DTU - DENMARKS TECHNICAL UNIVERSITY

Software Engineering 2 (02162)

Handbook - Group C

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

17. November 2013

Technical University of Denmark

CONTENTS

1 INTRODUCTION - 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.
The purpose of this document is to provide a user guide for the software product developed by group C as part of the course Software Engineering 2 (02162).

This handbook will cover all the functionality described in the system specifications document and is intented both for users who have just started using our software and experienced alike. There are no previous prerrequisites for readers but basic computer skills. Previous experience with the Eclipse framework is also advisable but not necessary.

The document is divided in eight sections:

- **Introduction**: The section we are currently in. It contains an introduction to Petri Nets and a brief explanation of PNVis
- **Installation**: This section will cover all you need to know on how to set up Eclipse and the software in your computer.
- **Overview**: This section covers all you need to know for creating a project in Eclipse to run our software
- **Creating a PNVis**: This section will cover all you need to know on how to create, save, edit and validate a PNVis.
- **Creating a Geometry**: This section will cover all you need to know on how to create, modify and validate a Geometry model.
- **Configuration**: This section covers the creation of a config model and how to set it up properly to run our software
- **Runtime**: This section will explain how to setup a simulation and how to interact with it once started.

The following subsections will help you get acquainted with what Petri Nets are, what are they good for, and how are they implemented in our software.

1.1 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. 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.

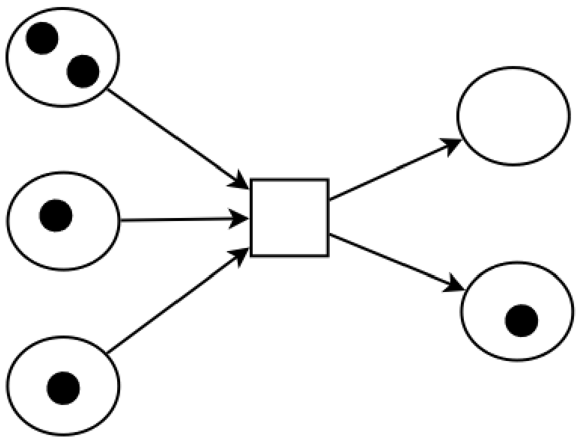


Figure 1.1: Before firing

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.

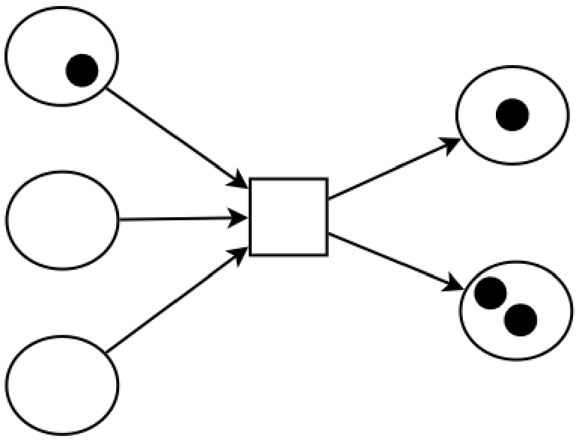


Figure 1.2: After firing

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.

1.2 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:

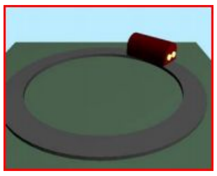


Figure 1.3: Graphical representation of a tracked based system

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

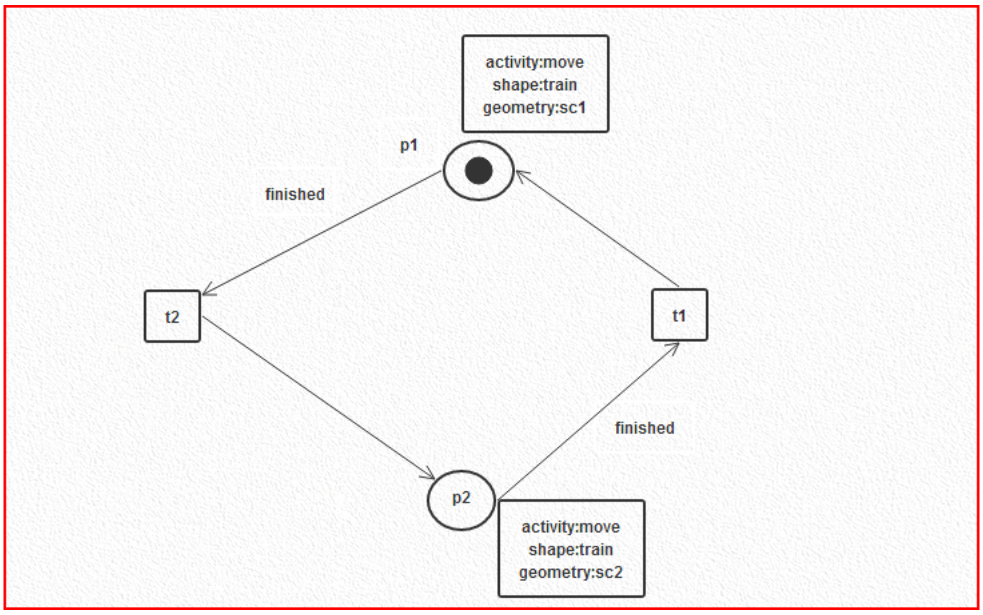


Figure 1.4: Abstraction of a tracked-based system

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

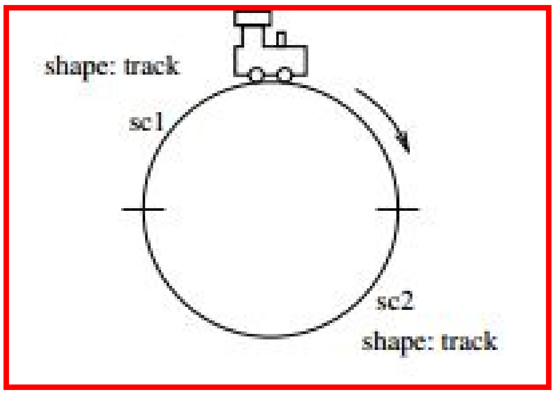


Figure 1.5: Geometry of the model

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:

[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)

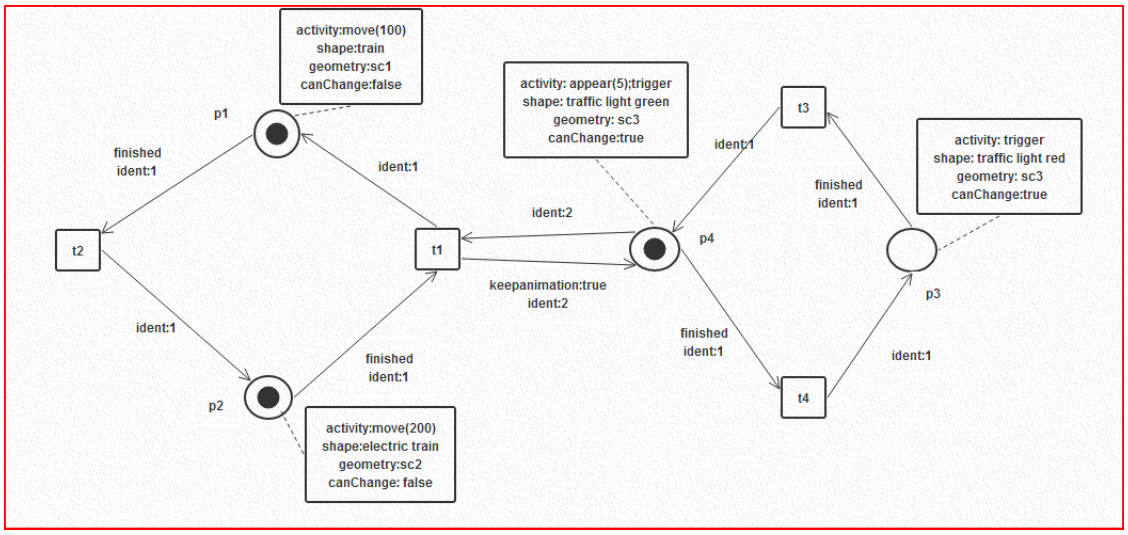


Figure 1.6: Example with the full version of a PNVis

And the corresponding geometry;

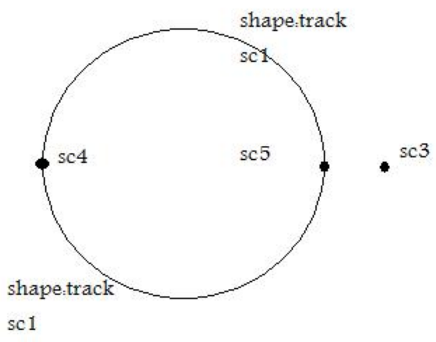


Figure 1.7: Geometry corresponding to PNVis of figure 1.6

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 INSTALLATION - JACOB

For this project, three things needs to be downloaded and set up.
First of all, the user will need the Java Runtime Environment version 1.7 or newer (which can be downloaded and installed from `www.java.com`).
Secondly, the user will need the Java SDK (which can be downloaded and installed from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`).
Thirdly, the user will need Eclipse 4.3 or newer (Eclipser Kepler) (which can be downloaded at `http://www.eclipse.org/downloads/`). The first two are installed normally through an installer, whereas Eclipse simply needs to be packed out into an appropriate folder.
Lastly, the user will need the ePNK which can be stalled by eclipses "Install New Software" feature, found under help → Install New Software (2.1). The url needed for the ePNK is http://www2.imm.dtu.dk/ ekki/projects/ePNK/kepler/update/ - this needs to be input into the "work with" field (2.2). All features should then be marked, followed by clicking next twice, accepting the terms and conditions, and clicking finish.

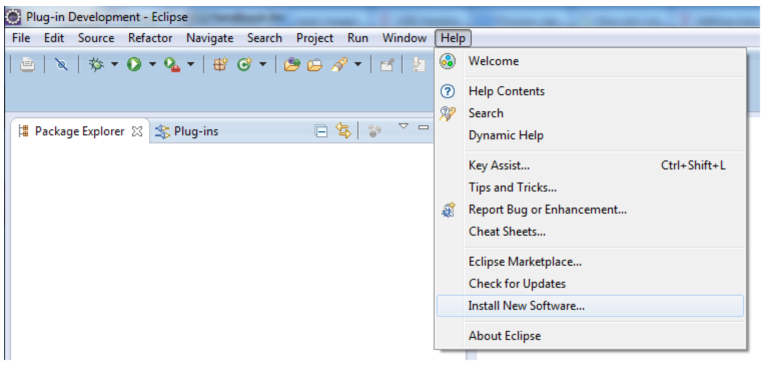
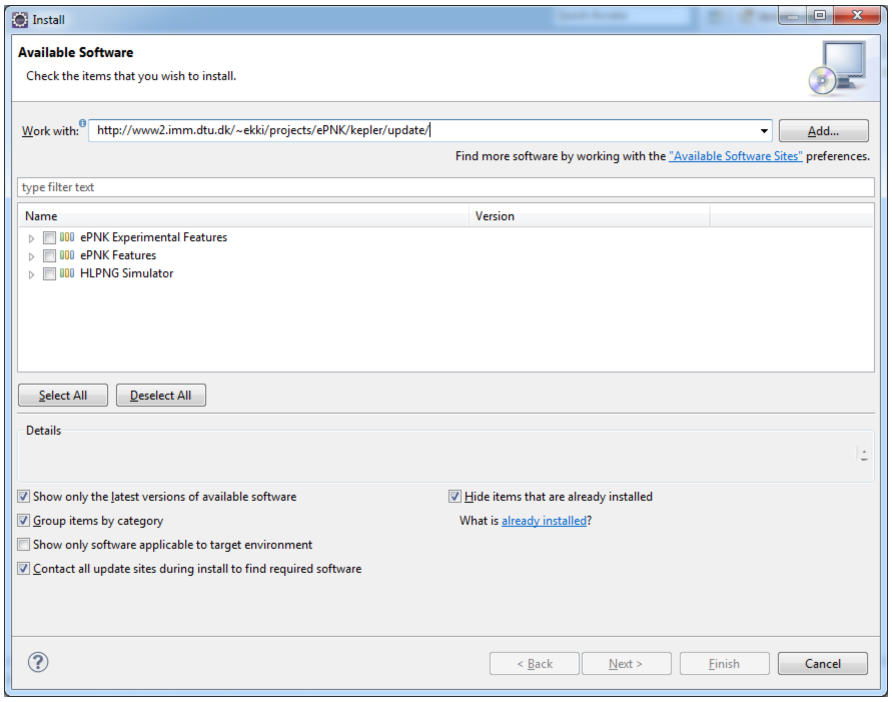


Figure 2.1: The Install new Software... button can be seen here


Install

Available Software

Check the items that you wish to install.

Work with: http://www2.imm.dtu.dk/~ekki/projects/ePNK/kepler/update/ Add...

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
ePNK Experimental Features	
ePNK Features	
HLPNG Simulator	

Select All Deselect All

Details

- [✓] Show only the latest versions of available software
- [✓] Group items by category
- [ ] Show only software applicable to target environment
- [✓] Contact all update sites during install to find required software
- [✓] Hide items that are already installed

What is already installed?

< Back Next > Finish Cancel

Figure 2.2: How the Install New Software looks.

Once the user has Eclipse and the rest installed, it is simply a matter of importing the plugins and running them as described in the following sections. This importing can be done through Eclipses built-in Import feature, and it can import many types - the easiest would be to have the plugins stored in an archive file (.zip or .rar for instance) and import → Archive File.

3 Overview - Diego

We want to simulate a tracked-based-system using Petri Nets. As we said in the Introduction, we modified the basic Petri Nets adding to them some tags to actually be able to simulate a track-based-system. This modification was called PNVis (Petri Net Visualization).

In addition to the PNVis, we also need a Geometry to bind the abstract concepts and models of the PNVis into the real world. As we explained in the Introduction, with this two elements we should be able to represent a tracked-based-system. But, if we look carefully into the definition, we can see that in the PNVis we have references to objects like trains or traffic lights and in the geometry we also have references to the shape of the tracks. To actually simulate a tracked based system, we also need to define those objects. In the case of the tracks it will be only an image file with the texture (what it is drawn on the screen) and to represent the trains (for example) we will need a 3d object and a way to draw it (to put the textures around it). So finally, to run the simulation will need to put all those elements (the PNVis, the Geometry and the files will the 3d models and the textures) together. This is the reason why we introduce another concept, the Configuration file. In this file, the user will be allowed to associate a particular PNVis with a Geometry and some runtime files (textures, 3d models...) and to run the application.

So far, to simulate a tracked-based-system with PNVis we need the following elements:

1 A file containing the PNVis. Our product provides a tool (the PNVis editor) to create, edit and store a Petri Net with the particular characteristics of a PNVis (see the Introduction for more details). This tool is explained in section 4.

2 A Geometry document containing the physical representation of our tracked based system (as we described on the Introduction). The tool provided to create, edit and store that files is described in section 5.

3 A Config document. As we said, this document will have references to the other documents and to some files. The Configuration editor provides the functionalaity to manipulate these files. A guide on it can be found in section 6.

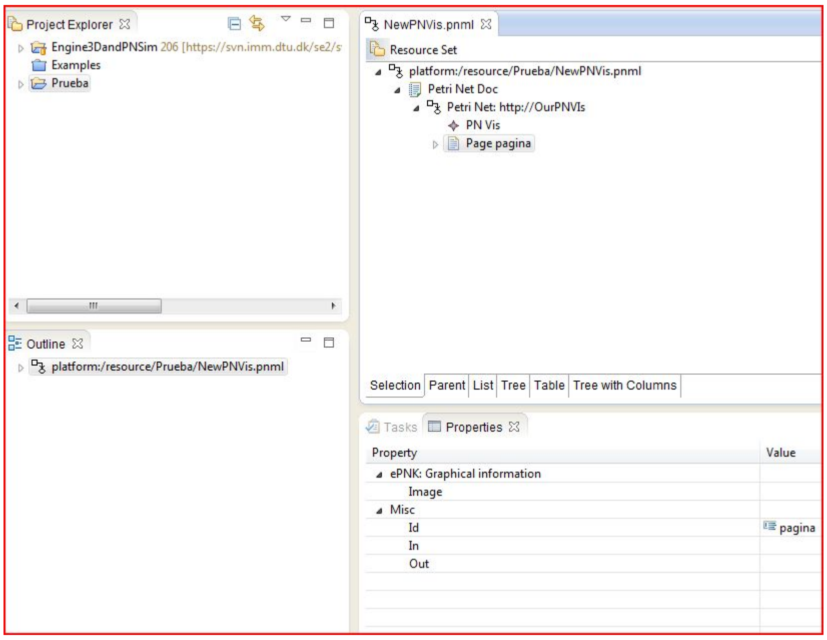
And at last, we want to actually see our simulation on the screen, in the section on the Configuration editor you can also find a way to run the application.

Once we've run the application, we would like to interact with the simulation. For example, we would like to change the color of the traffic lights, change the switches to make the trains go on other tracks or to just click on tracks to create more trains. The features that the simulation provides will be explained in section 7.

4 Creating a PNVis - Miguel

When you want to create a new PNVis all you have to do is select your runtime project in the Eclipse runtime workbench and in the context menu select "New..", "PNML document". Once you have your new PNML document opened in the tree Editor view, add a new Child

"OurPNVis", again through the context menu. From there you can add as many pages as you want to your document, just select "New Child.." and then "Page" from the context menu.


Project Explorer
Engine3DandPNSim 206 [https://svn.imm.dtu.dk/se2/s
Examples
Prueba
Outline
platform:/resource/Prueba/NewPNVis.pnml
NewPNVis.pnml
Resource Set
platform:/resource/Prueba/NewPNVis.pnml
Petri Net Doc
Petri Net: http://OurPNVIs
PN Vis
Page pagina
Selection
Parent
List
Tree
Table
Tree with Columns
Tasks
Properties

Property	Value
ePNK: Graphical information	
Image	
Misc	
Id	pagina
In	
Out	

Figure 4.1: Creation of a PNVis. You can see the current and only page, "pagina", is empty. The properties view is underneath

Double clicking in a Page will open the graphical editor for said page. From there you can start creating the PNVis itself by dragging and dropping Places, Transitions and Arcs. Remember that the editor will not allow you to connect a Place to a Place or a Transition to a Transition via an Arc.
We will proceed now to ddescribe the attribute list of each element. This properties can be modified from the properties view which by default is on the lower part of your Eclipse window.

- **Places**
  - *Shape*: This makes a reference to the 3D object to be drawn by the 3D engine. This will indicate at runtime the name of the texture file to load at said Place. Note that it only affects tokens created in that Place or tokens passing throught it if and only if *canChange* is set to true. The corresponding texture and .obj file should be inside the textures folder and 3dobjects folder, respectively.
  - *canChange* is a boolean, and it indicates whether or not the shape of an incoming *Item* should be forcefully changed when entering the place. This gives us more flexibility to create more complex track-based system. In principle, once an item is created, its shape is bounded to it and will not change anytime unless this attribute is set to true, then the new shape will be bounded

- *Tokens* set to true or false, this indicates whether or not a place has a token in it. This implies that only one token can be in a place at the start of the simulation. It is a desing choice which makes sense when translating it to the 3D Visualization, since two objects can't start in the same place at the same time.
- *Geometry*: This makes a reference to the line or point of the geometry on which the tokens will be drawn, and it is a name. If you are trying to model a traffic light or a track switch, this should make a reference to a *Point*. Else, it should be a reference to a *Line*. This attribute is set by attaching a *Label* to the place (drag and drop it, then connect it). See section 5 for more details.
- *Activities*: This 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. The syntax for them is "*animation*(*parameter, if any*);". Examples: "move(100);", "appear(50);move(5);", "trigger();move(99);". Any combination of them 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 should **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

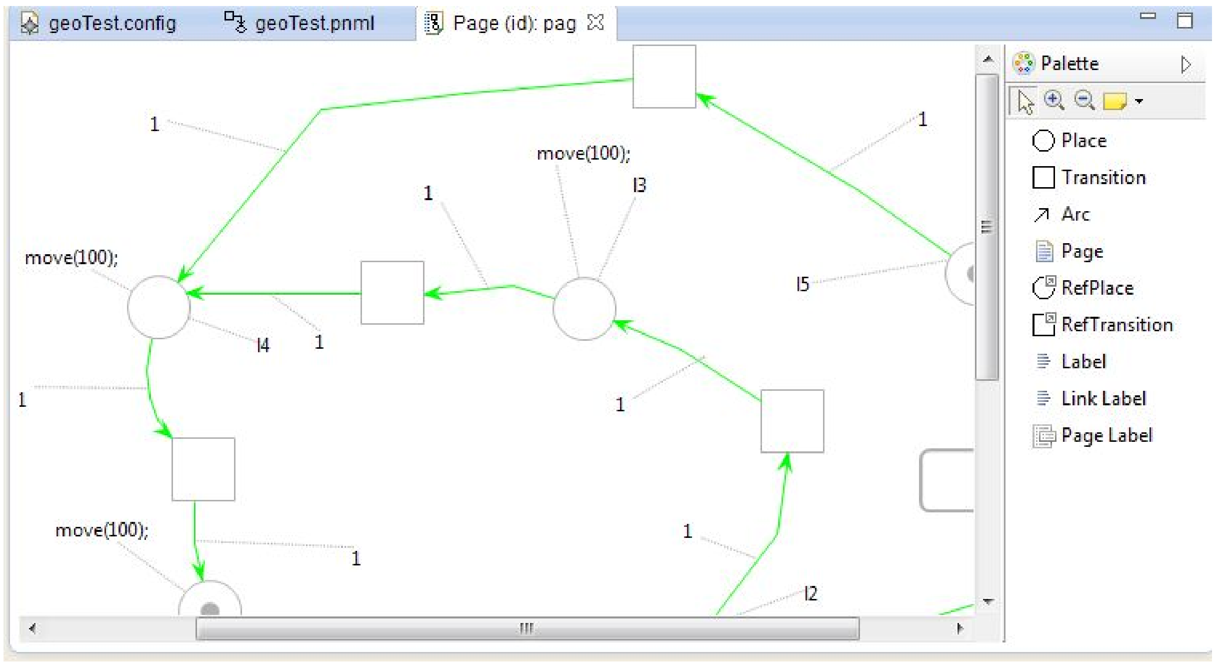


Figure 4.2: Graphical editor, with a Place selected and its attributes below

- **Arcs**
  - *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 must have the same number of incoming and outgoing arcs. Therefore for each incoming Arc with a specific *ident* tag, there needs to be and outgoing Arc with the same *ident* attribute. This tag is set by attaching a *Label* to the Arc (Drag and drop, then connect it to the Arc).
  - *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.
  - *KeepAnimation*: This tag can be set to true or false, 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. For example, if you're modelling a traffic light which is performing a trigger animation, you don't want the trigger to finish if the light is green, thus a transition can be fired without needing for the animation to finish.

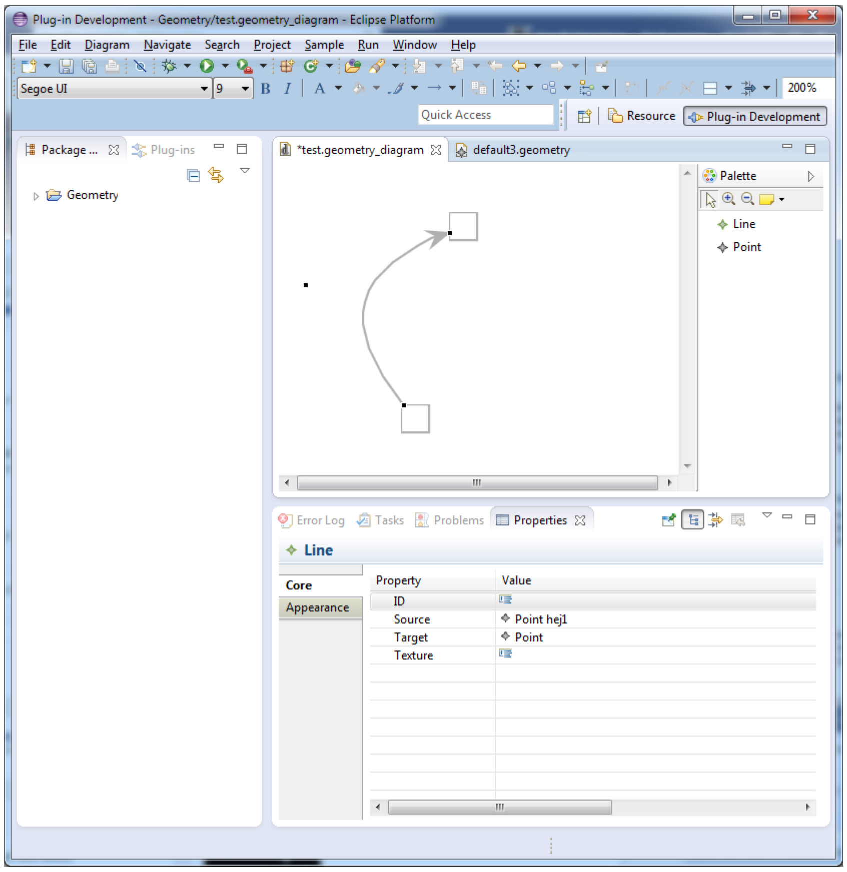
Besides the above, all elements have an *Id* string attribute which needs to be unique.

All the functionality supported by the graphical editor is also supported by the tree editor, and the latter should reflect all changes made in the former. Remember that you can save your project at anytime. Whenever you are finished, you can validate your document from your page's context menu. It will check if your file is correctly formed, but it does **not assure** that it is a valid file for simulation. By this we mean that it does not check if the referenced shapes are in the data folder, or if the geometry references exist in the geometry model. Part of this verification is done by the configuration before startup, so if any files are missing you will not be able to run the simulation.

Additional features implemented in this editor include: the "Fire Transition" command (right click in a transition and click in the command), colors for the arcs which are set automatically according to the tag *Ident* of the arc, and support for multiple pages.

5 Creating a Geometry - Jacob

A geometry is needed for this project to help bridge the gap between the Petri net model and the 3D visualization. This is done by giving a model of the situation by using only using three things - Points, Lines and Bendpoints. These 3 things are illustrated on 5.1 - the two squares are both points, the line connecting them are a line and the small black dot hanging to the left is a bendpoint.



Property	Value
ID	
Source	Point hej1
Target	Point
Texture	

Figure 5.1: A simple geometry

The Points are each a representation of a physical object - for instance, it could be a switch or a traffic light in a train-based model situation. The lines are what connects these physical objects - for instance, the tracks in a train-based model. Finally, the Bendpoints are used to create bends on the lines - the further they are dragged away from the model, the bigger the bend becomes. Finally, as can also be seen on the example above, the properties-view below shows the properties of a highlighted object.

5.1 Initializing a geometry

To create a new geometry you have to select your project on the Eclipse runtime workbench and in the context menu tab ”File” select “New” → “Other” → ”Geometry Diagram” âĂŞ this can be seen in 5.2.

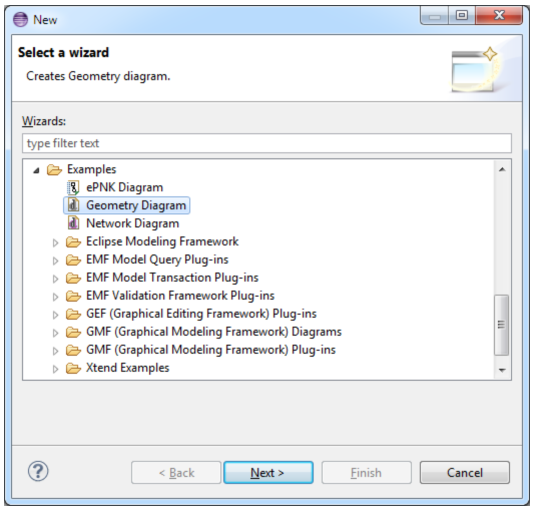
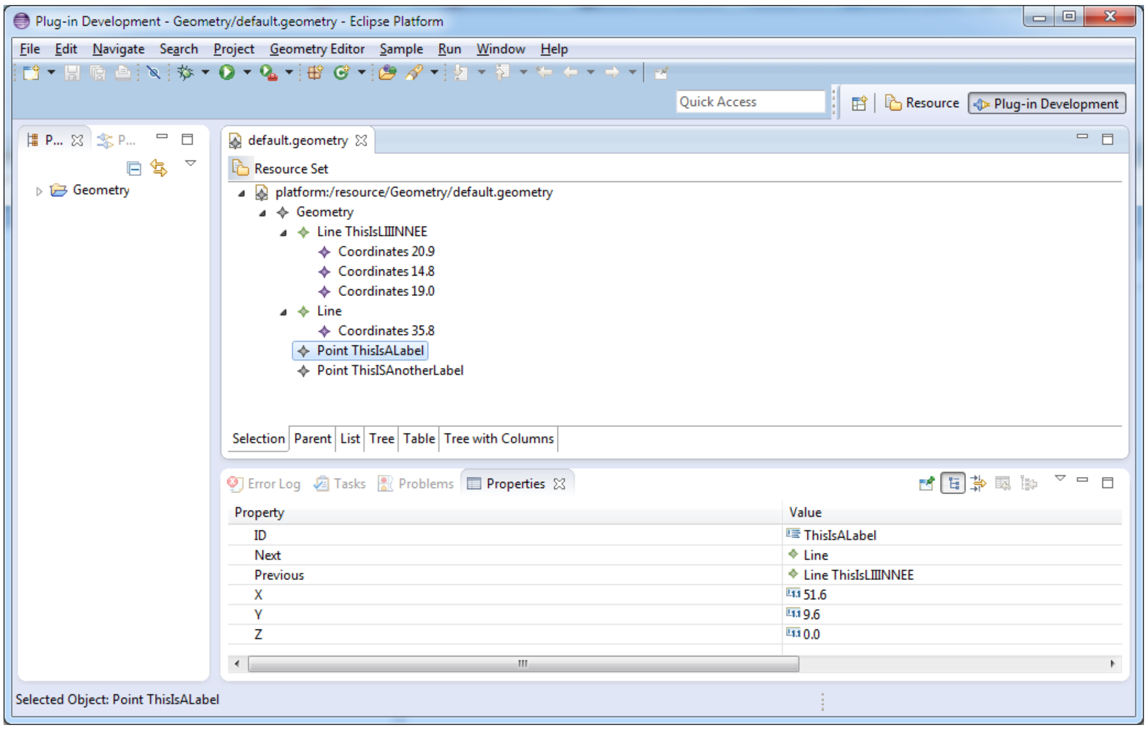

New
Select a wizard
Creates Geometry diagram.
Wizards:
type filter text
Examples
ePNK Diagram
Geometry Diagram
Network Diagram
Eclipse Modeling Framework
EMF Model Query Plug-ins
EMF Model Transaction Plug-ins
EMF Validation Framework Plug-ins
GEF (Graphical Editing Framework) Plug-ins
GMF (Graphical Modeling Framework) Diagrams
GMF (Graphical Modeling Framework) Plug-ins
Xtend Examples
< Back
Next >
Finish
Cancel

Figure 5.2: An example of creating a new Geometry. After clicking next, a suitable name should be picked - the initial name will simply be "default"

Your project now contains two new files; a "geometry" and a "geometry_diagram". The geometry file contains the model info, shown in a tree editor in which the raw data of each point, line and bend point can easily be viewed, and the geometry.diagram contains the model info, represented in a graphical editor

5.2 Geometry tree editor

The tree editor are provided to ensure easy readability of large projects - it makes it easy to spot just how many lines and points have been inserted, as well as viewing the amount of bendpoints. Finally, it also makes it very easy to view each points x, y and z coordinates, as well as giving an overview (through the properties menu) of which line is connected to which point and visa versa.
This tree editor should not be used for adding points, lines, bend points or changing coordinates, as these are not translated into the graphical editor - however, all changes in the graphical editor are translated properly into the tree editor.


Plug-in Development - Geometry/default.geometry - Eclipse Platform
File Edit Navigate Search Project Geometry Editor Sample Run Window Help
Quick Access
Resource
Plug-in Development
P... P...
default.geometry
Selection Parent List Tree Table Tree with Columns
Error Log Tasks Problems Properties

Geometry

Resource Set
- platform:/resource/Geometry/default.geometry
  - Geometry
    - Line ThisIsLIIINNEE
      - Coordinates 20.9
      - Coordinates 14.8
      - Coordinates 19.0
    - Line
      - Coordinates 35.8
    - Point ThisIsALabel
    - Point ThisISAnotherLabel

Property	Value
ID	ThisIsALabel
Next	Line
Previous	Line ThisIsLIIINNEE
X	51.6
Y	9.6
Z	0.0

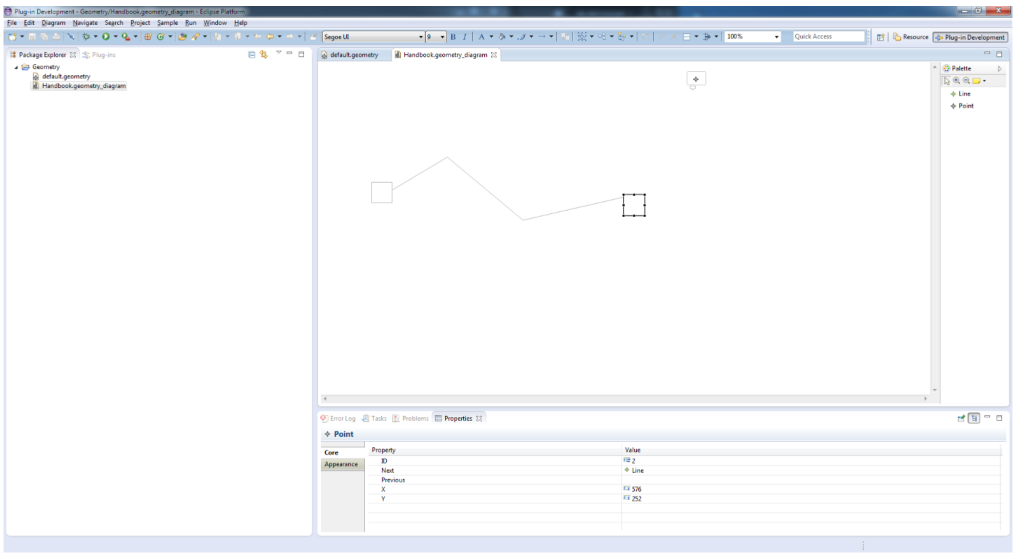
Selected Object: Point ThisIsALabel

Figure 5.3: A screenshot of tree editor.

On figure 5.3, the tree editor of the geometry can be seen with 2 lines, two points and 4 bend points inserted. Furthermore, the coordinates of the bend points can easily be seen, as can the name of each line and point, and the properties menu below shows all data of a highlighted object.

5.3 Geometry diagram

The geometry diagram is a graphical editor, in which you can add Points, as well as Lines between Points graphically. This is a visual representation of the Tree editor and every change done here will be stored and can also be viewed later in the tree editor.
Currently, the geometry supports creation of Points and Lines by drag-and-drop menu on the right side, as well as Bend Points that are made by clicking anywhere on a line and dragging the point to its appropriate place. Upon release of the mouse, a point is created and added to the Line and can later on be viewed in the tree editor - this point will also have coordinates which can be read by the visualization at a later point. An example of the geometry diagram can be seen here: 5.4



Property	Value
ID	2
Next	Line
Previous	
X	576
Y	252

Figure 5.4: Example of a geometry diagram with two points and one line between them that has two bend points.

6 Configuration - Diego

As we said in the Overview, this module is allows the user to associate a PNVis with a Geometry and a runtime data to run the simulation.

To do so, we store that information in a .config file. To create one of those files, we should use the wizard of Eclipse. This file is shown as a tree editor.

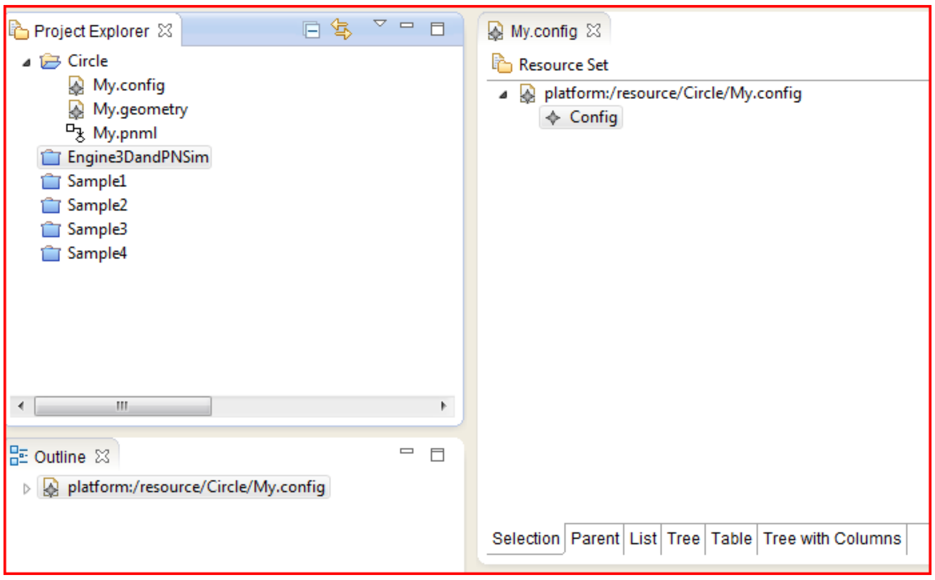


Figure 6.1: Example of a newly created configuration document

Now, we have to configure that document (i.e., associating it with a PNVis, a Geometry and some runtime data).

To associate it with a PNVis (contained in a pnml document) and with a Geometry (contained in a .geometry file), the user has to **Load** those resources in the document and then select them in the Properties view of the Config element.

To associate the Congif file with the runtime data, the user should create three different folders and place in each of them:

- The 3d models described as .obj files. To know how to create this files please take a look at the reference [1]. We only support obj files made of vertices (v followed by 3 coordinates), texture coordinates (vt followed by 2 coordinates) and faces (which must be triangles, a list of three vertices. To better understand this tags please check the reference [1] or use the default objects provided by the developers.
- The shaders with the intructions to the graphical card on how to render each primitive. Those files are provided by the developers of this tool with a default implementation but an experienced user may want to change it. There are two types of files contained here:
  - *.vert* files: A vertex shader containing per-vertex instructions for the graphics card driver; the code within this file is compiled at initialization and run for each vertex of an object that is being rendered (3 times for a triangle).
  - *.frag* files: A fragment shader (pixel fragment shader) containing per-pixel fragment instructions for the graphics card driver; the code within this file is compiled at initialization and run for each pixel fragment of an object that is being rendered (the number of pixel fragments per triangle varies greatly depending on the rasterization process [2]).
- The textures with the graphical appereance of the objects drawn in the screen. Our tool supports *.png* files to define the texture of the objects. Only one texture per 3d model. Regardless of the extension, those files must be called the same way.

And now, the user can associate this folders with the Configuration element. To do so, the user has to click on the Config element in the tree editor and then press the button with the shape of a blue train (you can see it in figure 6.2). Now three popup menus will show up asking the user to select the folder with the 3d models, the shaders and the textures. When the user has finished, he will see something like:

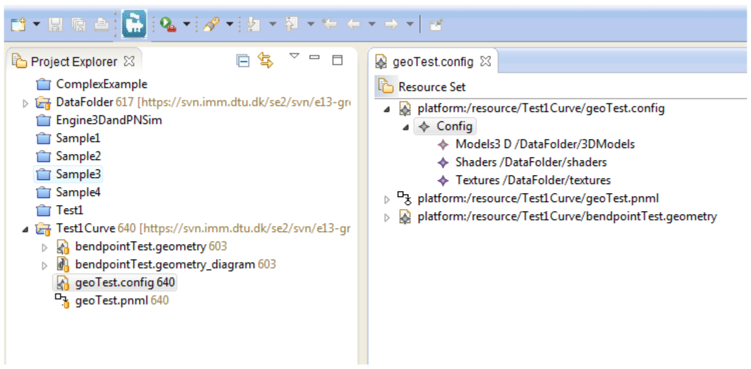

Project Explorer
ComplexExample
DataFolder 617 [https://svn.imm.dtu.dk/se2/svn/e13-gr
Engine3DandPNSim
Sample1
Sample2
Sample3
Sample4
Test1
Test1Curve 640 [https://svn.imm.dtu.dk/se2/svn/e13-gr
bendpointTest.geometry 603
bendpointTest.geometry_diagram 603
geoTest.config 640
geoTest.pnml 640
geoTest.config
Resource Set
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 6.2: Example of a project once we've filled all the data

Now we're ready to run the simulation. we just have to select the Config element of the tree editor of the Configuration file and right clicking on it a popup menu will appear. Select Run Simulator->Run and the simulation will try to start (Ctrl+2 is a shortcut on most computers, COMMAND+2 on MacOS). First, the command will check if all the references contained both in the Petri Net and in the Geomery are present in the folders (the textures and the 3d models) because otherwise the simulator won't know how to run the simulation. It will also make some verifications over the specific features of the PNVis (the constraint about the incomming and outgoing arcs of a Transition) and if there's any problem, the simulation won't start and the user will see a frame telling him the reason (what arcs are missing for example).

Apart from that, the user is responsible of making sure that the behaviour defined in the PNVis and the physical description of the Geometry make sense (if he wants to). For example, the user could define a Geometry with a Petri Net where the objects jump from one (physical) place to another because this is the behaviour described in the PNVis.

This is an exaple of a correct Petri Net and Geometry:

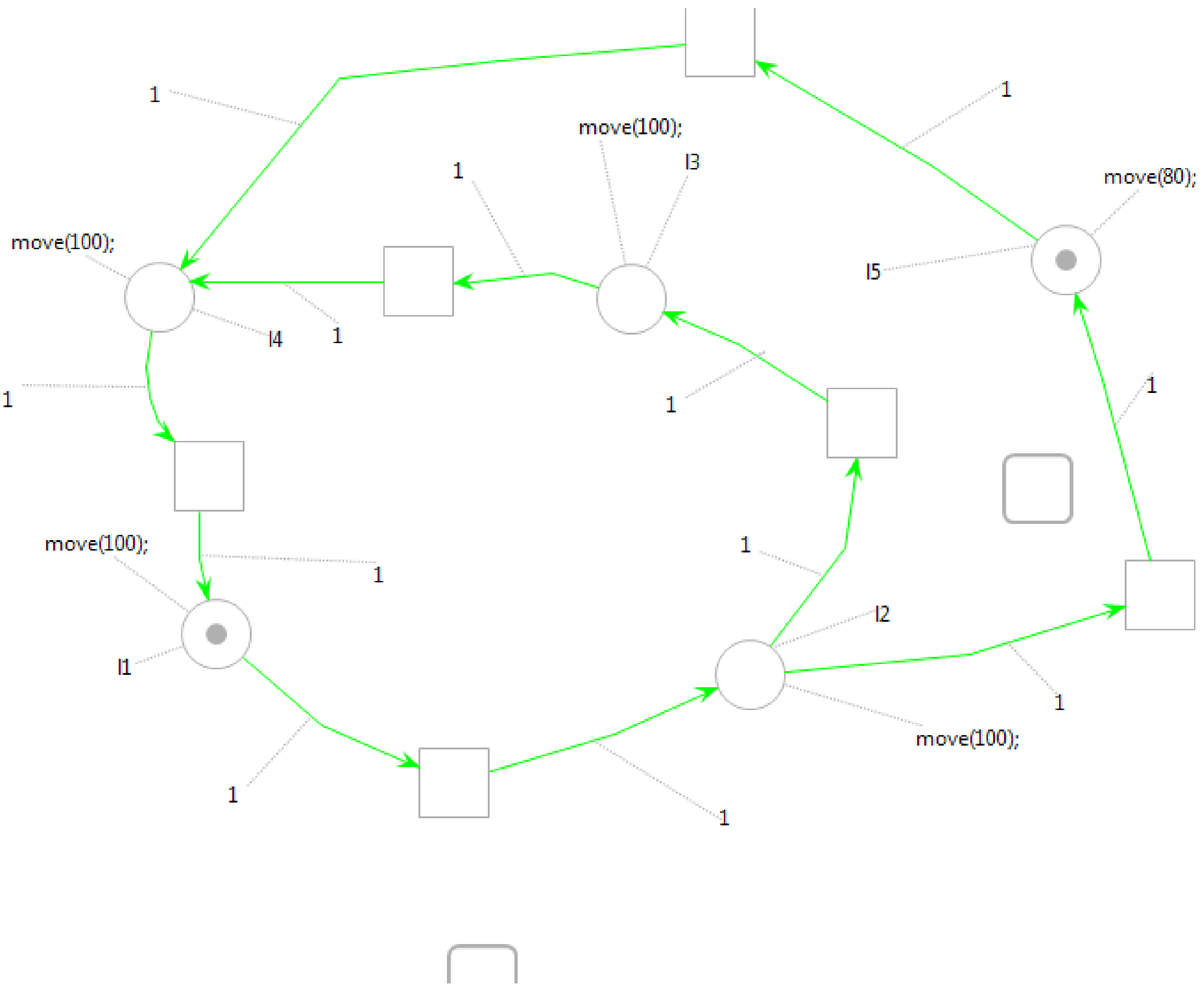


Figure 6.3: Example of a correct PNVis

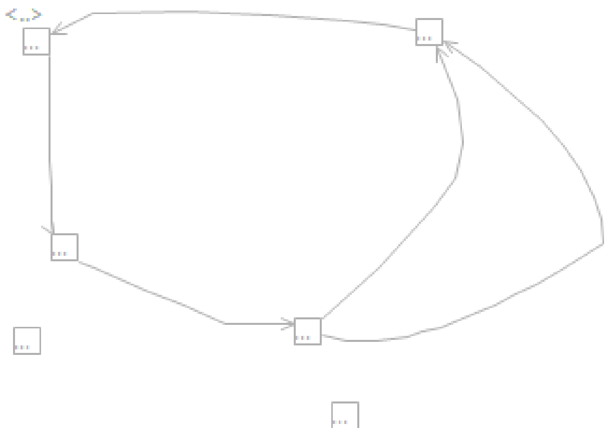


Figure 6.4: Example of a correct geometry

And the result will be:

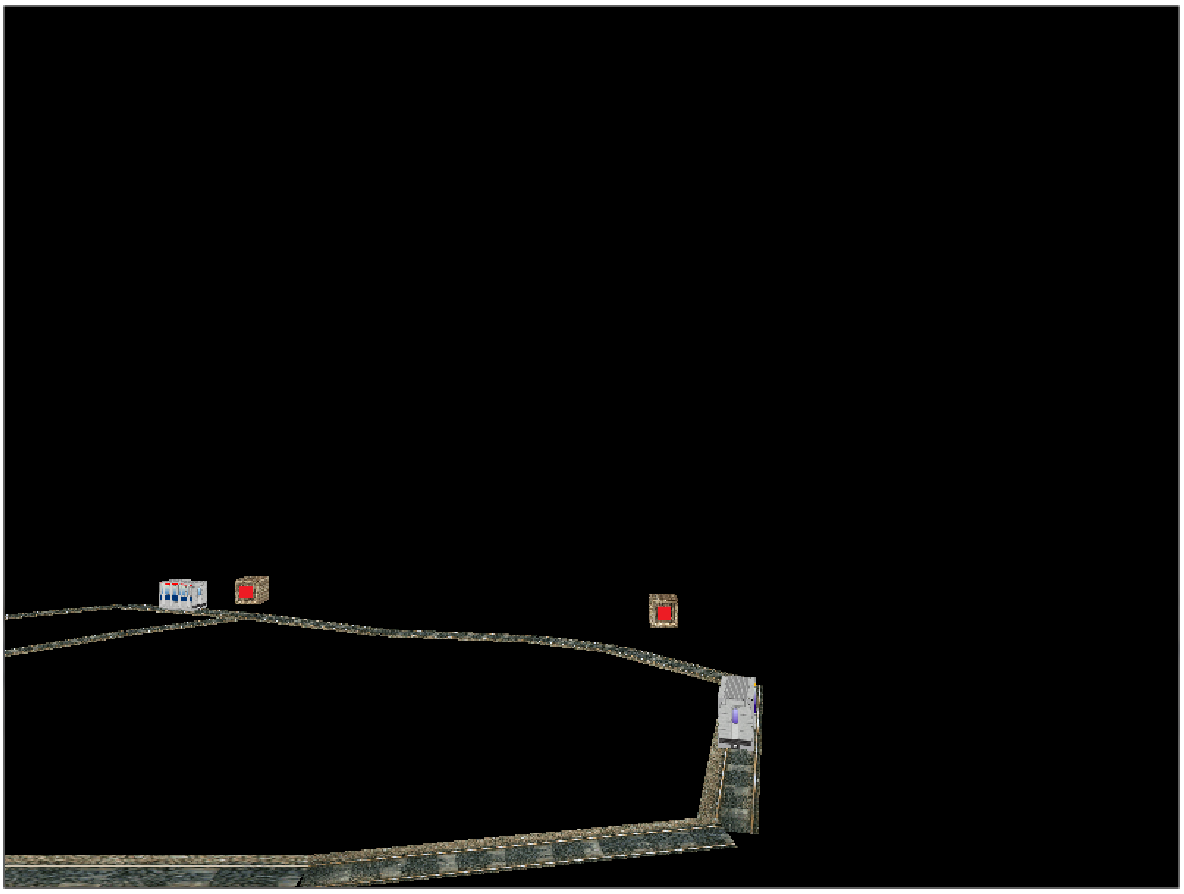


Figure 6.5: The result with the figures 6.3 and 6.4 and the default shape for trains and tracks

7 RUNTIME - JACOB

Once the petri net, the geometry and the config has been set up correctly, the project is ready to be run.Starting this is done by opening the .config file, expanding platform/resources/X.config (where X is the name of your file) untill you see Config. Once the user see the Config part, right click it, hover over Run Simulator and click Run (Alternatively, the shortcut is ctrl+2 in Windows). This process can be seen on 7.1

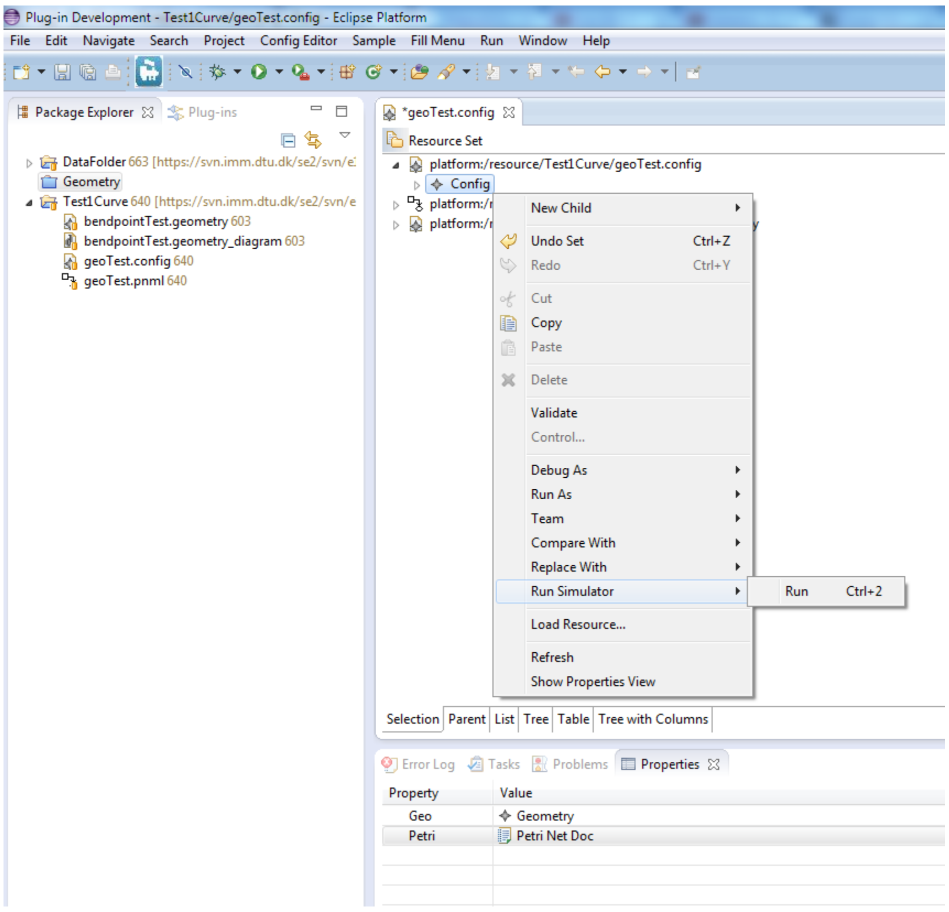

Plug-in Development - Test1Curve/geoTest.config - Eclipse Platform
File Edit Navigate Search Project Config Editor Sample Fill Menu Run Window Help
Package Explorer
Plug-ins
DataFolder 663 [https://svn.imm.dtu.dk/se2/svn/e
Geometry
Test1Curve 640 [https://svn.imm.dtu.dk/se2/svn/e
bendpointTest.geometry 603
bendpointTest.geometry_diagram 603
geoTest.config 640
geoTest.pnml 640
*geoTest.config
Resource Set
platform:/resource/Test1Curve/geoTest.config
Config
platform:/r
platform:/r
New Child
Undo Set Ctrl+Z
Redo Ctrl+Y
Cut
Copy
Paste
Delete
Validate
Control...
Debug As
Run As
Team
Compare With
Replace With
Run Simulator
Run Ctrl+2
Load Resource...
Refresh
Show Properties View
Selection Parent List Tree Table Tree with Columns
Error Log Tasks Problems Properties
Property Value
Geo Geometry
Petri Petri Net Doc

Figure 7.1: A screenshot of how to run the project.

REFERENCES

[1] To know more about these files consult: http://en.wikipedia.org/wiki/Object_file

[2] Rasterization: https://en.wikipedia.org/wiki/Rasterisation