run between two elements of the same type.

- A *place* is represented by a circle. Inside a *place* we can have *tokens*, which are represented by a black dot. A place can have any given number of *tokens*

- A *transition*, represented by a square, can *fire* when all the *places* from its incoming *arcs* have at least one token. Under these circumstances, we will say that the transition is *enabled.* Once it's *fired*, it consumes one token from each incoming *place* and puts one token in each of the outgoing *places. Firing* is an atomic action, i.e. a single, non-interruptible step.

In the next figure, we can see an example of an enabled transition - all the incoming places have at least one token inside.

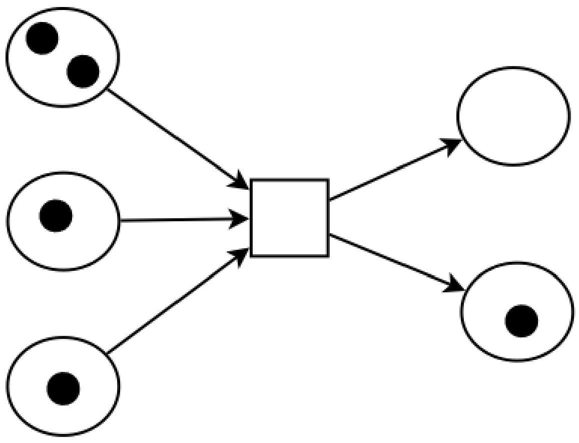


Figure 1.1: Before firing

After the firing takes place, one token will be consumed from each of the incoming places and a new one will be created in each of the outgoing places.

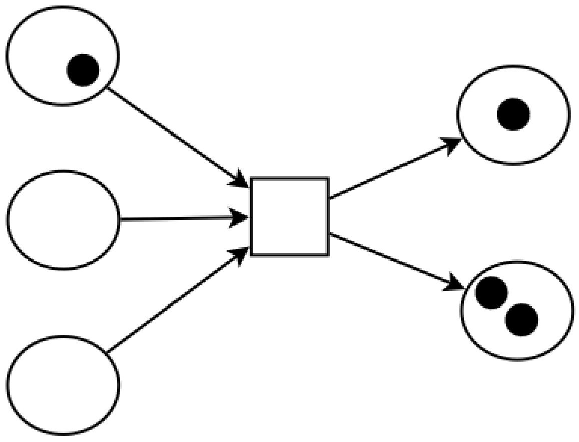


Figure 1.2: After firing

Unless an execution policy is defined, the execution of Petri nets is nondeterministic: when multiple transitions are enabled at the same time, any one of them may fire.

This sums up the basics of Petri nets. However for our software we will need to work with our own type of Petri net, which is named *PNVis*. This type of Petri nets includes addtional information on the physical world to visualize and animate the Petri net simulation in 3D. This way, people can see what is "really" going on in a simulation and interact with it in real time without even realizing that there is a Petri net behind the scenes. We will come back to our PNVis and explain it in more detail in the next section.

1.4 PNVis - Diego and Miguel

As we mentioned before, in order to be able to render a 3D Visualization out of a Petri Net, we need to include additional information compared to a basic Petri net. This extension is what we mean by PNVis.

Let's start with the motivation. As we have defined before, a Petri Net describes a discrete mathematical system, but we want to represent a track-based system which is continuous. To solve this, we discretize the real world; in our case, the sections of our tracked-based system. Here you can see a typical idea of a train system:

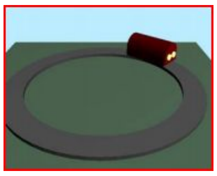


Figure 1.3: Graphical representation of a tracked based system

But, we would like to see this as two components, first, a Petri Net that models the behaviour of the system:

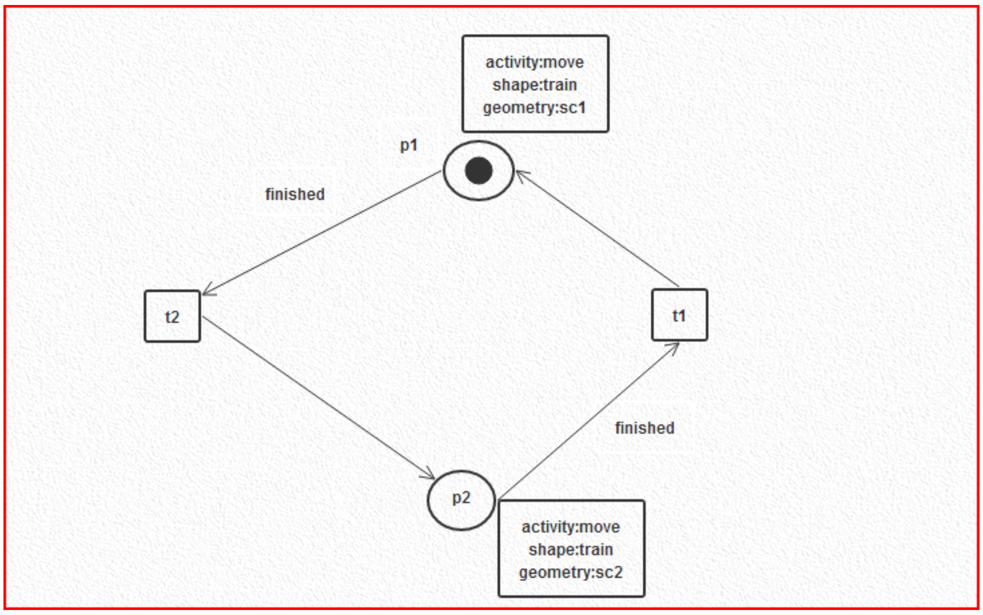


Figure 1.4: Abstraction of a tracked-based system

And now, a geometry that represents how this **should** look like:

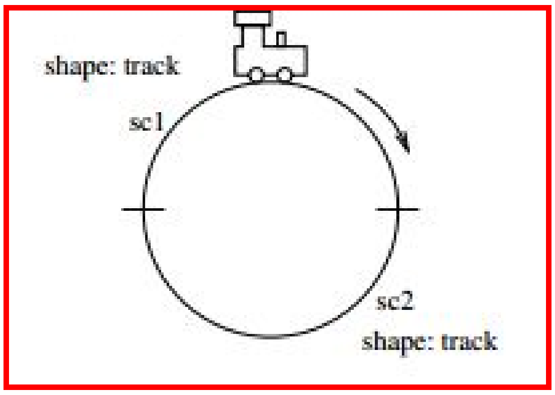


Figure 1.5: Geometry of the model

This abstraction allows us to easily go from the figures 1.4 and 1.5 to the figure 1.3. In figure 1.4 you can see that each place has a geometry label that has a corresponding piece of track in the figure 1.5 and a move animation. The token in the place represents the train (in this case, because the shape is "train") and the animation says move, which means that the train will move along the lines in the given geometry. What have we achieved?, We have split up the problem into three different parts: the geometry (figure 1.5), that represents all the information needed for simulation with lines and dots. The second part is the PNVis (figure 1.4), which is an extension of a simple Petri Net that allows us to define the behaviour of the components in a discrete way. The final part is a simulator that will use the previous two parts to effectively draw something that looks like the figure 1.3.

So far, we've added to the Petri Net some additional information. We've attached to the places the following information:

- *Geometry*: This makes a reference to the track of the geometry that corresponds to the place, and it is a name.
- *Activities*: This tag represents the animations that the tokens contained in that place **must** follow. It is a list of animations attached to the place. They can be as many as the user wants, and currently can be any combination of the following: *move(speed)* (in km/h), *appear(time)* and *trigger*. *Appear* can be seen as "wait *time* seconds", and *trigger* waits for a mouse click in its 3D model to finish.
- *Shape*: This makes a reference to the object to be drawn by the Petri Net simulator, and it is a name. In our example it is "train". The product will provide a way to use any (valid) shape provided by the end user. The simulator will load a set of files (or whatever) that represent a train and will use them to draw it in the screen.
- *tokens*: represents the number of tokens contained in a particular place. Despite the fact that in a Petri Net a Place **may** contain any number of tokens, the tokens in the PNVis represents the number of initials objects in that Place and as it makes no sense to have more than one object in the same place (physical place), in the PNVis this property

is a boolean, a Place can contain either one or no token. In our example, we've drawn a single token attached to the place p1.[1]

Additionally the outgoing (from the places) arcs **shall** have also information attached to them:

- Tag *finished*: This tag can be set to true or false, indicating if the following transition needs to wait for the animations in the place to finish in order to fire or not. In our example, both outgoing arcs have this tag.

In the next example we will introduce a more complex model which features a traffic light and uses all the attributes explained above. So let's start with the example:

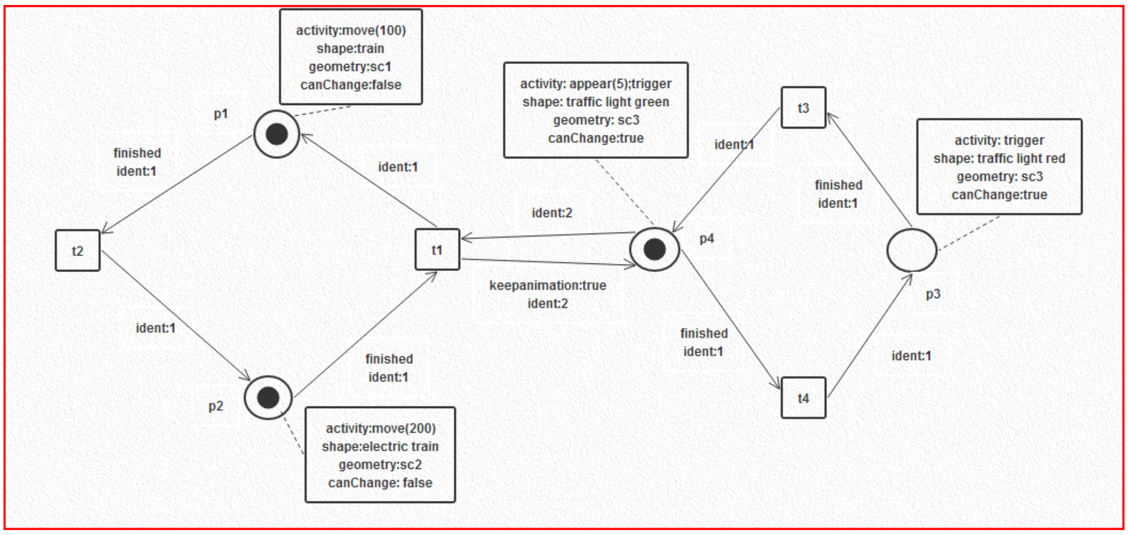


Figure 1.6: Example with the full version of a PNVis

And the corresponding geometry;

[1]Remember that the PNVis represents the initial marking of the Places, during the simulation, there will be possible to have more than one token in the same Place but not in the beginning (we don't want two things in the same physical place but they can be along the same track)

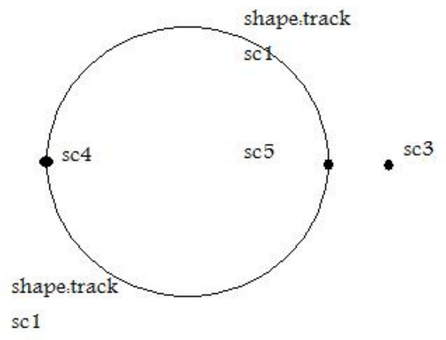


Figure 1.7: Geometry corresponding to PNVis of figure 1.6

As we can deduce from the names in the figure 1.6 this is only the same circle but we've added a traffic light at the end of the track sc1 (the place p1).

From this example one can understand the full version of the PNVis. As we can see in figure 1.6 the additional information attached to the places are:

- *Geometry*: This makes a reference to the track of the geometry that corresponds to the place, and it is a name. As we can see, both places p3 and p4 have the same geometry associated but they **should** only have one token at the same time so only one object is drawn. This makes the effect that the light is changing its color.

- *Activities*: This tag represents the animation that the token contained in that place **must** follow. In this case, we have attached to the places p1 and p2 move animations but with a different argument. This argument represents the speed, which means that a token in p2 will move twice as faster as a token in p1. In addition, in p4 there are two animations. This means that a token **must** finish first the appear for 5 seconds and then he have to wait for a trigger. When the trigger is finished then the token is ready for fire a transition (with the tag *finished*). Any combination of them (move, appear, and trigger) with the correct syntax will pass the validation test, however there are some **combinations which you should avoid**:
  - You **should not** put an *appear* animation after a *move*. This is equivalent to a *move* and an *appear* in the next Place.
  - You **shall not** put a *move* animation if the referenced geometry is a *Point*.
  - You **should not** put more than one *move* animation in one place
  - *Trigger* and *appear* **should** always take place before a move animation

- *Shape*: This makes a reference to the object to be drawn by the Petri Net simulator, and it is a name. In our example it is a train. The product will provide a way to use any (valid) shape provided by the end user. The simulator will load a set of files (or whatever) that represent a train and will use them to draw it in the screen.

- *tokens*: represents the number of tokens contained in a particular place. Despite the fact that in a Petri Net a Place **may** contain any number of tokens, the tokens in the PNVis represents the number of initials objects in that Place and as it makes no sense to have more than one object in the same place (physical place), in the PNVis this property is a boolean, a Place can contain either one or no token. In our example, we've drawn a token attached to the place p1, another one attached to the Place p2 and another one attached to the Place p4, which represents that the traffic light is green (because the Transition t1 will be enabled when a object finishes its animation in Place p2.

- *canChange* is a boolean, and it indicates whether or not the shape of an incoming *IItem* should be forcefully changed when entering the place. This gives us more flexibility to create more complex track-based system.

This is only a simple example, and in fact, we would like to have more animations than simply "move" and we would also like to give a speed to the "move" animation. Other animations could, for instance, be a trigger, to wait in the end of the track for the user to click on the train.

This additional information is mainly attached to the places. This means that each place **shall** have a series of text atributtes and labels, each containing information which will be relevant for the Engine3D. These are: *Geometry*, *Activities*, *Shape*, *canChange* and *tokens*.

Additionally the arcs **shall** have also information attached to them: *finished* tag, and *ident* tag and *keepAnimation* tag.

- Tag *finished*: This tag can be set to true or false (in figure 1.6 is drawn when it's true), indicating if the following transition needs to wait for the animations in the place to finish in order to fire or not.

- Tag *ident*: This tag will ensure that the identity of the tokens will be conserved when they are consumed in a transition. The basic Petri Net destroys the tokens of the incomming arcs (one token per place) and creates a new one in the places of each outgoing arc. In the PNVis we would like to move the tokens to one place to another, to preserve its identity. To do this, each transition **shall** have the same number of incoming and outgoing arcs and the arcs **shall** have an *ident* tag so that each incoming arc corresponds to exactly one outgoing arc (from the point of view of a transition).

- Tag *keepAnimation*: This tag can be set to true or false (in figure 1.6 is drawn when it's true), indicating wether or not a transition, if it's going to fire, needs to change the current animation of an on-screen object or can keep the current one. This is only possible if the token returns to the original place so this is a constraint of the PNVis: If an incomming arc (from a transition) has the tag *keepAnimation* set to true the corresponding outgoing arc (with the same ident) **must** return to the same place.

In addition, every element of the PNVis (arcs, places and transitions) **shall** have an unique name that identifies it among the other elements of the Petri Net. In the figure 1.6 we've only included the names of the places and the transitions but also the arcs **shall** have an unique name.

2 Overall Description - Jacob

This project seeks convert a petri net model of a track-based system into a 3D visual representation. To make a proof-of-concept, the first track-based system to be converted will be a train-based system, but ultimately, any track based system should be able to be added to this system.

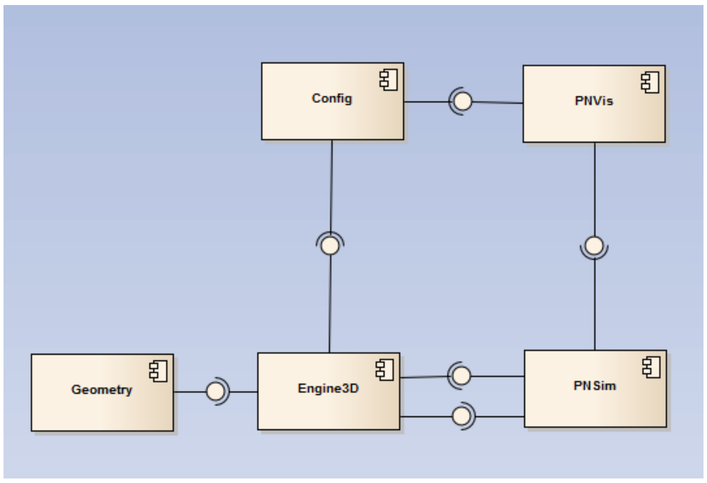


Figure 2.1: An overview of the 4 parts and how they are connected.

The project consists of four parts: A Petri net simulator, a geometry and editor, and 3D engine that draws the contents of the geometry, and a config editor that sets up the other parts. Each of these parts will be further explained in the sections to come - section 3 will list each of their features, as well as an example of how they can be used, whereas section 6 will discuss the architecture of each component and section 5 will explain the user interface of each part.
Figure 2.1 is a component diagram that shows how each of the four parts are connected. In this diagram, the Geometry refers to both the editor and the model, and the same is true for the config file. The Engine3D refers to itself, and the Petri net Simulator refers to the PNSim and all the components connected to this (i.e. the ePNK, PNVis and the Animation). As the diagram also shows, the Config is also connected to both the Geometry and the PNVis, having a reference to each of these.

3 System Features

3.1 Petri Net Editor - Diego

3.1.1 Introduction

The Petri Net Editor is the part of the product that allows the user to create, edit and save a PNVis (2) model via a graphical editor. We explain the details of PNVis in section 1.4. In the following section we will explain the feature list of the Petri Net editor that, in the end, will allow us to create a Petri Net with the additional information included in the PNVis. It is a (non-functional) requirement of this project that the tool **shall** be developed as a set of Eclipse plug-ins and that the extension of the Petri Net (the PNVis) **shall** be plugged in a tool called ePNK (you can find a brief explanation on the ePNK in section 6.3 and for a complete description you can check the reference [3]). Thus, it makes sense to define the PNvis with a domain model that follow the rules of the ePNK:

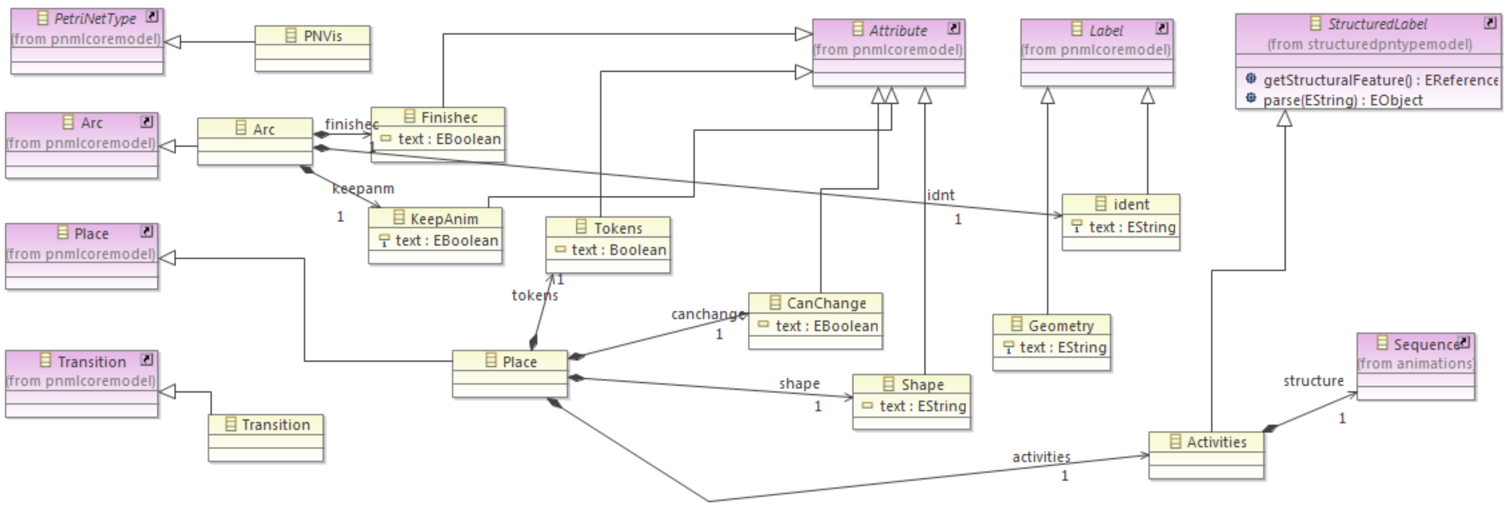


Figure 3.1: domain model for the PNVis

The editor is given for free with this domain model by the ePNK. The feature list of this editor is described in the following section and it includes all the features that the ePNK provides. It is important to note that in the definition of the PNVis there are some additional information attached to the Places and Arcs. Now, in the ePNK there are two different ways to attach this information to the Places and Arcs. You can either put it as a label that is shown graphically in the graphical editor or as an attribute that is shown in a view of the Eclipse workbench (in the properties view). We decided to put the information of the geometry and animations (of the Places) and the ident (of the arcs) as labels and the rest of the information as attributes. Moreover, in the definition of the PNVis a Place **shall** have attached a list of valid animations (move, trigger or appear). The ePNK has a special type of label called Structured-Label that allows the user to define a particular syntax for these labels by giving a function that parses the content of the label. The particular syntax in our case is described in section 6.4.2 but this is only our implementation and another one **may** be valid.

3.1.2 Feature list

- The Petri Net Editor **shall** allow the user create a new pnml document (3) in a new project within the Eclipse framework (4). This pnml document **shall** have the specific caracteristics of a PNVis.
- The Petri Net editor **shall** provide a tree editor where a new PNVis can be created and manipulated.[2]
- The Petri Net editor **shall** provide a graphical editor to edit the PNVis. This will be done via a "drag and drop" interface, and all the changes made here **shall** correspond to changes in the tree editor and vice versa.
- Both the tree editor and the graphical editor **shall** provide the following functionality:
  - They **shall** provide the functionality to add new places, transitions and labels.
  - They **shall** provide the functionality to connect places and transitions via arcs [3].
  - They **shall** provide the functionality to connect the labels with the places, arc and transitions in the correct way, i.e. the user **shall** be able to attach an animation label and a geometry label to a place. The user **shall** also be able to attach a finished label, a keepannimation and a ident label to an arc. [4]
  - They **shall** provide the user with a way to ensure that an animation labels has the correct syntax[5].
  - They **shall** allow the user to modify the attributes of the places, arcs and transitions via a view in the workbench (the propeties view).
  - They **shall** allow the user to roll back an unwanted operation, i.e. a previous operation. It also allows the reverse opperation, redo. Eclipse uses a stack so this can be done multiple times.
- The Petri net editor **shall** allow the user to store its Petri net in a pnml document.
- It **shall** provide the functionality to validate[6] a Petri Net so that the user will know if his PNVis is incorrect [7].

[2]Although this functionality is provided, the user **should** use the graphical editor instead of the tree editor as it is more intuitive and simple.

[3]It is not allowed to connect a place with a place nor transitions with transitions, the graphical editor will reject it.

[4]When we say animation label or ident label we refer to a label that contains the corresponding information of the PNVis as we've explained before

[5]As it is said in the definition of the PNVis, the places **shall** have a list of animations attached. The Petri Net editor **may** have a single label to represent this but from that label we **should** be able to extract a list of animations. The particular syntax in our case will be explained in section 6.4.2

[6]We've defined some rules that a PNVis **shall** follow. For example, each place **shall** have a geometry label attached to it and each element of the Petri Net **shall** have an unique name

[7]The word incorrect is used here because a Petri net validated with this method could not be valid for simulation. For example, there is no guarantee that the Petri net will fit with the given geometry and it does not verify that each incoming arc to a transition has its corresponding outgoing arc.

- It **shall** provide the functionality to view in the graphical editor if there is (or isn't) a token in each Place.
- It **shall** provide the functionality to fire the transitions accordingly with the rules of the basic Petri Net. This is because the fact that in the editor there's no simulation and the tags finished and keep animation makes no sense. In addition, as we allow a maximun of one token in each Place in the PNVis, when you fire a Transition and there's already a Token in a Place where it should appear a new one, that Token won't appaer. This is only to show the user how a Petri Net works (in this particular way).
- It **shall** provide the functionality to change the value of the attribute tokens of a Place by double clicking on it. If you haven't set a value of this attribute, it will appear a Token in that Place.
- It **shall** provide the functionality to draw the arcs in different colours depending on their identities (the tag *ident*).

3.2 Geometry Editor - Jacob

A geometry would be created that would hold the start- and end-point of any track. Furthermore, this geometry would hold the lines between the points, as well as these lines bend points (these bend points would be used for making curved lines). Each of these types would furthermore have a distinct look - points are represented by hollow squares, lines are represented by arrows (source → target) and bend points would be represented by filled squares.
To facillitate this, a graphical editor were implemented that
Below follows a list of the features that **shall, shall not, should, should not** and **may** be in the geometry, as well as a descripton of how this editor can be used.

3.2.1 Feature list

- The geometry **shall** be able to store the XYZ coordinates of each point, as well as lines connected to these.
- The geometry **shall** be able to be consistently saved and loaded, such that when a user shuts down the program and starts it again, the geometry remains unchanged.
- The geometry **shall** provide a graphical editor for the geometry. This graphical editor **shall** provide the following functionality:
  - The user **shall** be able to move the position of each point around.
  - The user **shall** be able to see how each line is connected.
  - The user **shall** be able to save the geometry once it has been changed.
- The graphical editor **should** allow the user to change the form of a given line - that is, turn a straight line into a curve or the other way around.
- The graphical editor **shall not** allow the user to draw lines that aren't straight or bezier curved tracks.

3.2.2 Usage

When using the geometry, the user will typically interact with the graphical editor. In this, the user will add points, as well as their connections and forms (straight line / semi-circle / Bezier curve). Once all connections has been placed, the user can then change the form of the tracks if it does not satisfy the wishes of the user.
Furthermore, the user can see where each point is placed in accordance to each track and **must** be able to move the point around the track after they have been placed.
Once the user is satisfied, the data **shall** be stored in an .xml file such that it can be loaded by the engine3D.

3.3 Configuration - Diego

3.3.1 Introduction

The configuration is a module that allows the end user to associate a PNVis (2) with a Geometry and run the application. How do we do this?, this module will allow us to create a new Configuration file (.config) where we can store the information (a reference) to a pnml documment (which **shall** contain a PNVis (2)) and a geometry file (which **shall** contain the geometry associated with the PNVis). Then, this document **shall** be shown as a tree editor and right clicking on the Config element the user **shall** be allowed to run the application (Ctrl+2 is a shortcut). To do so, the 3D Visualization needs also the 3D models of the objects displayed on the screen (which **shall** be obj files) and its textures (which **shall** be a single PNG file for each 3D object). This module will also allow the user to define the path where those files are. As it may be difficult to write a path manually, the application provides a button that will help the user to define those paths. The user **should** use this button.

3.3.2 Feature list

- It **shall** allow the user to create a configuration file in a particular project via the wizard of Eclipse.
- It **shall** provide the functionality to associate to the configuration to a given PNVis and a geometry.
- It **shall** provide a button (with the appearence of a train) to define the paths where the obj files, the shaders files (those ones will be provided by the developers) and the texture files are. The user could also define those elements by himself but he **should not** do so because the risk of failure is higher. The button **shall** be enabled only when the user has selected the Config element in the tree editor.
- It **shall** provide a pop-up menu from the configuration that allows the user to run the application. There is also a shortcut to run the application. The user just have to select the Config element in the tree editor and press Ctrl+2 (COMMAND+2 on MacOS X systems).

- It **shall** provide a validator to ensure consistency between the geometry file, the PNVis and the 3D models when the user tries to run the application. This is, that every 3D object or texture named either in the Petri Net and in the Geometry is present in the folders that the user had selected. If there is any problem, an error message will be displayed and the simulation won't be run.
- It **shall** provide the functionality to ensure that every transition has the same number of incoming and outgoing arcs and that every incoming arc has a corresponding outgoing arc (same tag *ident*).

3.4 3D Visualization - Mikkel and Simon

The 3D Visualization is the module of our software dealing with running the simulation of the PNVis and drawing it in 3D. This 3D Visualization will be interactive, meaning that the user can click on it and see the results of this interaction in the simulation in real time.

Depending on the chosen Petri net and geometry, the visualization **shall** render tracks and traffic lights and **should** render track switches. The trains **must** be rendered on the tracks and **shall** move according to the simulation.

3.4.1 Feature list

In the following the functionality of the user interface is listed:

- The user **shall** be able to interact with the simulation therefore the user inputs must be processed by the Engine3D and in case of valid input **shall** be executed.
  - For a more advanced usability non-valid input **may** be recognized and a message with a hint **may** be displayed.
- The user **shall** be able to create trains on a track, change the traffic lights from red to green and vice versa. Further, the user **shall** be able to interact with track switches, changing the track
- The user **should** be able to start and stop the simulation and **may** be able to pause and resume it.
- The user input **shall** be realised with shortcuts on the keyboard and by clicking with the mouse on the objects of the visualization e.g. traffic lights.
  - For a more advanced usability there **might** be a menu bar in the visualization window where the user **might** click on buttons to start or stop the simulation and to read some instructions on what are the possible and valid user inputs.

4 Non-Functional Requirements

The non-functional requirements of the project are requirements that are not directly related to the programming aspects of the development process. These relate instead to platform, system and documentation aspects.

4.1 Implementation- Mikkel

- The software **shall** be implemented as an Eclipse plugin.
- The programming language of the project **shall** be Java (JRE 6.0 or higher).
- For the 3D Visualizer, the OpenGL API (through LWJGL)[9] **shall** be used.
- EMF[4] **shall** be used for modeling and code generation purposes.
- GMF [8] components **shall** be used for making graphical editors allowing for changes in Petri nets models and geometry models.
- ePNK[3] **shall** be used for provide existing functionality for the Petri Net modules (the Editor and the simulator)

4.2 Documentation - Mikkel

- Code shall be commented to a reasonable extent; packages, classes and methods need to be commented in accordance to javadoc standards: `http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/index.html`.
- The following dates of delivery **shall** be met:
  - November 14th: First prototype, simple example.
  - November 21st: Draft version of handbook. The handbook shall be a guide to using the implemented software.
  - December 5th: Second prototype, feature complete.
  - December 20th: Final documentation and software version.
    - Javadoc comments.
    - Author tags.
    - Updated Systems Specification and handbook.
    - Test documentation.

4.3 Quality Assurance - Niklas

The system development follows the concept of prototyping. The objective of the first prototype is component interaction, which provides the basis to build all the functional requirements upon.
To assure the quality of the system throughout, a brief quality control process is implemented in the development of the system. The development team will take several precautions described in the subsections to ensure the system is developed to the best of our collective experience and with the aid of professor Ekkart Kindler in areas, which the development team might fall short. The quality assurance of each development step is elaborated further below.

4.3.1 Analysis, Design and Implementation

The project team conducted the system analysis in conjunction in the beginning of the project, in which we discussed in details our expectations, the system structure and how the system components should interact.

The delegation of component development was made by interest and experience. A team of two developers was assigned to each component. Throughout the project, familiarity with the project frameworks is acquired by following assignments and tutorials.

Quality assurance will be upheld by acquainting the different teams to the other components by redeploying the teams for a short amount of time. This has an added benefit of each team member will gain a thorough insight in the development of the other components, which means developing or modifying the components and component interaction will be more manageable.

The group will work in close cooperation with professor Ekkart Kindler throughout the project development by taking the feedback under consideration, received from the presentations and documentation deliverables.

4.3.2 Documentation

Throughout the project, we have a number of documentation deliverables, which are corrected by professor Ekkart Kindler. The remarks are taken into account and the documentation is modified accordingly.

Furthermore, on each deliverable and especially the last, all team members must thoroughly read the documentation and write remarks. Based on these remarks the group will discuss the contents of the documentation.

5 User Interface

5.1 Introduction

In this section we explain how the interface of our project will be. As we said before, to run a simulation, the user has to create a Petri Net, a Geometry and a Configuration file. Then, he has finished setting up this elements, he will be able to run the application.

The different features of each part are described in section 3 so we're not going to repeat them. To know how to run the project, we will provide a HandBook in a different document so we're not explaining here (deeply) how to use our tool, only how it will look like.

5.2 Petri Net Editor - Diego

This section describes the different elements that are involved in the creation of a Petri Net and how those elements are presented to the user.

To create a new PNVis (2) we **should** use the wizard provided by Eclipse to create a new pnml document. This document will contain the specific information of our PNVis (this information is described in section 1.4).

In the documentation of the ePNK [3] it is described how to actually create a new Petri Net of a Specific type (in our case, as we want to create a new PNVis, we have to select the element "http://OurPNVis").

The ePNK provides us a tree editor to see the information of the PNVis. After creating a new pnml document with a Petri Net and a Page the user **should** see something like this:

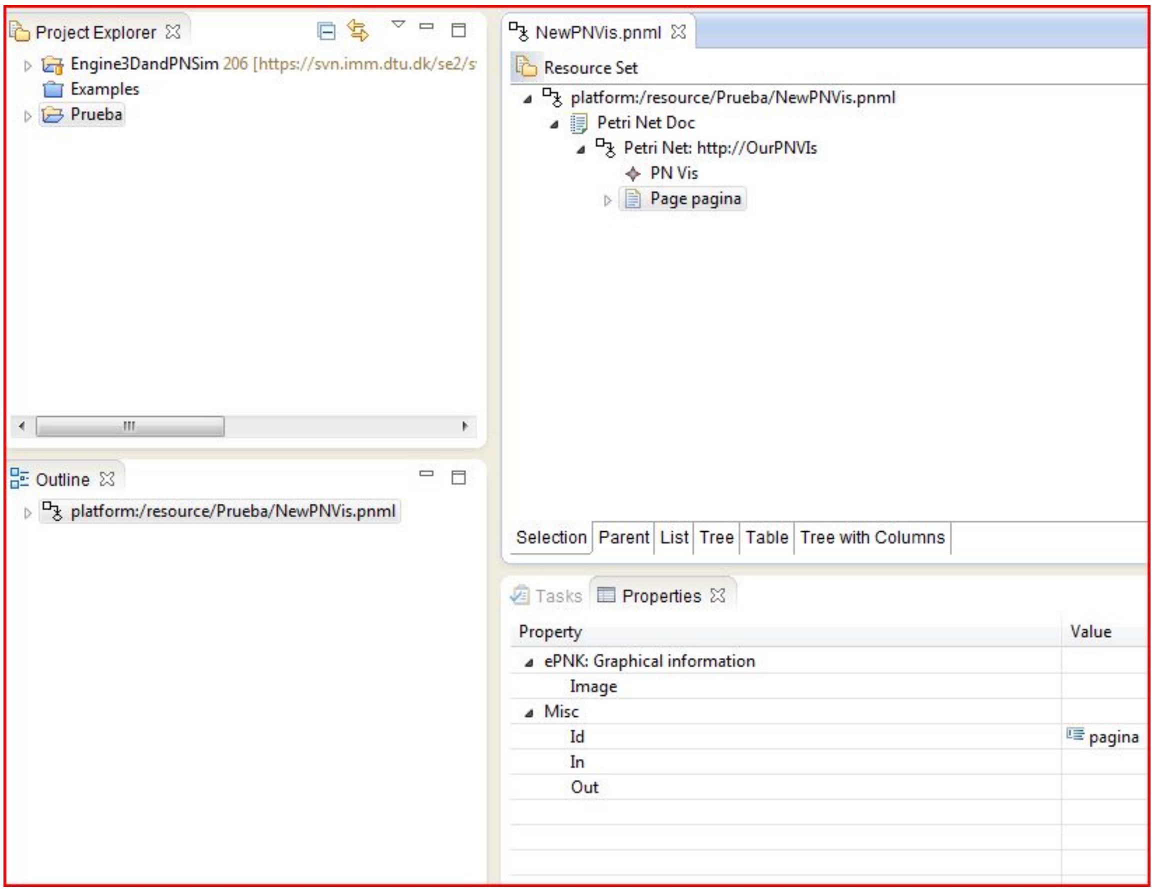
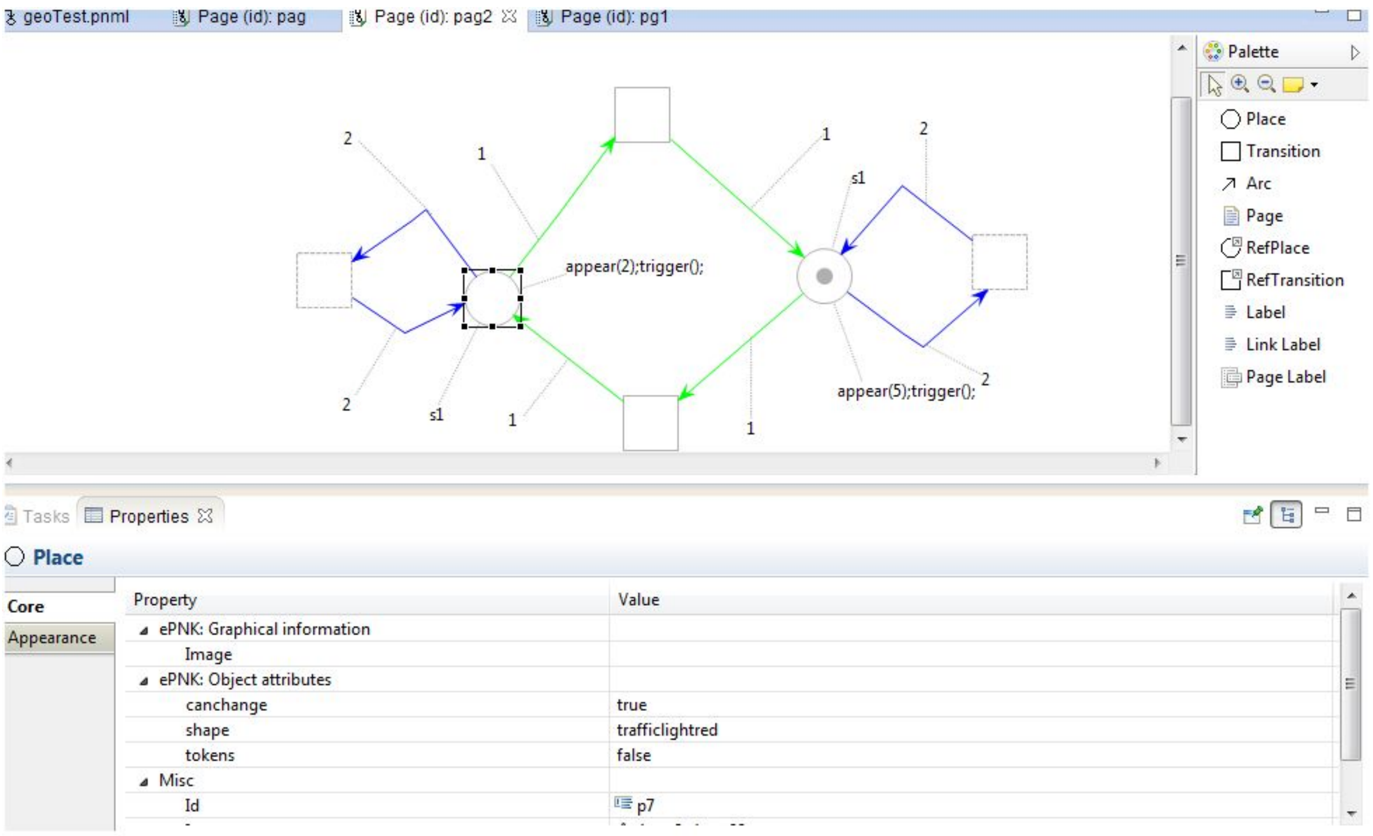


Figure 5.1: Tree editor for a pnml document

As it is better to define a Petri Net with a graphical editor, the ePNK also provides the functionality to view our PNVis in a graphical editor. This editor **shall** contain the specific information of the PNVis. Here we can see an example of the graphical interface to create an edit a PNVis:



Property	Value
ePNK: Graphical information	
Image	
ePNK: Object attributes	
canchange	true
shape	trafficlightred
tokens	false
Misc	
Id	p7

Figure 5.2: Graphical editor for the PNvis

This is a "drag and drop" interface, as we can see, at the right hand side the user can select the different elements of a PNVis (like Places, or Transitions) and he will be able to put them graphically in the editor. It is also remarkable that all the specific information of the PNVis is represents in this editor (and it can be edited and modified). For example, we can see that all the arcs have an ident tag (which is a label that we attach to the arc) and they are drawn in different colors accordingly to that identity.

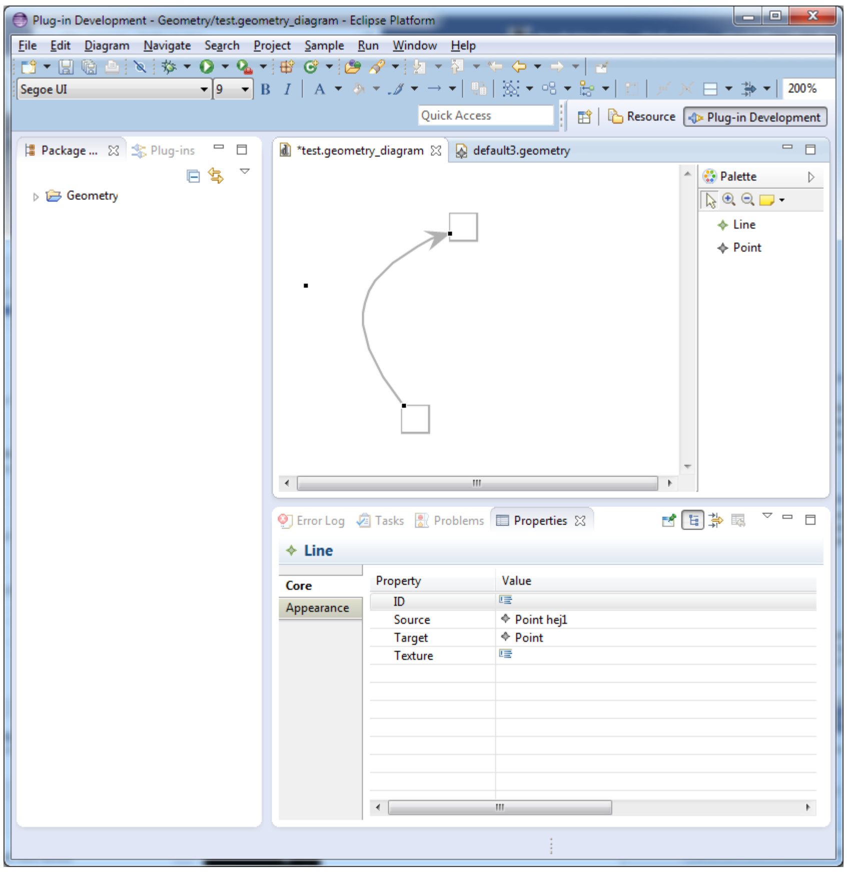
Some information is attached to the different elements as attributes, this means that it will be shown in the properties view instead of in the editor (as a label).

In this editor is where the user will be able to add or remove a token by double clicking on particular Places and to fire Transitions (if they are enabled) by right clicking on them and selecting "Fire Transition".

All the changes that the user makes on the graphical editor will be shown in the tree editor as well and they will also be stored in the pnml document.

There are more features that we inherit from the ePNK like the possibility to assign automatically the ids of the different elements of the PNVis but we're not explaining them. We rather refer to the ePNK manual [3].

5.3 Geometry Edtitor - Jacob



Property	Value
ID	
Source	Point hej1
Target	Point
Texture	

Figure 5.3: A small geometry

Figure 5.3 is a screenshot the graphical editor of the geometry, as well as a sample geometry with two points, one line and one bend point, all represented as described in 3.2.
To the right of the example, the "palette" can be seen, along with two entries - line and point. These are used to place new lines and points, and bendpoints can then be extracted by clicking on a line and dragging it away.
Finally, below the example of the geometry is the properties view, which shows various information about highlighted objects.

5.4 Configuration - Diego

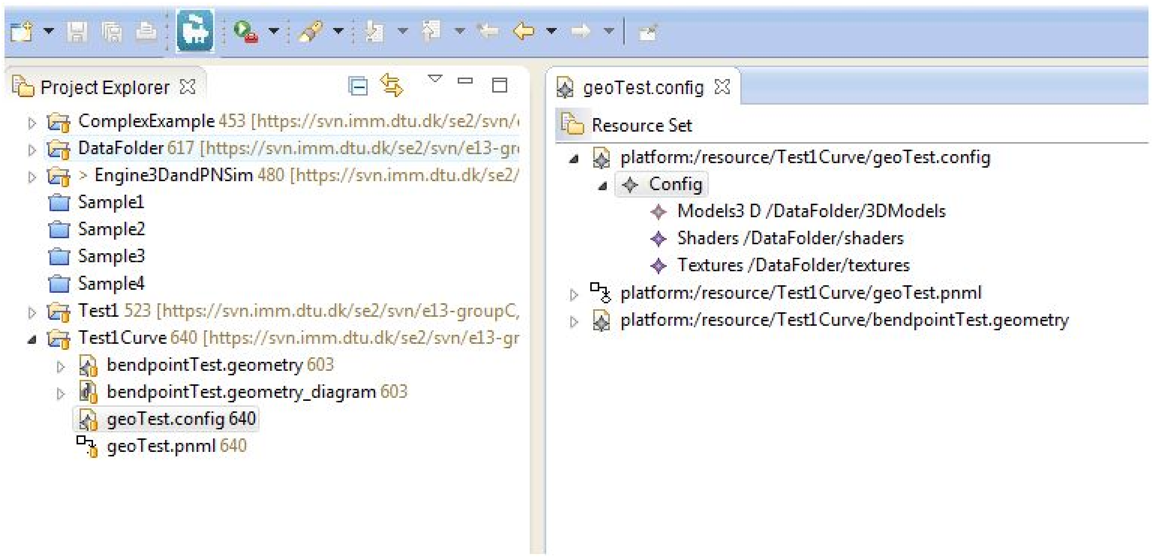
This section describes the appearance of the Configuration document and we introduce briefly the features that this editor has.

Again, we **should** use the wizard to create a new "Config Model" of the category "Example EMF Model Creation Wizards". Double clicking on it you **shall** see a tree editor. As we said, the user will be able to associate in this document a Petri Net (with a PNVis), a Geometry and some data (3d models, textures...).

To make the reference to the Petri Net Doc and the Geometry, the user will have to load that resources and selecting them in the properties of the Config element.

To select the path to the runtime information (3d models, shaders and textures), there is a button with the shape of a train. If we select the Config element in the tree editor that button will be enabled and clicking on it three popup menus will appear asking the user to select the different folders with the 3d models, shaders and textures.

The workspace **shall** look like this:



ComplexExample 453 [https://svn.imm.dtu.dk/se2/svn/
DataFolder 617 [https://svn.imm.dtu.dk/se2/svn/e13-gr
> Engine3DandPNSim 480 [https://svn.imm.dtu.dk/se2/
Sample1
Sample2
Sample3
Sample4
Test1 523 [https://svn.imm.dtu.dk/se2/svn/e13-groupC,
Test1Curve 640 [https://svn.imm.dtu.dk/se2/svn/e13-gr
bendpointTest.geometry 603
bendpointTest.geometry_diagram 603
geoTest.config 640
geoTest.pnml 640

platform:/resource/Test1Curve/geoTest.config
Config
Models3 D /DataFolder/3DModels
Shaders /DataFolder/shaders
Textures /DataFolder/textures
platform:/resource/Test1Curve/geoTest.pnml
platform:/resource/Test1Curve/bendpointTest.geometry

Figure 5.4: A glimpse of the tree editor of the Config file once we've loaded the Petri Net Doc, the Geometry and we've selected the correpponding folders

The other feature present on this editor is the way to actually run the simulation. the user can either select the Config element in the tree editor and right clicking on it select "Run Simulator -> Run" or he can use the shortcut Ctrl+2 (COMMAND+2 on MacOS).

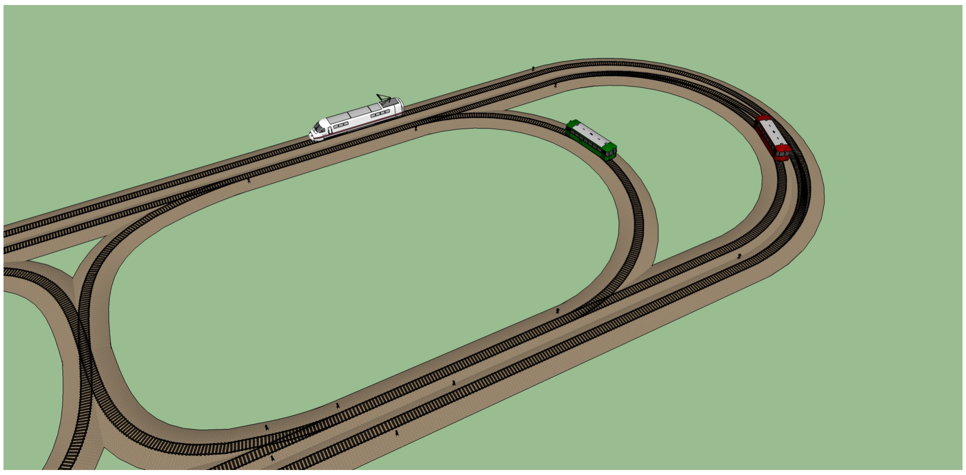


Figure 5.5: A mockup of the 3D visualization made using the program Sketchup and free models from Trimble 3D warehouse. These models were made by user Eneroth3 (Christina Eneroth)

The mockup shows the visual representation of the ongoing simulation, rendered in 3D graphics. While the mockup is an image of a stationary setup of non-interactive parts, the 3D visualizer of this project will be moving according to the input from the Petri net Simulator and also changing state depending on user input. Clicking on a traffic light will change the state of that traffic light to allow or disallow trains from moving past it. Clicking on a train track will attempt to create a new train at the beginning of that track piece. The visualization and simulation can also be paused and resumed using a keyboard key.

Since the point of the 3D visualization is conveying the state of the Petri net simulation, emphasis will be placed on using visually simple and colorful representation of the 3D scene's objects. The ability to rotate the virtual camera around the objects in the scene will also allow for a different perspective.

6 SOFTWARE ARCHITECTURE

6.1 OVERVIEW - MIGUEL

The following subsections will give an overview over the architecture of each module of the project. We will explain some of the specific details of each component and how they work together in a more technical detail

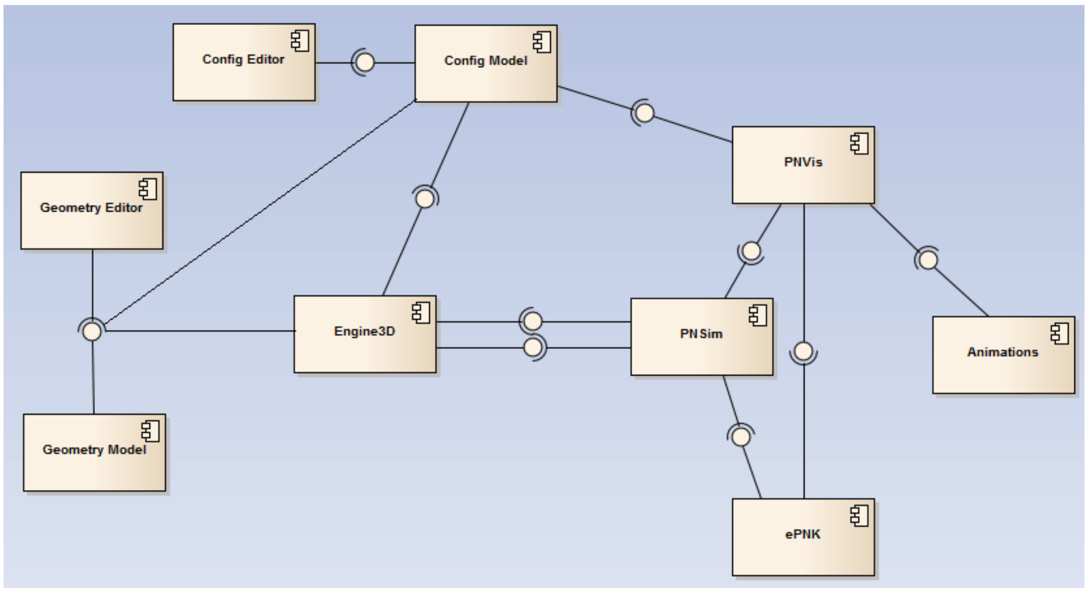


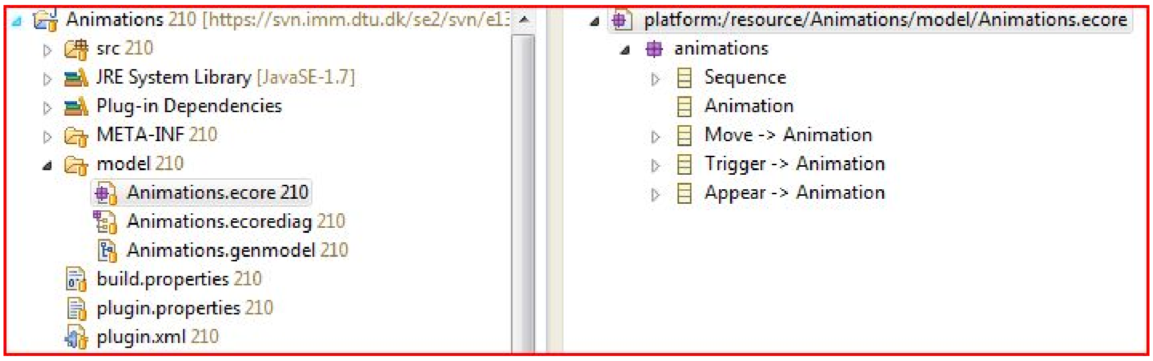
Figure 6.1: An overview of the 4 parts and how they are connected.

To develop this project, we make use of two powerful tools called EMF and ePNK. The first one allows us to generate code from UML class diagrams and the code to manipulate and store it. The second one allows us to define a new type of Petri Net (the PNVis explained in section 1.4). We're explaining these concepts in sections 6.2 and 6.3. Later on, we will explain the components of our product.

6.2 Introduction to the EMF - Diego

The Eclipse Modeling Framework is a powerful tool that allows us to generate code from UML models. In our case, several parts of this project are generated automatically from domain models and that is why I include here a brief introduction. For further details, please check the references EMF [4].

The basic information to generate code in EMF is an .ecore file which contains the information of class diagram. EMF provides a graphical tool to develop the diagram. The information to view the model is contained in a .ecorediag file. As an example, I'll show an example on how this works. Here we have the ecore document as seen in eclipse:


Animations 210 [https://svn.imm.dtu.dk/se2/svn/e1
src 210
JRE System Library [JavaSE-1.7]
Plug-in Dependencies
META-INF 210
model 210
Animations.ecore 210
Animations.ecorediag 210
Animations.genmodel 210
build.properties 210
plugin.properties 210
plugin.xml 210

platform:/resource/Animations/model/Animations.ecore
animations
Sequence
Animation
Move -> Animation
Trigger -> Animation
Appear -> Animation

Figure 6.2: Ecore model, as a tree

and the ecodiag diagrams of a sequence of animations:

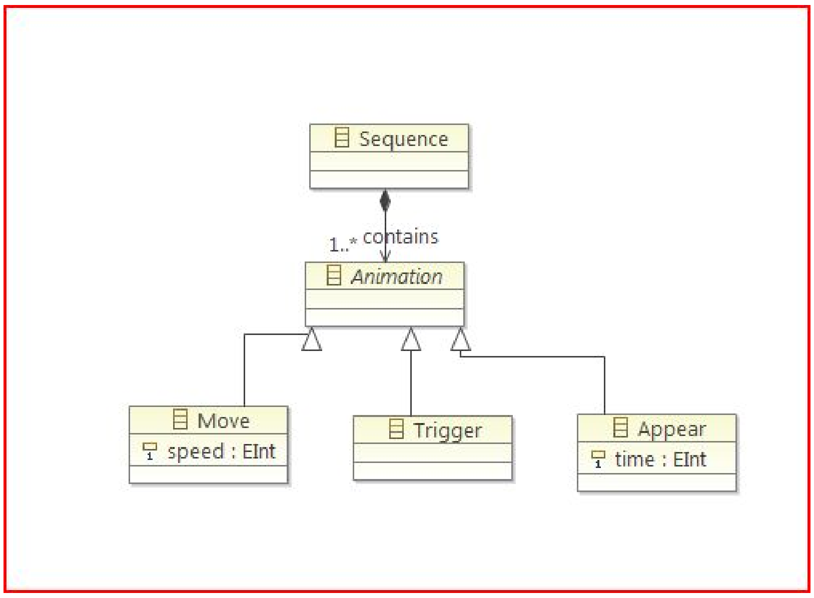


Figure 6.3: Ecore model diagram

From the ecore document, you can create a .genmodel file (which is shown in the left hand side of figure 6.2). That file is the one which generates the code of our projects. We use mainly functionality of EMF to create the Model Code (the code generated implements the domain model given in the .ecore file), the Edit code (to manipulate the objects with commands in a tree editor) and the Editor code (which makes contributions to Eclipse with menus, provides a wizard to create a document containing the objects etc).

The code generated in the Model Code implements the class diagram defined in the ecore file using the some of the Design Patterns [6]. It creates three packages (a root and two packages inside). In the root one we can find the interfaces of the classes defined in the ecore diagram. This is because EMF generates code following the pattern of Abstract Factory (see

[6]). The implementation is inside a subpackage called "impl". Here we can find the implementation of our classes. And there's a third subpackage called "util" that contains the code to serialize the objects in a xml document so that we can store a particular implementation of our class diagram.

The code generated in the .edit project contains the code to manipulate the objects defined in the class diagram as a tree editor (inside the Eclipse framework). It makes use of the pattern of Observer (see [6]), when we change anything in the tree editor, we want to see this changes in the actual model (and the other way around).

The code from the .editor project contains the code to plug-in this model in the Eclipse framework so that when we run this plug-in we will be able to create a xml document that contains the specific information of our class diagram (inside the wizard of Eclipse). It will also associate the a particular extension with your model to easily identify it.

6.3 Introduction to the ePNK - Diego

In order to represent a PNVis (the new type of Petri Net developed in this project, that contains the necessary information to simulate it, see section 1.4 for more details), we make use of a tool called ePNK [3] that allows us to easily develop a new type of Petri Net and plug it in the Eclipse framework to use it. What does ePNK offer?. It allows us to define a new type of Petri Net (like the PNVis) that contains more information than the basic Petri Net. In our case, as we've seen in section 1.4, we want to have some information attached to places and arcs to represent a tracked-based system. The way this is done is via an EMF model and our particular implementation is described in section 6.4. Once this is done, the ePNK allows us to create a new pnml document (3) that contains the information our particular Petri Net (in addition to the usual information of a normal Petri Net). Moreover, the ePNK provides a graphical editor to easily create our Petri Net via a "drag and drop" interface.

6.4 PNVis and graphical editor - Diego

It is supposed that the reader is acquainted with the ePNK. For a brief introduction you can take a look at section 6.3. The reader should also know the main ideas of the PNVis which are explained in section 1.4.

To plug in a new Petri Net type, we **should** follow the guidelines provided in the ePNK. This is the domain model of our new type of Petri Net, that includes the information requaired for the PNVis:

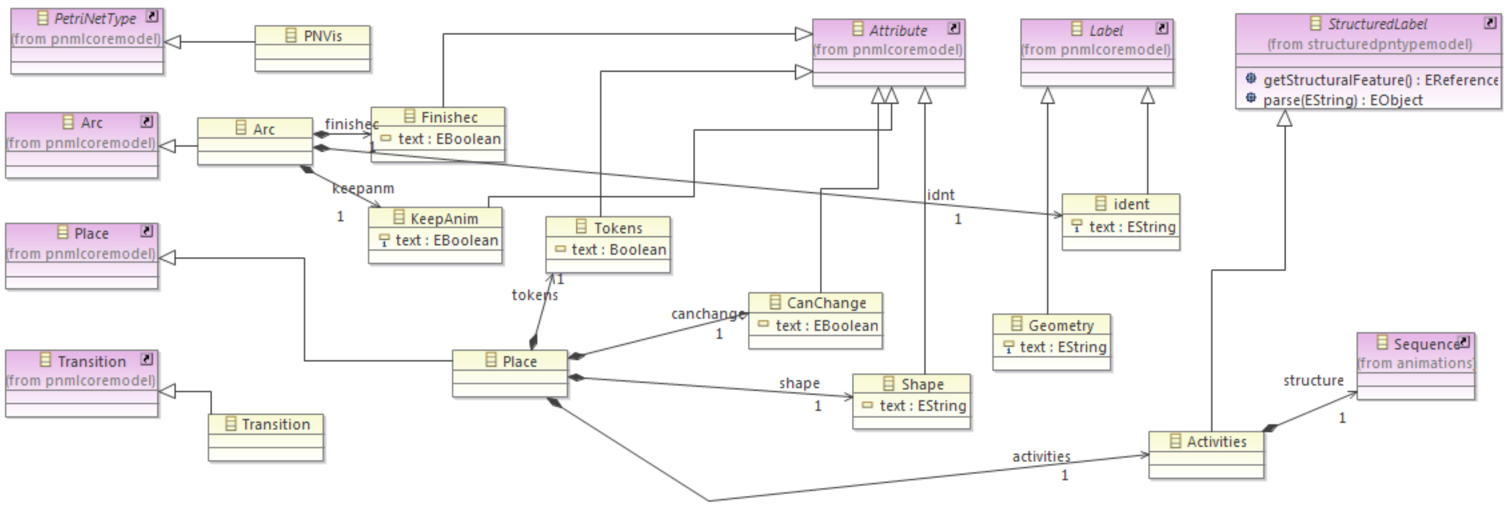


Figure 6.4: model for the PNVis

We've decided to move almost all the information to attributes, rather than labels, for not crowding the graphical editor with too much information. The only element that requires a separate section is the class Sequence of the plug-in *animations*. The concepts behind the classes of figure 6.4 are explained in section 1.4. For example, the Tokens attribute of the Places represents the initial marking of the objects. As we said in section 1.4, it makes no sense to have two (or more) object in the same (physical) place that's why we only allow to have one or no token per Place. The label *ident* of the ars represent the identity of the label as defined in section 1.4 and can be any string.

This is an auxiliary project that represents the animations we can have in a place. As we've defined, a Place **shall** have one or more animations. An easy way to represent this is by using the Sequence class of this plug-in, the structure of these sequence of animations is represented in this domain model (example in section 6.2):

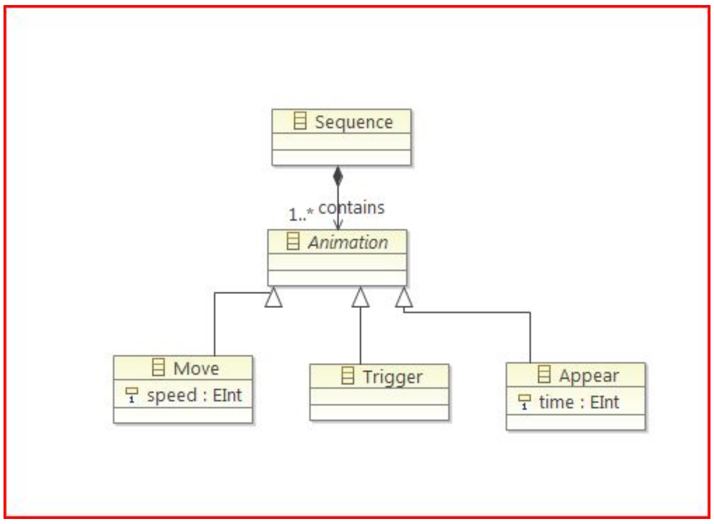


Figure 6.5: model for the Animations

As we can see, the Sequence contains at least one Animation. The code of these project has been generated with the EMF [4] and it consists of two plug-in projects: *Animations* and *Animations.edit*. These plug-ins are part of the final product. A brief description of EMF can be found in 6.2 of this document. We didn't include here the geometry because we thought that is more clear to attach that information to the Places. It's true that it is more general to have the geometry inside the animations but to represent a tracked-based system we thought that this was enough.

Now we can continue with the explanation of the PNVis. As we've already said, we **should** follow the rules defined in the ePNK manual to generate the code of our new Petri Net type. From the PNVis domain model, we use the EMF to generate the plug-ins: "PNVis", "PNVis.edit" and "PNVis.editor". The functionality that this plug-ins include can be found in the documentation of the EMF [4]. We made some manual changes to the generated code, because we want to use some of the functionality that the ePNK provides. To the "PNVis" project, we've made a new extension for the ePNK to know that the domain model we designed is in fact a new type of Petri Net that **must** be taken into account. In particular we're interested in storing our PNVis as a pnml document and also in the graphical editor to easily create and modify our PNVis. To make this features run the ePNK requires the projects "PNVis.edit" and "PNVis.editor".

The other change we've made to the code generated by EMF is adding a constraint that prevents the new Petri Net type (the PNVis) to have arcs that go from one place to another or from one transition to another.

As it is said in the ePNK, the Structured labels **shall** have a parser that creates, from a provided string, a set of objects that represents the information contained in the structured label. In our case, we use Xtext to make a parser that creates a "Sequence" with the animation contained in the label. The parser of this section and the syntax of these labels are explained in the next sections (6.4.1 and6.4.2).

We've also included another project called PNVisFeatures which contains all the additional features that this project contains. You can find there the impelmentation of the colors in the arcs, the graphical representation of the tokens in the Places, the command to fire an enabled transition[8] and the code to create and destroy a token of a particular place by double clicking.

6.4.1 Xtext Parser - Niklas

The purpose of the Xtext framework is developing a domain-specific language (DSL). In our case the parser will accept predefined keywords in the structured labels. Based on the keyword a corresponding animation object will be created.

6.4.2 Syntax for the animation labels - Niklas

The Xtext syntax allows three types of keywords as seen below. Each keyword when invoked creates a corresponding animation object. The structured labels can contain many animation keywords in a sequence.
- Move(int time);
- Trigger();
- Appear(Int speed);

[8]According to the basic rules of a Petri Net, as we said in the Feature list of the graphical editor of the PNVis, section 3.1

6.5 Geometry - Niklas

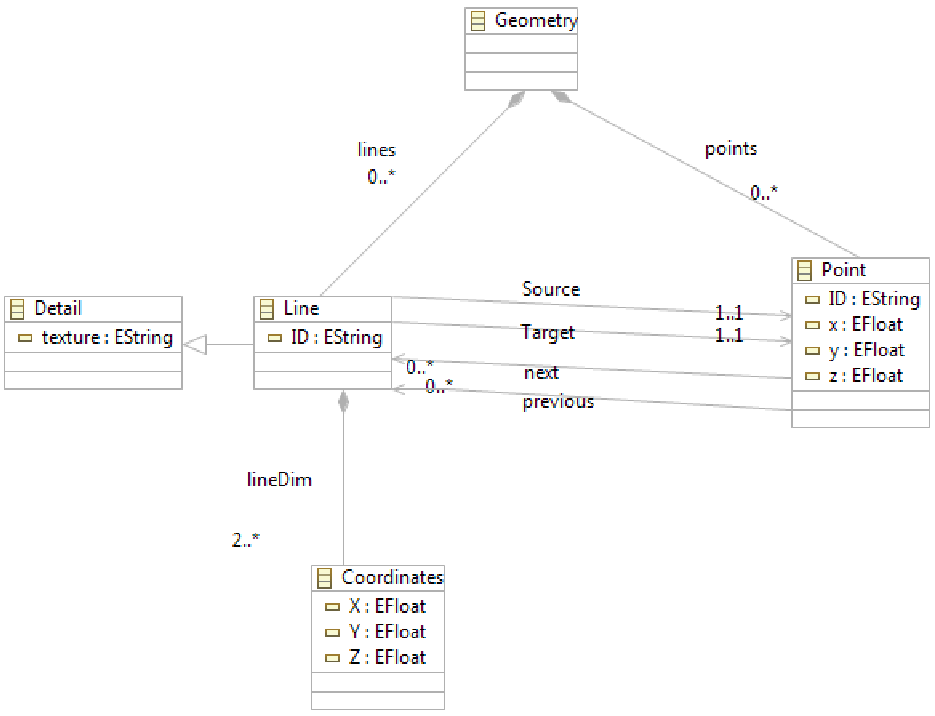


Figure 6.6: Ecore geometry model

The EMF meta model as illustrated in figure 6.6 shows the interrelations between the classes created using the EMF framework. The geometry contains the static data used by the 3D engine.

The *Geometry* contains four associated classes *Line*, *Point*, *Coordinates* and *Detail*. Each Line and Point class contain a unique attribute ID, which is used to identify a specific instance member and the data contained in the respective associated classes. All classes are elaborated further below

6.5.1 Line

The *Line* is associated with the *Point* class. For a line to be drawn it needs a source and target point. The 'next' and 'previous' association describes which point is the respective source and target.
The *Coordinates* class contains the dimension data used by the 3D visualization to draw a line. The drawn lines are used as the foundation for the train tracks by specifying the curvature and length.

The curvature of the lines are defined by a Quadratic Bezier. The Qadratic bezier curve is implemented in the geometry diagram.
A line has en an amount of coordinates which are calculated based on the point dimensions and the coordinate dimensions. In a quadratic bezier curve a line has 3 points in which the calculations are performed. Based on these points using the De Casteljaus algorithm, we populate the coordinates with interpolated points on the bezier curve.

The *Line* class is also associated with a class labelled *Detail* with an attribute *texture* that specifies which kind of track image to layer above the drawn lines.

6.5.2 Point

The *Point* class is used to set a fixed point in the 3D visualization. It can be utilized as a line endpoint or to contain visual attributes like a traffic light.

A noteable feature in the line

6.6 Configuration - Diego

6.6.1 Introduction

This part of the software connects the different components and provides a way to run the application via a popup menu. This module consists of five projects: "Config", "Config.edit", "Config.editor", "RunCommand" and "FillData". The three first projects are automatically generated by EMF [4] via a class diagram, the fourth one contains the extension of Eclipse of a simple command that will run the application and the last one will help the user to define the paths where the textures and 3D objects are stored.

6.6.2 Config

The domain model of the Config project is:

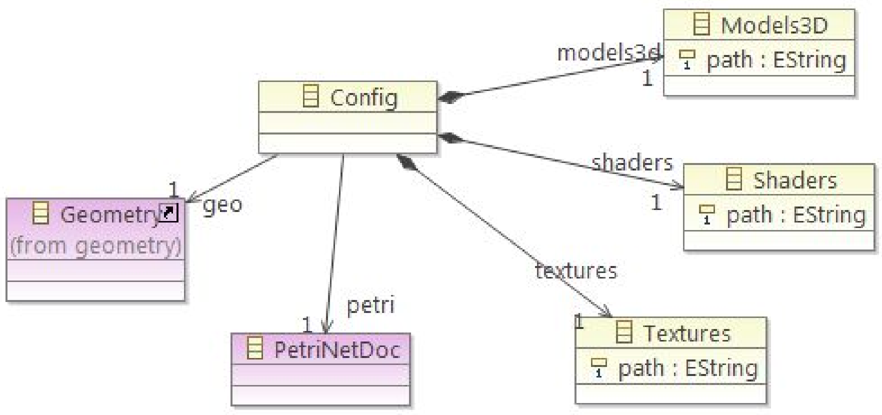


Figure 6.7: Configuration domain model

It consists only of one class that has references to a *PetriNetDoc* (the root element of a pnml document) and a Geometry (the root element of a Geometry document). The other classes represents the paths to the 3D models, the shaders and the textures.

The code of the projects "Config", "Config.edit" and "Config.editor" has been generated using the EMF. As we said in the intruduccion to EMF, this code contains the functionality to create a new config document, store it as xml, see it as a tree editor and modify it.

The project "RunCommand" contains the implementation of the command to run the project. This could be done either by right clicking on the Config element in the tree editor and selecting Run Simulator or by using the shortcut (Ctrl+2 on most platforms or COMMAND+2 on MacOS). The user won't be allowed to run the application if there are any resources missing (for example, we haven't selected the PetriNetDoc yet). When the program sees that the Config element has all the required elements, it will try to verify that all the elements that are referenced in the PetriNetDoc and the Geometry (such as textures and obj files) are present in the Resources that the user has selected (the folders with the 3d models, the shaders and the textures). If there are something missing, a popup menu will appear and the simulation won't start. Otherwise, the command will start the simulation. The details on how to actually set up the Config element can be found in the section 5 (User Manual).

The project "FillData" implements a contribution to the Eclipse workspace as a button. This button is the one the user **should** use to create and fill the objects with the information of the paths where to find the 3D Models, the Textures and the Shaders. This button **shall** be enabled only when the user has selected the Config element in the tree editor.

6.7 3D Visualization

The 3DVisualization component is, as we explained before, the module of the software tasked with simulating the PNVis and render the simulation in 3D. From the user's point of view it can be seen as just one component. Behind the scenes, however, that's not the case. The

3DVisualization functionality provided is divided in two modules: *PNSim* and *Engine3D*. *PN-Sim* is tasked with running the PNVis simulation, whereas *Engine3D* will take the information given by the simulation and draw it in 3D. Neeedless to say, the communication between this two modules is critical for our software so we will be explaining it in full detail. The following sections describe the *PNSim* module and the *Engine3D* module, along with the protocol of communication between them.

6.7.1 PNSim - Miguel

The Petri net simulator is one of the major components in this software. It provides functionality to simulate the behaviour of our own *PNVis*, which can be created and modified via the Petri Net Editor. This module will interact wil the Engine3D, as it will be the simulator the one who will be telling the Engine3D what to draw. The protocol for communication between the Engine3D and the PNSim is detailed in the Software Architecture section.

A very important remark must be made here, and that is that the PNSim does never modify a pnml document or even need one. The PNSim will recieve a *PetriNetDoc* object from the Configuration module. The pnml document from which it was created will not be modified in any way herefrom.

6.7.2 Explanation & Algorithm

As we've mentioned before, PNVis are a particular type of Petri net which we are using to model material flow or transport systems. This poses two problems:

1. Petri nets are discrete, which means that states change in steps. By contrast, material flow and transportation systems are continuous
2. *Tokens* relate to real world objects, so we need to do some work if our intention is to relate those tokens back to what they were modeling in the first place

The simulator is tasked with dealing with this two problems, and it does provide a solution for both of them. To overcome the discretization problem, we have the additional information provided by a PNVis which should be enough.
As for the second problem, instead of *tokens*, the PNSim will deal with *items*. Each *item* will have some attributes that will help the Engine3D draw them on scren. Thus, an *item* will have the following attributes:

- *Shape*: a string reference to the object the Engine3D needs to draw.
- *Geometry*: a string reference to the corresponding line in the geometry in which the item currently is.
- *Ready*: When set to *true* indicates that the *item* has finished its *activity*.
- *Activity*: a reference to the current animation of the *item*

The items are shared among the PNSim and the Engine3D so when the PNSim changes any attributes of a given item, then the Engine3D is notified of such changes. As the items move through the *Places* of the PNVis some of its attributes will change, such as *geometry*, *activity*, and *ready*. This will give clear indications to the Engine3D as for where to draw and how to animate a given *item*.
From a high level perspective, this is the algorithm for the simulation:

```
if item is ready:
  if item.activity isn't the last activity of the place it's in:
    setActivity(nextactivity)
    setReady(false)
  else:
    for each adjacent Transition:
      if is enabled:
        fire(Transition)
```

While arguable that the Engine3D should take full responsability for the handling of the animations, we decided to implement it this way so as to relieve work from the Engine3D and move it to the PNSim.
If, when running the algorithm, any of the adjacent transitions fires, then the simulator **shall** ensure that the effects are propagated, i.e., if as a consequence of the last firing another transition becomes enabled then it's fired, and so on.

Figure 6.8 shows the class Diagram for the PNSim, along with the packages of the interfaces associated with it. As shown, it features two qualified associations, one of which is named itemmarking and the other one placemarking. Both serve to save the state of a running instance of a simulation of a PNVis. We have designed it this way for efficiency reasons: sometimes it's prefereable to search indexing by *place*, whereas other times it's better to search by *item*. This implies additional control in order to keep both consistent, however it this extra measure is well worth the price paid in both memory space and consistency measures.

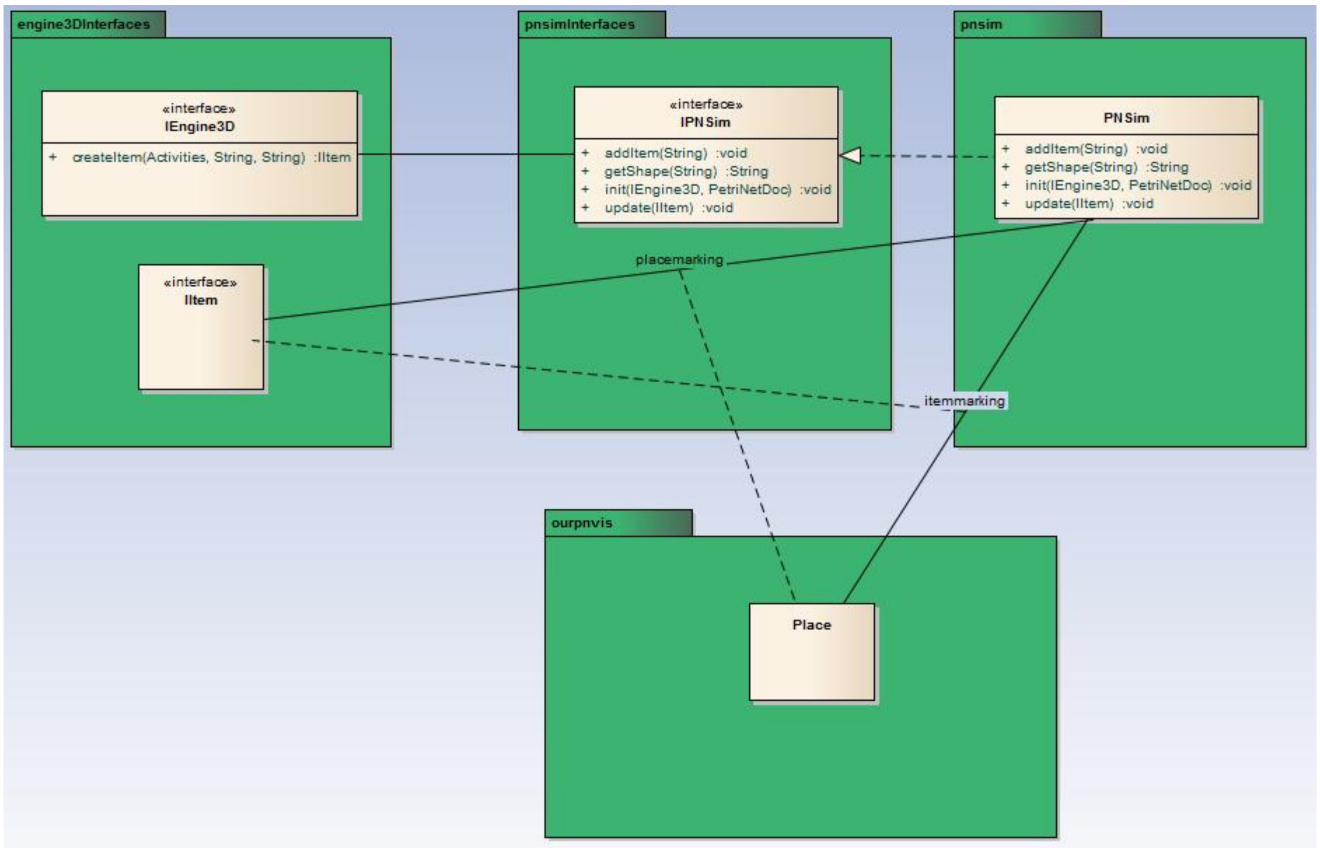


Figure 6.8: PNSim package Diagram

PNSim is the module responsible for keeping track of a PNVis and simulate its behaviour according to the PNVis rules and the interaction with the Engine3D.
PNSim implements the interface IPNSim, which is the interface exposing all necessary methods for the communication with the Engine3D, as we will explain in the next section.

6.7.3 PNSim - Communication between PNSim and Engine3D

As seen in Figure 6.8, the interface IPNSim has all the methods necessary for the interaction between PNSim and Engine3D. They are: *init*, *getShape*, *addItem* and *update*.

- The *init* method initiates an instance of a simulation of a PNVis (which is passed in the PetriNetDoc argument) and an Engine3D with whom to communicate.
- *getShape* returns the *shape* of the place which has the corresponding *geometry* given in the argument. If several places have the same *geometry*, then it can't be predicted from which place the *shape* will be returned. This method is called whenever a user clicks on a track of the 3D visualization. The Engine3D would like to know if the shape of the IItem to be created can fit in the track being cliked and does not incur into any colisions.

- The method *addItem* is called whenever the Engine3D wants to add a new IItem in the simulation as a consequence of the user clicking in a track of the visualization, and the IItem can actually be drawn without its drawing resulting in any collisions whatsoever.
- The method *update* is called whenever the current activity of an IItem has finished. This method will simulate all the valid steps in the simulation and change the attributtes of the IItem given as an argument with the results of the simulation.

Note that all this methods will make asynchronus changes to the Engine3D which will only take place in the next run of the main loop of the Engine3D. For example, the PNSim is not responsible for initializing an animation. Instead, it will change the *IItem* attributes and the Engine3D will then react accordingly. More information on this can be found in Section 6.7.7. Let's explain more in detail each part of the protocol.

6.7.4 PNSim - Init

Once the Engine3D is ready to start simulating, it will make a call to the init method from the PNSim. Init will read the PetriNetDoc *filename* and for each token it finds it will make a call to createItem, which returns an *IItem* succesfully created. Once this is done, the control is given back to the Engine3D

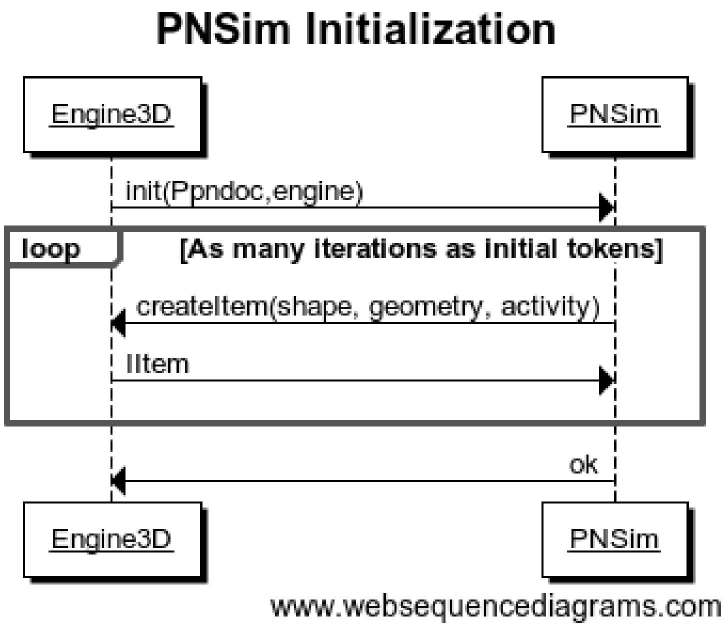

PNSim Initialization
Engine3D
PNSim
init(Ppndoc,engine)
loop
[As many iterations as initial tokens]
createItem(shape, geometry, activity)
IItem
ok
Engine3D
PNSim
www.websequencediagrams.com

Figure 6.9: Init Protocol

6.7.5 PNSim - Update

A call to update happens everytime an *IItem* finishes its activity, that is, sets its *ready* attribute to *true*. Here is where the algorithm of the PNSim (which was briefly described in section

3.2.2) runs. Each time the update is called, the following algorithm is executed:

```
if item is ready:
  if item.activity isn't the last activity of the place it's in:
    setActivity(nextactivity)
    setReady(false)
  else:
    for each adjacent Transition:
      if is enabled:
        fire(Transition)
```

First, we look at the place the *IItem* is in to see if the animation that finished was the last one of the place. As each place can have, in principle, any number of animations it is established that an IItem **must finish all the animations of the place they're in** before they are allowed to move forward.

If this requirement is met then all reachable transitions must be checked to verify if they are enabled for firing, and if they are, fire them and propagate its effects accordingly. Whenever an *IItem* reaches a new place some of its attributes must be replaced by the new values that are kept in the new place. For example, the geometry may change to refelct the new position of the object, ready will be always set to *false*, and the first animation of the new place will be set as the current activity. A shape may be changed if and only if the new place has its *canChange* attribute set to true. Then the new shape associated with the place replaces the current shape of the *IItem*.
Each time an attribute is set the Engine3D is notified as it needs to know which *IItems* have changed and react accordingly.

When no more transitions can be fired, the control is given back to the Engine3D.

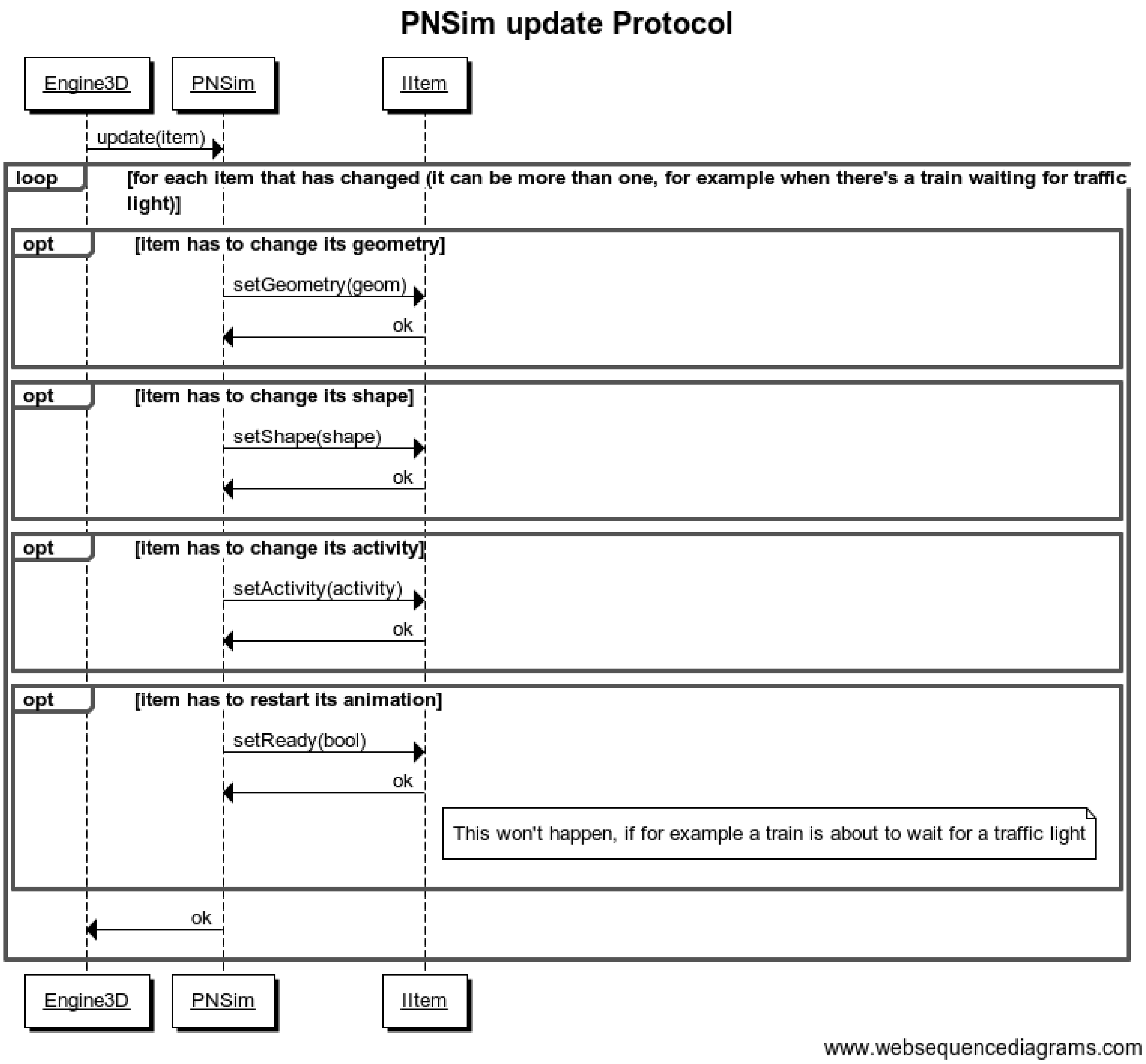

PNSim update Protocol
Engine3D
PNSim
IItem
update(item)
loop
[for each item that has changed (it can be more than one, for example when there's a train waiting for traffic light)]
opt
[item has to change its geometry]
setGeometry(geom)
ok
opt
[item has to change its shape]
setShape(shape)
ok
opt
[item has to change its activity]
setActivity(activity)
ok
opt
[item has to restart its animation]
setReady(bool)
ok
This won't happen, if for example a train is about to wait for a traffic light
ok
Engine3D
PNSim
IItem
www.websequencediagrams.com

Figure 6.10: Update Protocol

6.7.6 PNSIM - ADD ITEM

Takes place whenever the user clicks on any track in the 3D Visualization. This action triggers the creation of a new train in that track.
The engine3D will first ask for the shape associated with the geometry track that was clicked. If such shape can't be drawn for some reason (for example because it would collide with another object) then the protocol ends here. If this is not the case, then the Engine3D will call

the method *addItem*. The PNSim will ask the Engine3D for a new *IItem*, which will be set up according to the place that corresponds to the geometry track that was clicked. If two places have the same geometry associated then it can't be predicted in which *place* the *IItem* will be added.

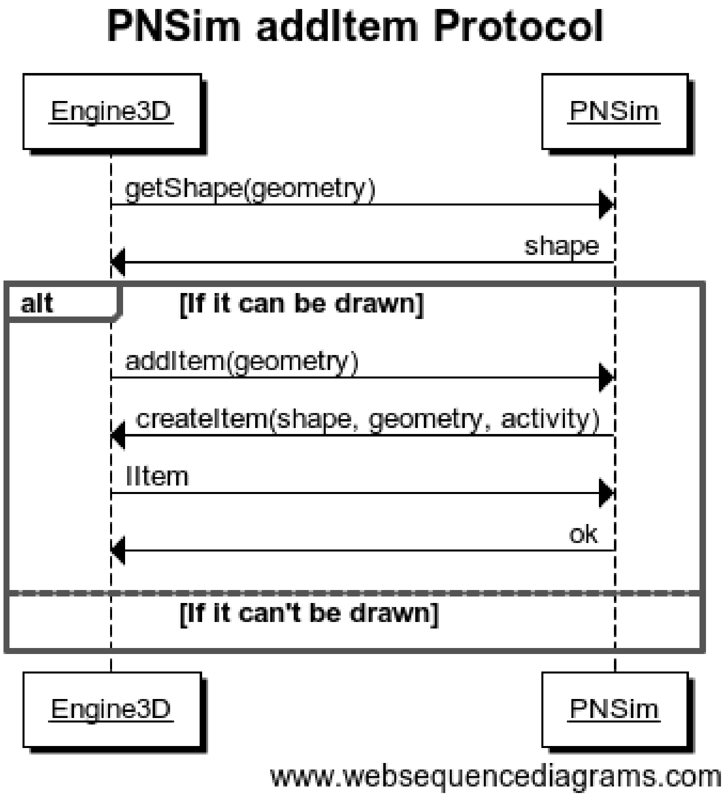


Figure 6.11: AddItem Protocol

6.7.7 ENGINE3D - MIKKEL AND SIMON

This subsection explains the architecture of the Engine3D implementation. The Engine3D makes use of a combination of the two main objects. A SceneObject and a SceneComponent. A detailed explanation to every single component of the diagram can be found in the following text. Figure 6.12 shows a component diagram of the Engine3D.

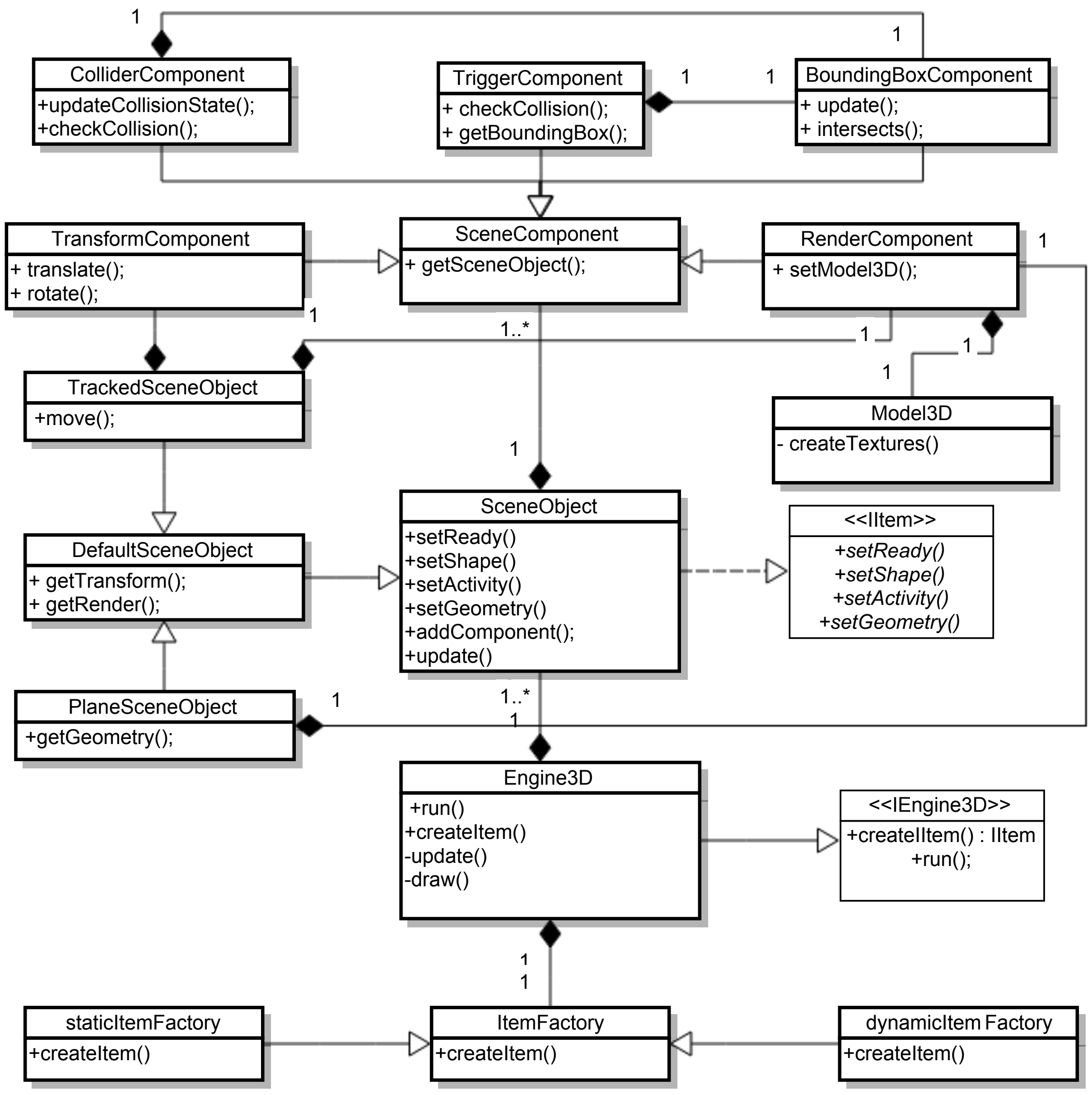


Figure 6.12: Engine3D component diagram

- **IEngine3D** represents the interface between the *Engine3D* and the Petri Net Simulator (*PNSim*). Used to delegate a *create item* call from the *PNSim* to the *Engine3D*. The *Engine3D* calls the ItemFactory to receive a *SceneObject* that is casted to an IItem and send back to the *PNSim*-
- The **Engine3D** is main component for the visualization of the simulation. Draws ob-

jects with *RenderComponents* on the screen, handles user input and updates the positions of moveable objects with *TransformComponents* around.

- **IItem** represents the interface between *PNSim* and *Engine3D*. Provides the functionality to call setters and getters from the *Engine3D* which the *PNSim* can use to change the necessary behvaior of objects such as change the animation of animation or movement of a SceneObject.
- **SceneObject** is the base object of the visualization. It contains a collection of *SceneComponents* which allows the object to have one or more behaviors. The *SceneObject* can be casted to an *IItem* that can be used for sending to and receiving from the *PNSim*. The PNSim itself can change the state of a SceneObject. The shape can be changed, which results in a different appearance (Texture) of the object. The ready state can be set which is used to determine if an objects needs to be updated by the *PNSim* or not. The activity is used to allow different behaviors for the object. It is possible to make the object wait (so it does not move around) or move around which translates the *TransformComponent*. Further an object can be triggered (if it has a *TriggerComponent*) for example by a user clicking with the mouse on an object.
- **SceneComponent** is the base component for the behavior of a SceneObject. All behavior components inherit from this base class. Every SceneComponent knows to which SceneObject it belongs to (which ca be important for example if the Engine3D needs to know to which SceneObject the TriggerComponent belongs to after having detected a collision with the mouse click).
- **TransformComponent** represents the movement behavior component. When an object contains this component it is able to move around in the 3D space.
- A **RenderComponent** is the drawable behavior component for an object. When an object contains this component it is able to be drawn on the screen. A Model3D has to be added to the Component to draw textures (images) or more complex 3D Objects such as trains ships or airplanes.
- The **ItemFactory** contains concrete factories such as the *DynamicItemFactory* and the *StaticItemFactory*. The concrete factories are used to create the *SceneObject*. The command for creating a *SceneObject* comes initially from the *PNSim* and is delegated to the *Engine3D*. The *Engine3D* than makes the final call to the *ItemFactory* that returns a *SceneObject* corresponding to the passed shape attribute of the createItem function call.
- **StaticItemFactory** is a concrete factory that is used to create still standing *SceneObjects* that do not move or animate e.g. simple train tracks.
- **DynamicItemFactory** is a concrete factory that is used to create moving or animating *SceneObjects*.
- The **DefaultSceneObject** has the ability to be rendered and drawn according to its two attributes, the *RenderComponent* and the *TransformComponent*.

- A **TrackedSceneObject** inherits from the *DefaultSceneObject* and has the ability to be moved in the 3D space according to its origin and target vector (that are represented with the *Transform component*). With the RenderComponent the visible appearance of the TrackedSceneObject can be changed. If the object should not be able to collide with other objects a ColliderComponent can be added to the components collection which takes care about signaling a future collision which can be handled properly (e.g. stopping the movement).
- The **PlaneSceneObject** is used to draw simple flat 2D objects such as tracks, streets or the ground.
- The **BoundingBoxComponent** contains a bounding box which is used to define the minimal guise of an object in form of a box.
- A **TriggerComponent** is used to allow an object to be triggered. A TriggerComponent has a BoundingBoxComponent which allows to detect collision of a mouse click in the BoundingBox.
- The **ColliderComponent** makes use of its BoundingBoxComponent. By checking if the BoundingBox of the BoundingBoxComponent collides with points, lines or other BoundingBoxes the component allows to detect collisions. The internal state of the ColliderComponent can be used to set and get the value if the component is colliding or not.

7 Appendix

7.1 Glossary

1 DTU, Danmarks Tekniske Universitet.

2 PNVis, Petri Net Visualization, our new type of Petri net which has the necesary information to simulate it. More details can be found in section 1.4.

3 pnml document, extension of the document were a Petri Net (of any type defined in the ePNK) is stored.

4 Eclipse framework, the IDE where the final product **must** be runned.

REFERENCES

[1] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, Harvard University, March 1997. http://www.normos.org/ietf/rfc/rfc2119.html

[2] http://www2.imm.dtu.dk/courses/02162/e13/project/, Ekkart Kindler *Project Notes for Software Engineering 2* 2013.

[3] You can find here a guide on the ePNK, http://www2.imm.dtu.dk/ ekki/projects/ePNK/PDF/ePNK-manual-1.0.0.pdf

[4] EMF, Eclipse Modeling Framework. You can find information about these here: www.eclipse.org/emf/

[5] You can find information on the Eclipse framework here: http://help.eclipse.org/kepler/index.jsp and also in the tutorials: http://www.vogella.com/articles/EclipsePlugIn/article.html

[6] Gamma, Helm, Johnson, Vlissides: Design Patterns. Addison-Wesley 1995.

[7] 3D-Visualization of Petri Net Models: Concept and Realization Ekkart Kindler and Csaba Pales: http://www2.imm.dtu.dk/courses/02162/e13/project/PDF/PNVis-PN04.pdf

[8] Graphical Modelling framework. More info at: http://en.wikipedia.org/wiki/Graphical_Editing_Framework

[9] http://lwjgl.org/