A framework for building 3D animations on top of Discrete Events Systems simulations

Filip Łęczyński
AGH-UST Kraków
DTU Lyngby

Kongens Lyngby 2011

Abstract. In this thesis, I describe the concept and design of a 3D visualization system for Discrete Events Systems. Many products have been previously made, but presented framework is independent of any of DES notation. Moreover, I present a prototype solution for concrete example of using it with the Petri-Nets build using ePNK.

1. Introduction

Building a system, such as a traffic control or production system was always a challenging and difficult operation. In case of bad design, if a product has been already made, it could be very costly to replace it. If, for example, an assembly line is malfunctioning, it causes a lot of loss. Usually costs of redesigning and building the system again are unacceptable. Because of that, engineers invented methods to ensure that system will work correctly. One method of validating the system is to simulate it by creating a model and running a simulation on it. By that, an engineer can be sure that his concepts are correct, at least in that concrete model.

Another reason for modeling the system is a demonstration of concepts during early stages of system creation. In that case, a low cost, model of the system is presented to investors. When a model can be visualized in 3D, as it would look like in the reality, it eases to connect visions of the product. By that, all involved persons can be sure that they are thinking about the same final product.

Among many other modeling technologies, Discrete Events Systems (DES) can be found. This class of systems is based on the idea of discrete occurring of events. Noncontiguous in an event occurring is an important feature of DES and has great importance during building simulator for them. It will be described in detail later in this paper. Coming in various notations they allow to model even very complex system by using, usually, simple notation. Creating a simple simulator that can show the behavior and control flow in the model is a relatively easy task for most of the DES notations. Unfortunately, 'simply' means also that, in most cases, visualization of the system is meaningless. For example, Petri Nets come with graphical notation by default. However, that graphic is not easy readable, especially when someone is not familiar with the notation. In addition, it is even impossible to see the purpose of the model without additional documents, which describe meaning of model's parts. This might lead to misunderstandings, especially if someone interpreting the model is not familiar with Petri Nets.

In order to eliminate, or at least to limit, this disadvantage, a 3D-visualisation simulator can be used. It combines the DES model with 3D objects and environment, so that system can be seen more clearly. It helps to see the vision behind the model. Furthermore, it allows spotting the conceptual errors in the modeled system. This is the case when a model is correct in notation and working fine by itself. However, when an engineer adds visualization to the system, he might see that it does not do what it supposed to. Then he can correct the model. As for the non-engineers, they cannot usually see what is hiding behind the model. In case of Petri Nets, it is very difficult for most of the people to know what firing specific transition does in the system. When they can see, for example, move of a produced car, they are able to see the usefulness of the model.

Various 3D-visualisation simulators were created for different DES notations. They differ a lot in complexity and capabilities. However, most of them were created for one, specific notation or even subset of it. This limits a number of potential users. In most cases if for some reasons a user would like to change the simulator, then he would have to learn to use a new tool. In addition, it would be impossible or hard to port animations and configuration of 3D environment. That leads to repeat the work. Especially in bigger models, the amount of work would be too big.

The purpose of the framework created in this thesis is to allow a user to avoid above problems, both learning new tool and repeating already done work. It is achieved by creating a platform that completely separate configuration of the 3D environment from DES model. Additionally, it allows using it with different notation, if the simulator for it is provided.

In following sections, I describe my realization of concept of an independent 3D Visualization System for DES notations. Section 2 provides concepts and detailed description of proposed application. It also provides the requirement of the system. Section 3 goes through analysis of the problem that was solved in the thesis. Section 4 provides design of the framework. It covers both static and dynamic aspects of proposed solution to the problem. Section 5 gives a short overview of used libraries and technologies. Additionally, it gives reason for choosing those not others. Section 6 describes implementation details that are not covered by the design. Section 7 describes a model used for validation of prototype implementation. Additional it presents some screenshots from simulation. Section 8 provides few ideas of potential extensions to the existing implementation. Appendixes contain a complete ecore diagram of DES3dv project.

2. Concepts

In order to visualize a DES simulation, we need to provide DES model with some additional information. In addition, we need to connect and synchronize 3D models behavior and DES behavior. Since it is not the first resolution of a problem of creating the 3D visualization for DES systems, many concepts can be taken from previous papers. I will base my solution on PNVis project [1]. It purposes was to create 3D visualization for Petri Nets. I will also use Petri Nets notation to illustrate concepts used in my work.

On figure 2-1, we can see a simple example of Petri Net. It is composed of two places (p1, p2) and two transitions (t1, t2), connected by arcs. By definition, transition can be fired (execute action) when in incoming places there is at least one token in each (in simplest Petri Nets). Firing the transition means, that one token is removed from all incoming places and one token is added to every outgoing place.

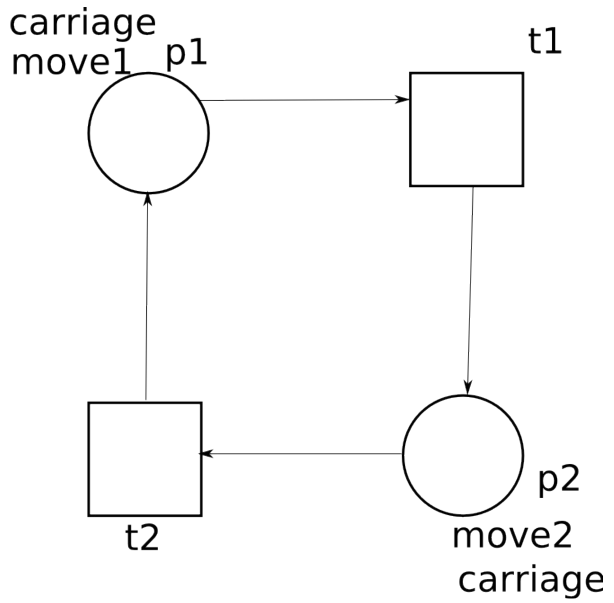


Figure 2-1 Example of Petri Net model

For the purpose of simulation, every token in the place is interpreted as independent object. That object can be visualized and animated. We will call this object a 3D model, as its main purpose in the framework is defining a shape in 3D space.

We also need to choose an animation for that object. All animations' types are predefined by the framework.

We can now add 3D models and animation to our example Petri Net, to give it meaning. On Figure 2-2, we can see labels that define 3D models and animations, representing carriage moving between two places in a factory. What we are now missing is a path along which carriage can move. We will define it in object called Geometry. Now we have a connection between Petri Net behavior and 3D models behavior.

One thing that needs to be resolved is passing 3D models between animations/places. For now by removing/adding token in place, we also destroy/create the 3D model. However, in our example, we can see that it is the same carriage moving between places. We resolve that by assuming that if both incoming and outgoing places for transition have the same 3D model label, then instance of the 3D model from incoming place is passed to outgoing place. By that, 3D models lives in simulation as they would in a real system. Additionally, we gain better performance by eliminating those redundant creations/destructions of 3D models.

Very important issue to resolve is synchronization between DES simulation and 3D simulation. In Petri Net, transition is fired as soon as all incoming places have at least one token. This means that DES simulation does not depend on time flow. In contrast to that, running most of the animations takes some time. In our example, it is obvious that moving carriage from one place to another requires some time. On the other hand, waiting for animation to finish is not always desirable. In some cases, we might want to stop animation when a corresponding transition can be fired. We deal with both cases by adding a label to incoming arcs of transition. If the arc is provided with label 'synchronize' transition cannot be fired until animation in place connected by that arc is finished. If the arc does not have that label, or if it is set to false, then transition can stop animation and fire.

To illustrate that, we can extend our example with a switch that allows, or not, carriage to move. As can be seen on Figure 2-2 transition t1 can be fired only when there are tokens in both places p1 and allow. Additional condition is that animation attached to place p1 is finished. However, t1 does not need to wait for animation in allow to finish. In contrast, transition t3 has to wait for animation in allow to finish. Animations in places allow and stop are of type trigger, which means that they are finished when they are triggered. Example use of that kind of animations is to react to user's clicks within the simulation window.

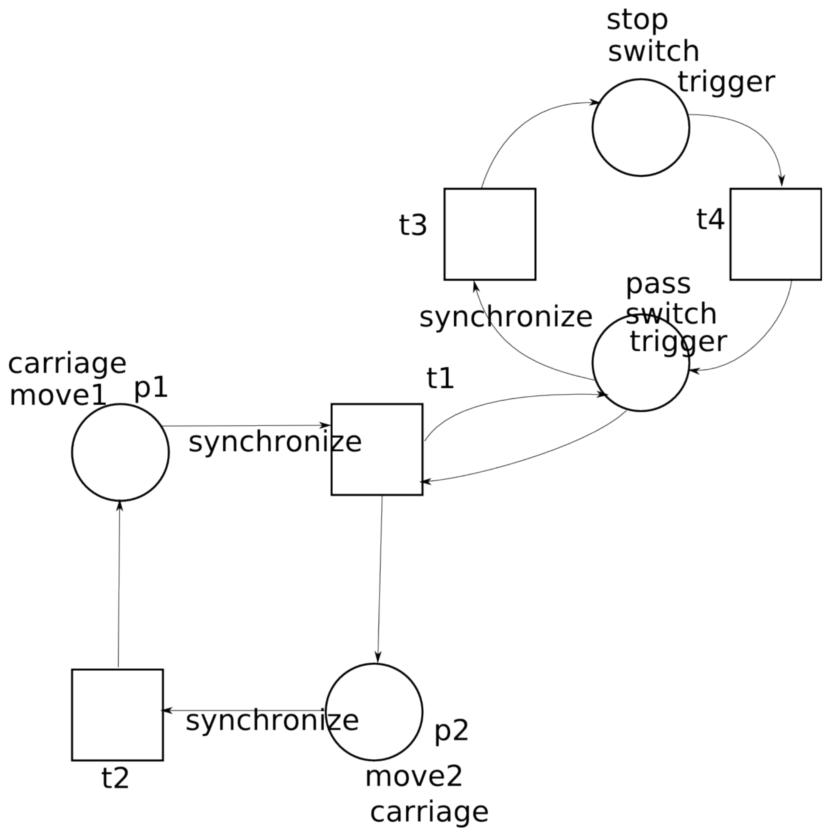


Figure 2-2 Extended version of example Petri Net model

Another thing to deal with is a place-transition pair connected in a loop. We can see that place allow and transition t1 creates that loop. Nevertheless, we cannot just finish animation in allow during firing transition t1. If we do that transition t3 can be fired as well. To eliminate that we check if animation in such a loop is of type of trigger. If it is, we do not destroy it during firing t1.

When we add 3D visualization to DES simulation, another problem occurs. Since we deal with moving objects in 3D space, there is a possibility that they will collide. It is likely to happen especially in situations when a user controls parts of the simulation by triggering animations. In our example, this can occur when we have two or more carriages, with overlapping paths of animations. If some carriers can be, hold and realized by the user, they can collide. Currently the framework tries to resolve collisions by itself without informing about collisions (explained in details later in this paper).

3. Analysis

- From previous sections, we can extract those main tasks of proposed application are:
- create new DES simulator
- create a new model for existing DES simulator
- allow to run a simulation of created DES model

This allows creating different types of users of application.

End user is a person that can start and interact with 3D part of simulation via mouse. This task does no require any additional knowledge than being a computer user.

Engineer is a person who knows the DES notation for a given DES simulator. He also has knowledge of application. That allows him to create new DES models that can be simulated. He also must provide all graphical and synchronization informations to model, which requires a bit of 3D modeling skills and overall 3D knowledge.

Another user, named developer, is the one with the most width knowledge. At that point, he must not only be an expert in concrete DES notation but also have some designing/programming skills. Those two allow him to create a new DES simulator that can be used with a framework designed by me.

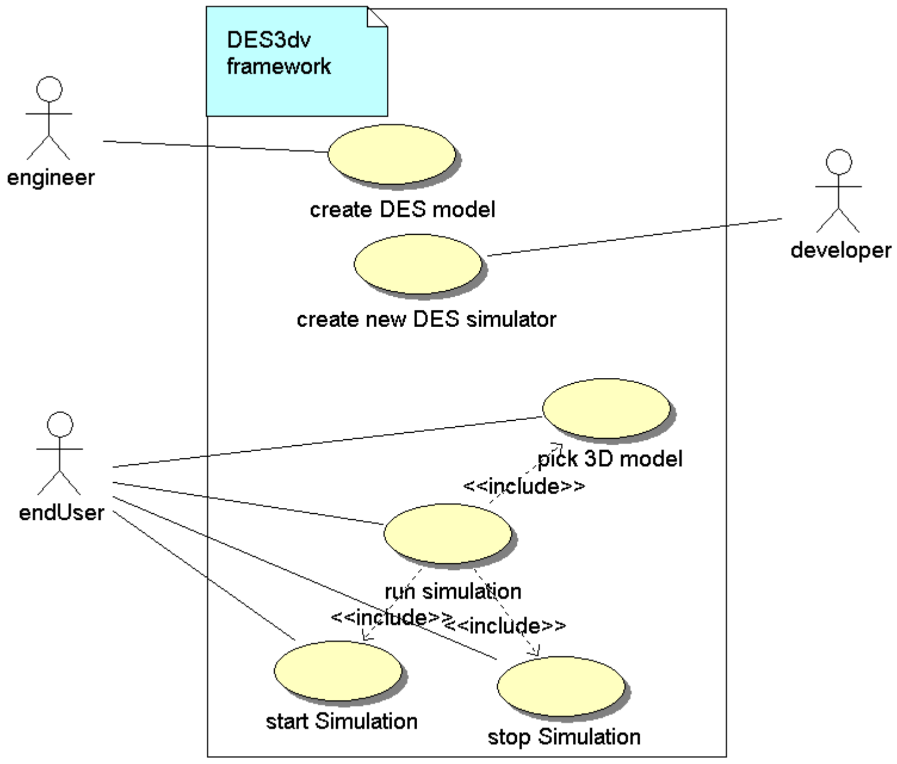


Figure 3-3 Use cases for DES3dv framework

During CreateDES Simulator use case, developer crates new DES simulator for the framework. It happens when there is a need to provide a simulator DES notation, which wants to use the framework. It has to be able to simulate a DES model created using specified DES notation. In addition, it has to communicate with existing 3D simulator by provided interface.

Create DES model takes place when a model of new system is created. Engineer first creates a DES model as usual for used DES notation. Then, he delivers new extension for 3D simulation. Alternatively, he can use previously defined one. He also has to have 3D representation of models3D. Those are created using 3D modeling software. At last, he extends the DES model with information that connects it with a graphical description of the model. After that, DES model is ready to use in the framework.

Run simulation, along with included use cases, is used when the end user wants to run simulation on existing DES model in existing DES simulator. As shown of Figure 3-1 end user can start, stop simulation and pick model3D.

From concepts described previously, namely, animation, 3D model and geometry, we can create a domain model for the framework. We have to distinguish domain models during runtime and design.

The domain model during design can be derived from use cases, and user description made previously.

Framework can be divided into following components: 3D Simulator (Simulator3D on diagrams), DES Simulator, DES model with extension and user interface.

Simulator3D is a main part of the domain. It is independent of any DES notation. Its role in the framework is to manage 3D visualization part of simulation running on a designed model. As can bee seen on Figure 3-2 it depends on DES model 3dvExtensions (described below). In addition, it relies with a user interface. As for interaction with DES Simulator, it is made both ways, through API provided by Simulator3D.

DES simulator is provided by developer. It contains all functionalities necessary to run simulation on a given DES model. Using 3dvExtensions, it can interpret them and DES model behavior. By that, it can use Simulator3D to create and run 3D part of simulation.

Engineer is responsible for creation of DES model with 3dvExtensions. Extensions should be added to the DES model in a way, that DES model itself can be used by other simulators (not necessary 3D-visualisation). 3dvExtensions contains a graphical description (Geometry on Figure 3-2) used by Simulator3D and DES Simulator to connect both the 3D and DES parts of simulation. They are also used to set up interactions between simulators. 3dvExtensions are dependent of graphical description. However, they are independent of used DES notation and model. Graphical description contains a definition of animations, models3D and geometries used in simulation. All three rely on 3Dmodels created by third-part programs.

User interface is used by the end user to start, stop and interact (if possible) with simulation.

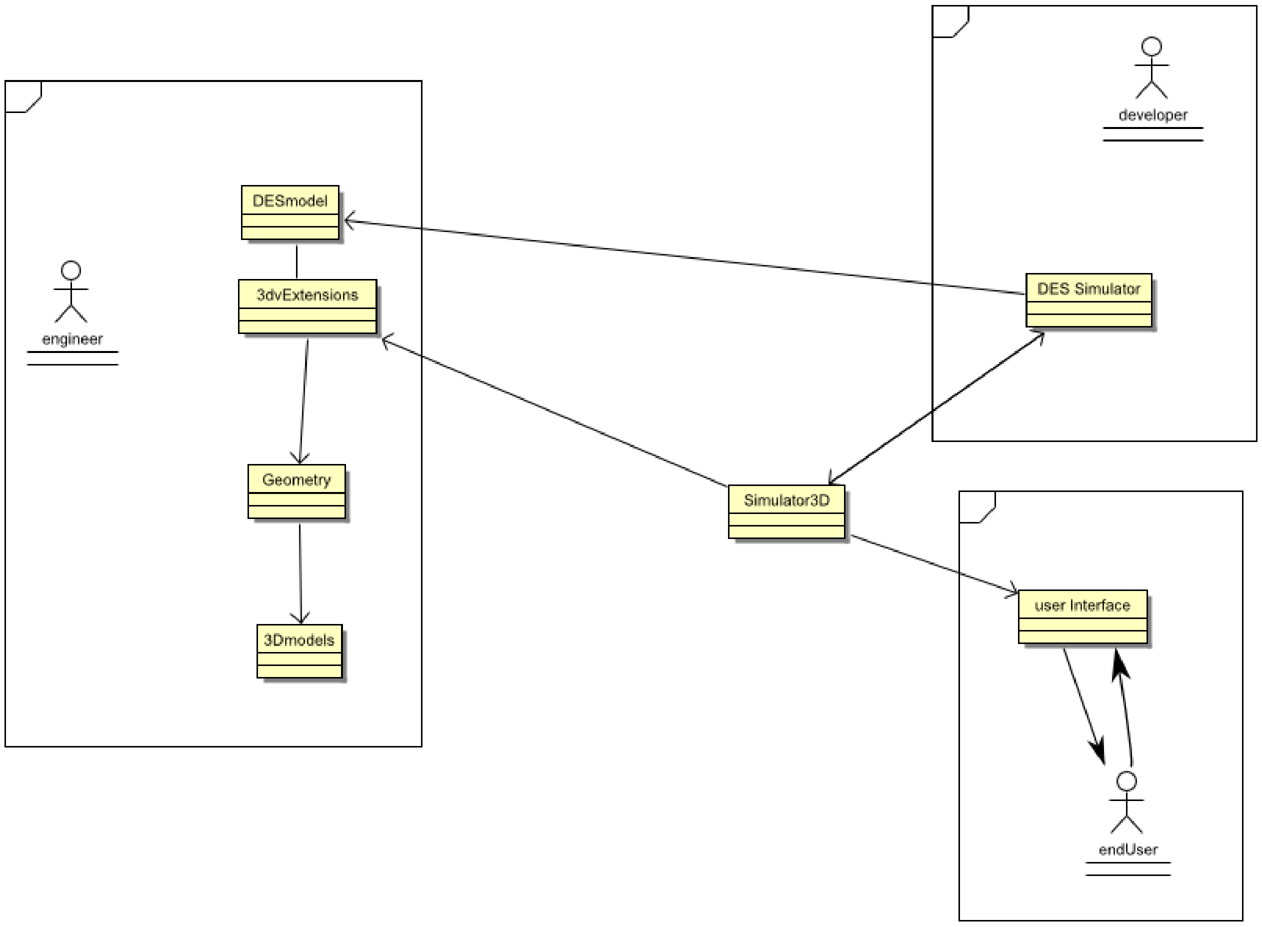


Figure 3-4 Design domain model

Domain model during runtime is based on concepts of Simulator3D, DES Simulator and animation, models3D and geometries. As can be seen on Figure 3-3 Simulator3D contains animations, models3D and geometries used by simulation. Additional animation has references to geometry and model3d instances. Simulator3D is itself own by DESSimulator.

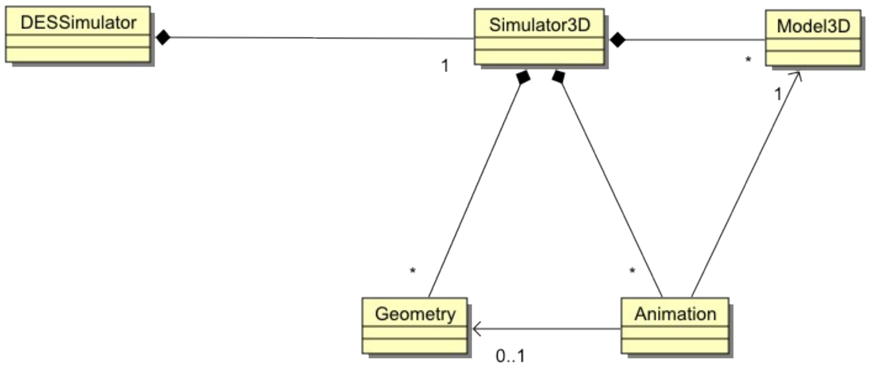


Figure 3-5 Runtime domain model of DES3dv framework

4. Design

Both design and runtime domain models, described in previous section can be used to design of prototype solution of the framework. Main parts of the framework are simulator3D and DES Simulator. They are used to run respectively, 3D and DES part of simulation. They communicate with themselves through an API defined by Simulator3D (explained later). Simulator3D does not change with the change of used DES notation for DES models. It is responsible for coordinating, running animations, react to end user actions, exchange messages with DES simulator. Additional it is used to create and dispose models3D and animations.

DES Simulator is subject of change in implementation for every DES notation. Its main responsibility is to coordinate simulation on the DES model, with 3D part of simulation. It can start/stop simulation, start/stop animations in the simulation and create/dispose models3D. It also makes a decision of simulation flow, based on notification received from simulator3D or animations.

- Interface between DES and 3D part of simulation is built from:
- startAnimation
- stopAnimation
- createModel
- disposeModel
- stopSimulation

in Simulator3D

- notificationChanged
- stopSimulation

in DESSimulator

StartAnimation allows DESSimulator to create and start specified animation in 3D part of simulation. Concrete animation is built from template assigned to that animation.

StopAnimation is used to stop and dispose animation that is no longer needed in simulation. A main reason for this is that animation finished its execution or flow of simulation in DES part has to terminate animation to proceed in simulation.

CreateModel is used when new instance of a specified model3D is needed in simulation. Since every model3D is assigned to animation, createModel is always fallowed by calling startAnimation. However, in some cases model3D have to be passed between animations. In that situation, model3D used in one animation is not disposed, but passed to next animation.

DisposeModel is called when model3D is no longer needed in simulation. It removes model3D from 3D part of simulation and releases all its resources.

StopSimulation allows simulation to stop and release all resources. It is defined in both simulator classes. Every one of them cleans a part of the simulation it is responsible for. It is called when end user request termination of simulation. Another case is when DES part of simulation reached its final stage of execution.

- From above interactions, fallowing sequences of action can be created:
- start animation
- stop animation
- stop simulation

Start animation begins with optional sequence with creating model3D. If there is a need to create new instance of model3D, DES simulator requests new one from Simulator3D. When a model is created, it is returned to DES Simulator. After that, DES Simulator requests new animation passing its name and moedel3D, either newly created or existing from previous animation. Simulator3D creates new animation from template and starts it. Then it returns that animation to DES simulator (as seen on Figure 4-1).

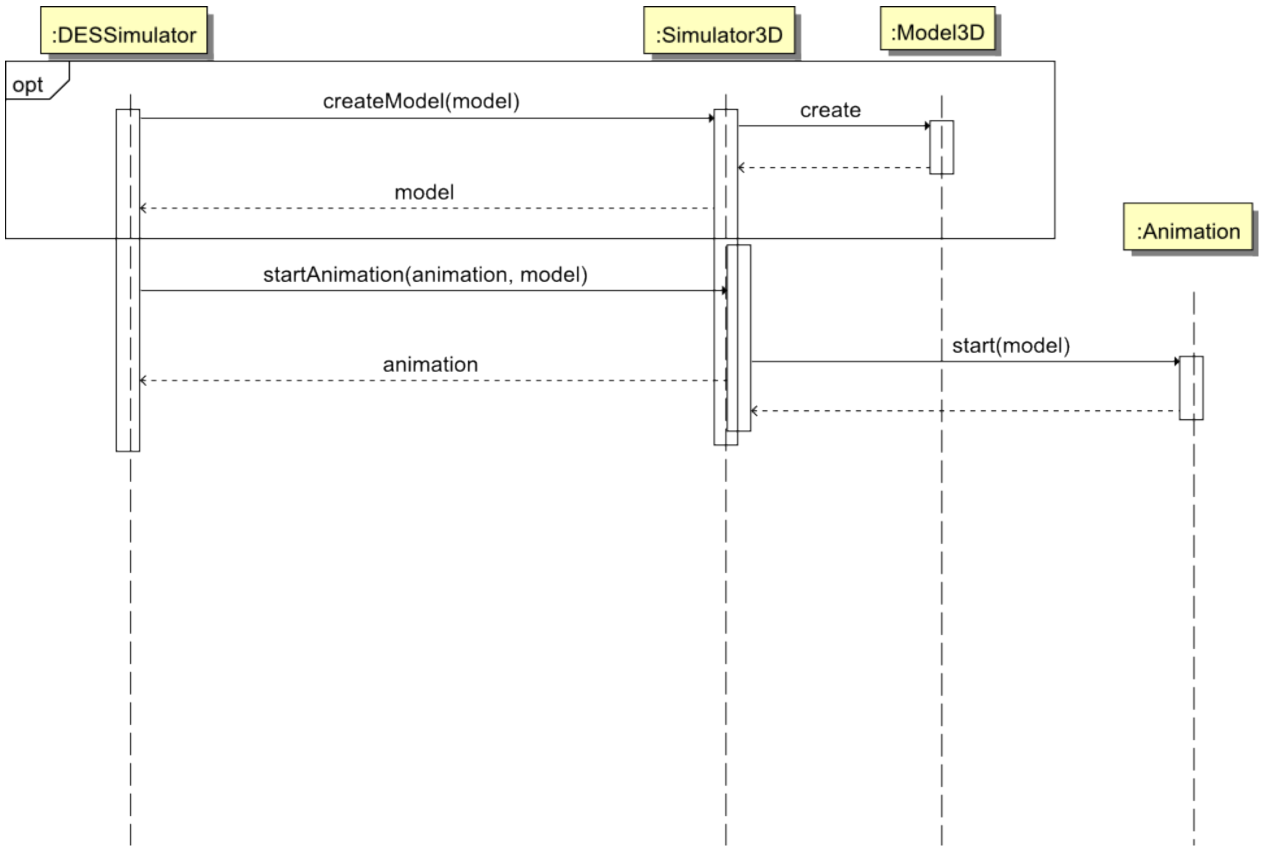


Figure 4-6 Start animation sequence diagram

Stop animation also has optional sequence, but with destroying model3D. If model3D is no longer needed in simulation, DES Simulator calls diposeModel from Simulator3D, passing model3D to destroy. Next, it calls stopAnimation passing animation to destroy (Figure 4-2).

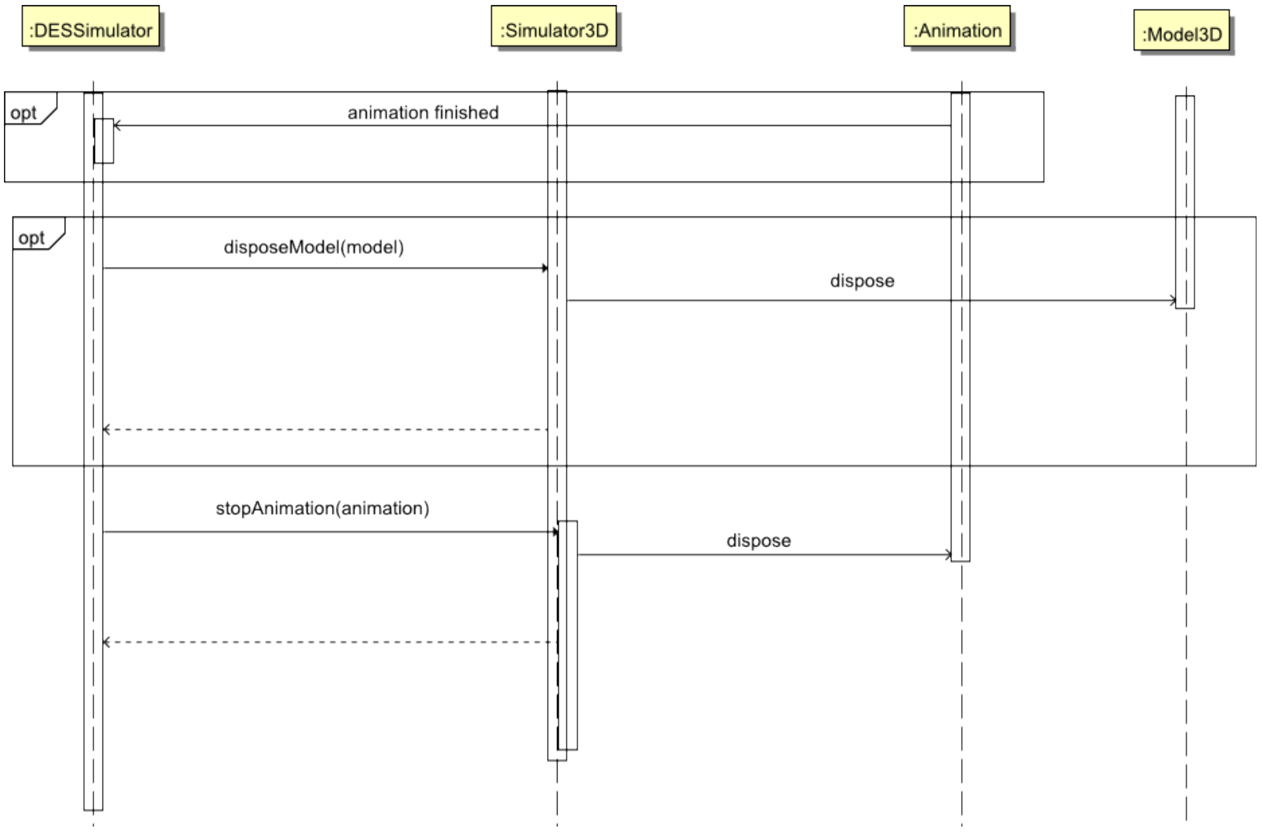


Figure 4-7 Stop animation sequence diagram

Last sequence, stop simulation can be executed by the end user of DES Simulator. In first case, stopSimulationEvent is passed to Simulator3D. It sends notification to DES Simulator. Then DES Simulator calls stopSimulation in Simulator3D. It cleans 3D part of simulation. After that DES, Simulator can releases all resources taken by DES simulation (Figure 4-3).

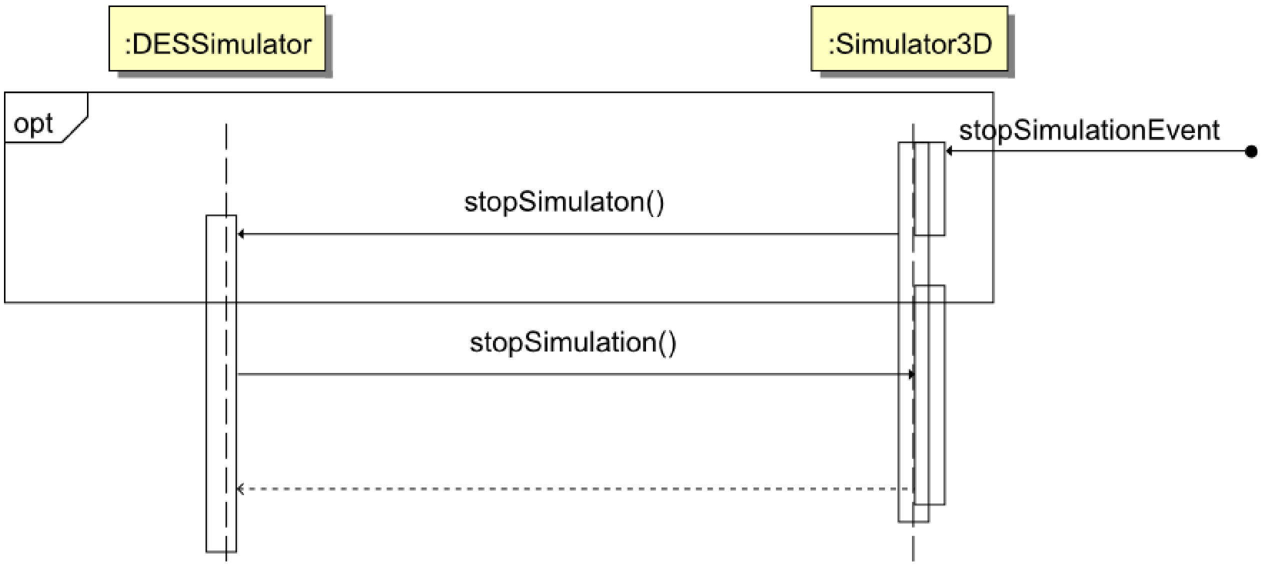


Figure 4-8 stop simulation sequence diagram

The framework is designed to work on an event-based approach. Both Simulator3D and DES Simulator for Petri net, provided in a prototype follow that approach completely. By that framework should perform better than implementation with loop checking state of simulation in regular time intervals.

As mentioned above, the core class diagram can be derived from domain models and use cases. Complete diagram can be seen on Figure 4-4. New classes in comparison to domain model are: Loader, Model3Dtemplate, AnimationTemplate and descendants, animation descendants.

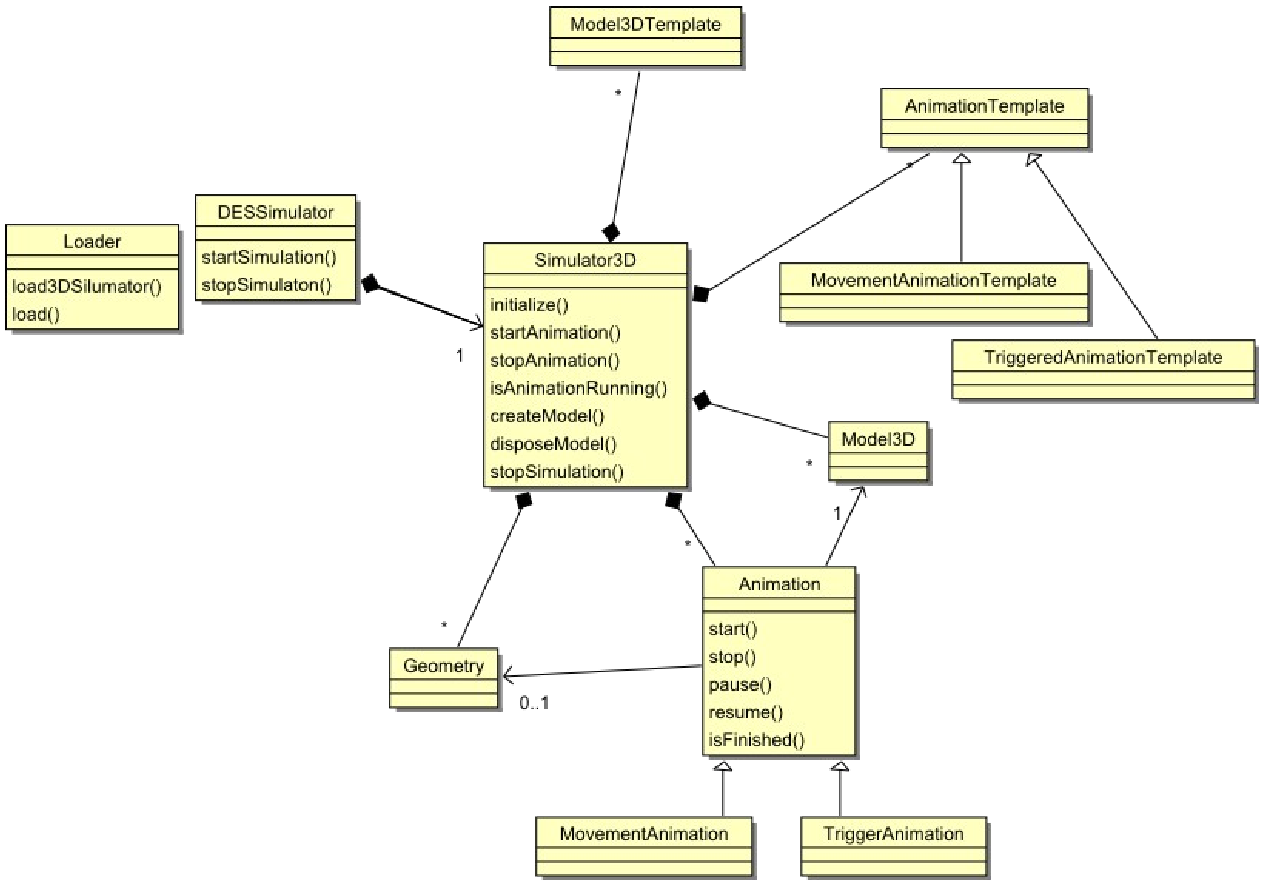


Figure 4-9 Class diagram for core system of framework

Loader class is responsible for loading both DES and graphical models used in simulation. It is abstract because loading to differ between different DES notations. In this abstract class, however a method load3Dsimulator is defined. It is used to load graphical informations about animations, models3D and geometries.

In current version, the framework supports two classes of animations: MovementAnimation and TriggerAnimation. First allow to define all kinds of movement along the path or rotate of model3D and their combination. By that it is a bit tricky to create such animation, but it reduces a number of animation classes (one per every combination of move and rotation otherwise). It will not be a problem when the framework will have an editor that will automatically create those animations. Second is used when a system is waiting for an event/trigger to proceed in simulating the model.

Model3D and AnimationTemplates are use to store information about models3D or animation that is defined in a graphic model. By that Simulator3D has quicker access to information needed to create new instances of this class. It is especially important when we read a file with 3D representation of medel3D. If it would be loaded every time model3D is to be created, it would greatly affect performance of the framework.

One of the ideas for this project was to eliminate any unnecessary information in the DES model. It was achieved by creating a model for all graphical information for the 3D simulation. It is built from several classes. All of them are just containers of information so none of them has any methods. Complete class diagram can be seen on Figure 4-5.

The main class is called Graphics. It contains both information created for definition of animations, models3D and geometries, as well as information required to set up 3D scene. Second are the definitions of background scene and lights used in the 3D scene.

Classes created because requirements of setting 3D scene.

BackgroudScene has only one attribute. It is a String that points to the file with a background scene used by the model.

ViewSetup class allows defining initial viewer position and orientation in the environment. Because of the used functions from Java3D, it is defined as three Point objects: userPos, lookAt and upVector. userPos holds the position of the viewer. LookAt defines a point on which user is looking at. upVactor allows finding the top of the user orientation. It also defines height and width of a simulation's window. Those are represented by integer numbers of pixels stored as int type.

Light is an abstract class used as a base for different types of light. It has one attribute of type Point, called color. It holds a color of the light represented by three real numbers in a range of 0 to 1.

AmbientLight defines a light that has no direction. It is used to simulate reflection in the environment. It has no additional attributes.

DirectionalLight is used to define a light that has a direction, but it is far from the scene. It has one attribute of type Point to hold the direction.

Classes created for framework:

Model is responsible for holding information about 3D model used by the application. It has three attributes: name, modelFile and pickable. First is a String used as an identifier. This allows differentiating between different models. Second points to the file with definition of the 3D object. It is also of a String type. The last one is a boolean used to decide whether the model can be pick or not by the user.

Geometry class allows defining any kind of transformation, except for the scaling of an object. It has a Point attribute that holds rotation axis. In addition, it contains lists of Points, Knots and Quats objects. All of them must have the same number of elements.

Animation is an abstract class for all possible kinds of animations in the model. It has two attributes of type String. First, called name is an identifier. Second is a name of a model.

MoveAnimation is a subclass of Animation. It contains two additional information. Time is used for setting how long, in milliseconds, should animation be running. It type is float. Other is a name of specific geometry object. By using complex representation of Geometry, it is possible to cover different types of movements' animations by only this class. It covers simple moving along the line, as well as complex path movement. It is also possible to use it as a rotation animation and combination of all of them. I decided to implement it that way, because it reduces number of class required if use different representation. It requires more data to input for geometry used in animation, but it will not be a problem when all data will be generated, not written manually.

TriggeredAnimation is used for holding simulation of DES at a given point of simulation. It has an attribute of Point for holding the position of the model used by this animation. It is required since model class does not contain such information. This comes from the fact that MovementAnimation changes the position if used model, so it is useless to kept position of the model by itself.

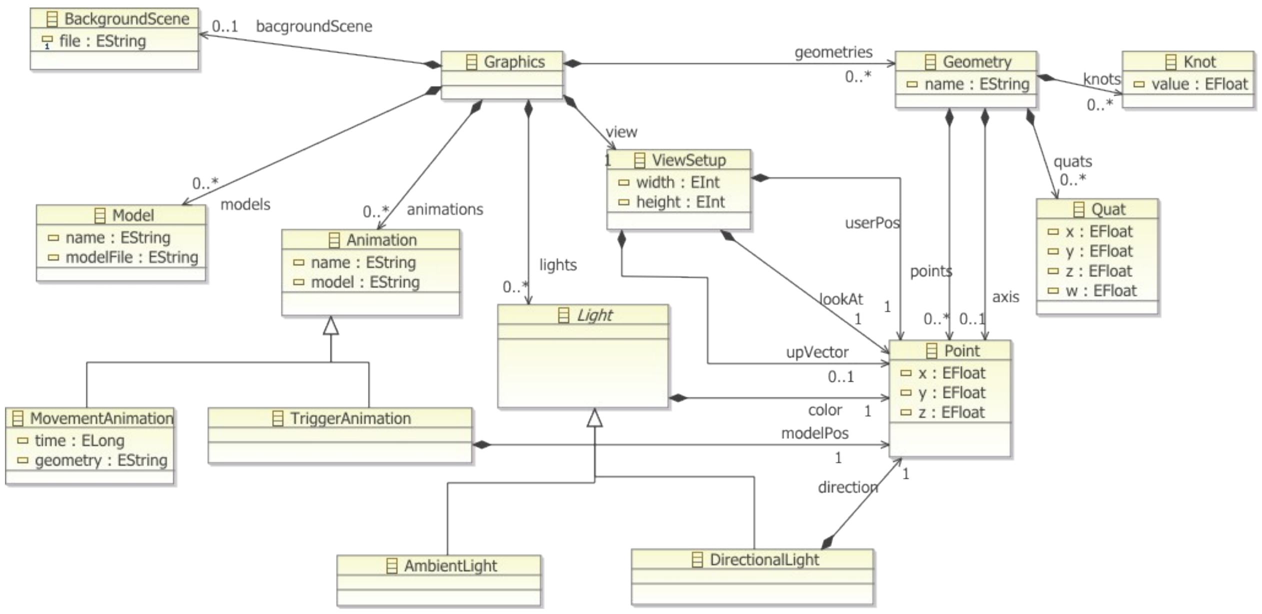


Figure 4-10 Class diagram for graphical model

This new type of Petri Net defines extensions to Petri Net type from ePNK. They are needed to store information required to run 3D simulation.

vNet inherits from PetriNetType. It purposes is to define a new petri net type. By that ePNK editor is aware of existence of this type.

Page inherits from Page defined in ePNK. It has a label that holds a file name pointing to Graphics model. It also has a list of labels, holding geometries' names used in the model.

Place contains three additional labels. One holds a name of animation attached to this place. Second states if that animation should be run to its ends or can be stopped by the simulator. Last one is used to place tokens in that place. None of those labels is required but setting synchronization makes sense only when the place has defined animation.

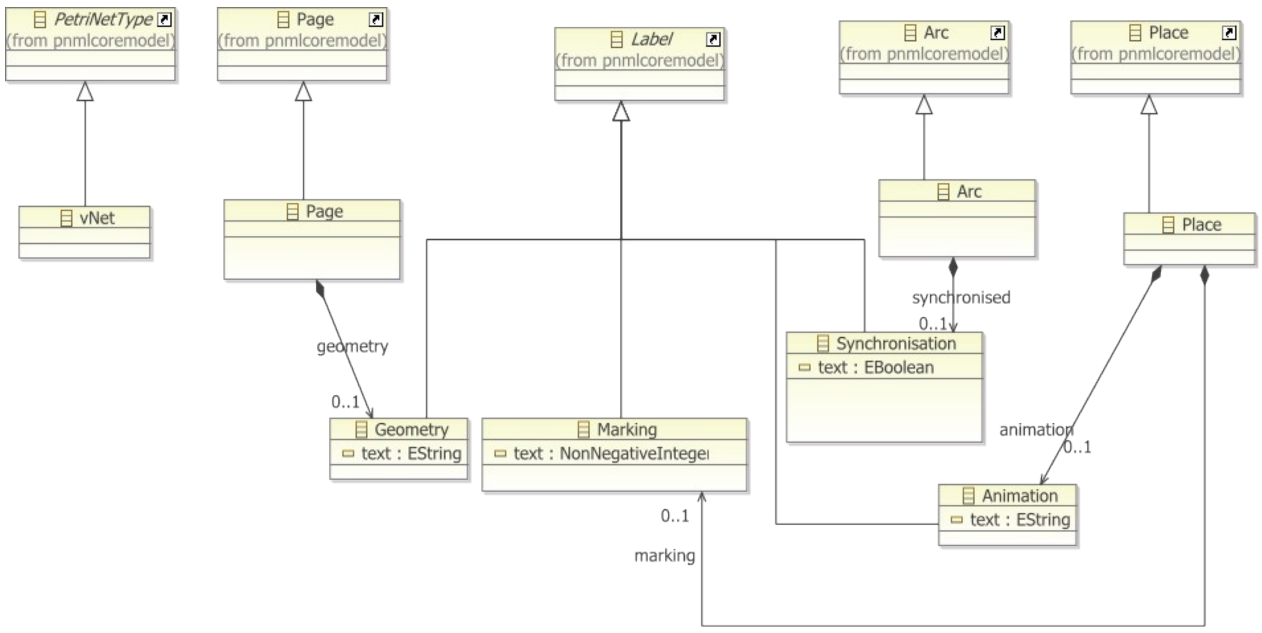


Figure 4-11 Class diagram for vNet

As part of the prototype concrete implementation of Petri Net simulator was created. This part of the framework extends 3D Visualization system and ePNK to provide implementation of Petri Net simulator used as a validation tool for 3D visualization. Since ePNK does not provide a runtime model of defined Petri Net, it is necessary to create one. However, because Petri Net model and its runtime representation are close related there is no need to define runtime representation for whole model.

Place is used as a runtime representation of Place from ePNK Petri Net. It holds a number of tokens in that place in an integer number. A string variable is used to keep the name of the animation template used by Simulator3D for animation attached to that place. Additionally, it has three lists. Models keep the identifiers of models used by running animations started in that place. runningAnimation holds identifiers of animation created and running in that place. finishedAnimation stores animations that have already finished but are not yet stopped by the simulation. To ease token manipulations this class defines two methods responsible for adding and removing tokens from that place.

PNLoader extends Loader class from the 3D simulator project. It implements the load method in order to load Petri Net model. It adds an additional method to initialize its instance concerning Petri Net object.

PNSimulator extends DESSimulator from the 3D simulator project. It provides the implementation of startSimulation method. Simulation is started by calling checkPlace for every place in the simulation and checkTransition for every transition. checkPlace is called as many times for a place as it has tokens. In addition, PNSimulator implements changedNotification from Listener interface. This method checks the value of notification and calls appropriate method base on it. Their description can be found in implementation section. Additionally, it defines three methods to run simulation on Petri Net. fireTransition takes a transition object and performs operations related to firing that transition. It assumes that all conditions for fire are met. It clean ups after finished animations, pass models that can be reused to another places. It also moves tokens from incoming to outgoing places. It is also responsible for calling checkPlace for all places connected with this transition. checkPlace check if it is possible to run animation attached to that place. checkTransition is responsible for checking if passed transition can be fired. It can be if all synchronized animation in places connected to it are finished, and every place has at least one token. If above criteria are fulfilled it calls fireTrinsition on that transition.

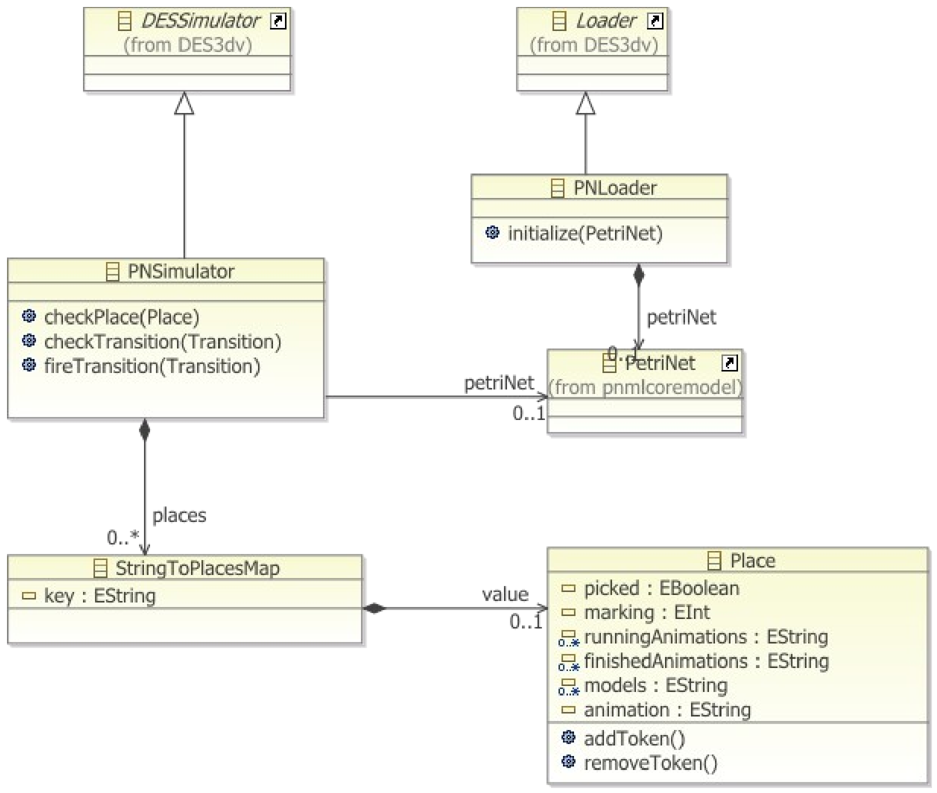


Figure 4-12 Class diagram for implementation of DES Simulator for Petri Nets

5. Tools and technology

Nowadays, writing a complete system from scratch would be tedious work, in many parts of the application. The purpose of the Software Engineering is to eliminate, or limit as possible, the need to write code of the application by hand. That is why I have used set of tools/technologies/libraries in my implementation.

During choosing the platform and tools for implementation the prototype, I considered mostly quality and my knowledge of proposed technologies/tools. It would be very risky to try to learn a completely new language during writing the thesis. Having that in mind, I have chosen Java language and Eclipse IDE. Both proved themselves as useful language and IDE. In addition, they are well documented and have large community. That assures that I have always access to a lot of materials and forums.

Eclipse IDE, because of its modular nature, encourage community to creating an extension to original IDE. Among others, an Eclipse Modeling Framework (EMF) can be found. It allows software engineers to create models of application in UML, and later generate Java's code from it. It speeds up creating an application drastically. What is more, the amount of code that needs to written by an engineer is reduced, in some cases to none.

Choosing the 3D library was the most difficult decision to make. Number of available products, their complexity and level of graphical abstraction are very large. At one point, we can find libraries such as JOGL, which is a wrapper around OpenGL. It works on a low level of abstraction. This allows having a fine control of animations, but requires more work and control over graphics. It was my first choice for library. Main argument for it was my knowledge of OpenGL. It would require least work to use it in the application. However, after few tests of it, I realized that there would be too many things that I would have to implement by myself. To give an example, the library has no support for loading models. Another reason is that major version is currently developed. So there is no guaranty that use of current, stable realize of it will be working with next version.

Another option is a high- level, 3D library. Many of them exist as a gaming engine, for example, jMonkey Engine. By that user is provided with most of the functionality he needs. Nevertheless, using that kind of library usually requires acquiring knowledge of it structure, and sometimes its own scripting language. Using any of them could lead to a situation that I ended up reading documentation instead of creating simple function.

My final choice is Java3D. It is a high level, but only graphical library. It has an object-oriented approach to creating a 3D scene. That allows simple, one-to-one relation between model in the simulator and its graphical representation. Moreover, it has interface for reading 3D models in various formats. The main disadvantage I found when working with it is that official documentation is outdated. There are also not many external teaching materials. However, existing ones are in good quality and allows creating a prototype of the application.

Feature	JOGL	Java3D	jMonkey Engine
Low/high level	low	high	high (game engine)
Support for loading 3D models	No	Yes	Yes
Collision detection	No	Yes	Yes
Need to learn new language	No	No	Yes

Figure 5-13 Comparision of discussed 3D libraries

The last tool used in my implementation is ePNK. It is built using EMF, platform for creating Petri Nets. It allows a user to define new types of Petri Nets from the one provided by ePNK. Any additional information can be attached to any of Petri Net component. It also supports reading/writing models in PNML format, providing that it met the requirements of PNML standard.

PNML is an xml-based format, that allows Petri Net application exchange Petri Net models [3].

6. Implementation

In order to fallow the design, as well as for the Eclipse/EMF requirements prototype of the framework is applied into fallowing EMF projects:

- DES3dv
- DES3dv.Graphics
- DES3dv.vNet
- DES3dv.actions

DES3dv is a core of the runtime system of the framework. It contains Simulator 3D, and all classes related to 3D part of simulation. In addition, abstract classes for DES simulator and Loader are defined her as an extension points in EMF. That allows extending functionality of that project from another one. Its class diagram is extended in few points in comparison to the one discussed in design section. It is due to implementation requirements of EMF. Complete diagram can be seen in Appendix.

DES3dv.Graphics contains all classes needed to implement a model for graphical information used by the framework.

DES3dv.vNet defines new Petri Net type used by ePNK to create extensions that allow Petri Net model to be run in the framework.

Additional to those DES3dv.actions project was created. It contains action to start simulation from ePNK editor.

6.1 Resoling of collision in simulation

Collisions resolving provided by the prototype is rather simple and narrowed. It uses the Java3D behaviors built with wake up conditions collision enters and collision ends for every movement animation. When animation starts to collide with some object in 3d part of simulation, it pauses its movement and notifies simulator3D of it. Simulator3D tries to resolve situation. Currently it only deals in situation when it is possible to determine which object is in front during the collision. It check if objects are moving on line or if their make a small angle between their trajectories. If Simulator3D can decide, which one is in front of other it calls resume on that one. When collision ends, animation is resumed by itself, since there are no reasons for it to be paused anymore. In case when Simulator3D is not able to decide, in current implementation, simulation will be frozen in a state with colliding objects.

6.2 Implementation of MovementAnimation class

Movement animation class is the only part of the framework that does not completely follow the event-base approach in running. For prototype implementation, it contains a Jave3D behavior that is checking if animation is finished every 15 frames. That solution could be improved changed in next versions of the framework.

7. Validation

For validation of concepts used to design the framework, one Petri Net model and graphics model were created. They present part of the sweet bunny factory. It has two kinds of bunnies from previous (not covered by model) parts of a production process. Model presents part of the production process where bunnies are wrapped by the same kind of package. They move along shared path in the factory. That way, it is possible to see collision resolving provided by the prototype. Complete model can be seen on Figures 7-1 and 7-2.

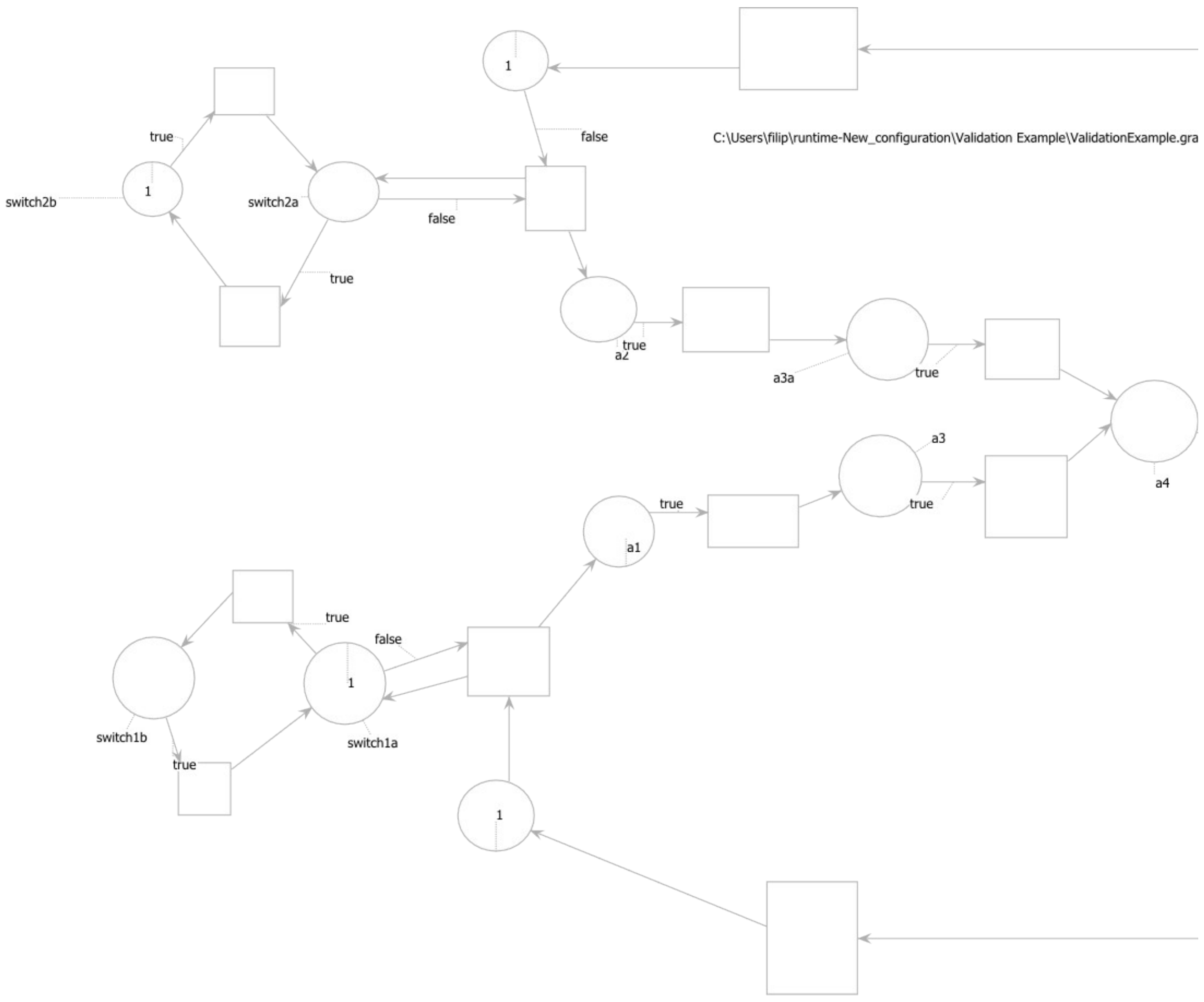


Figure 7-14 Bunnies factory Petri Net model part1

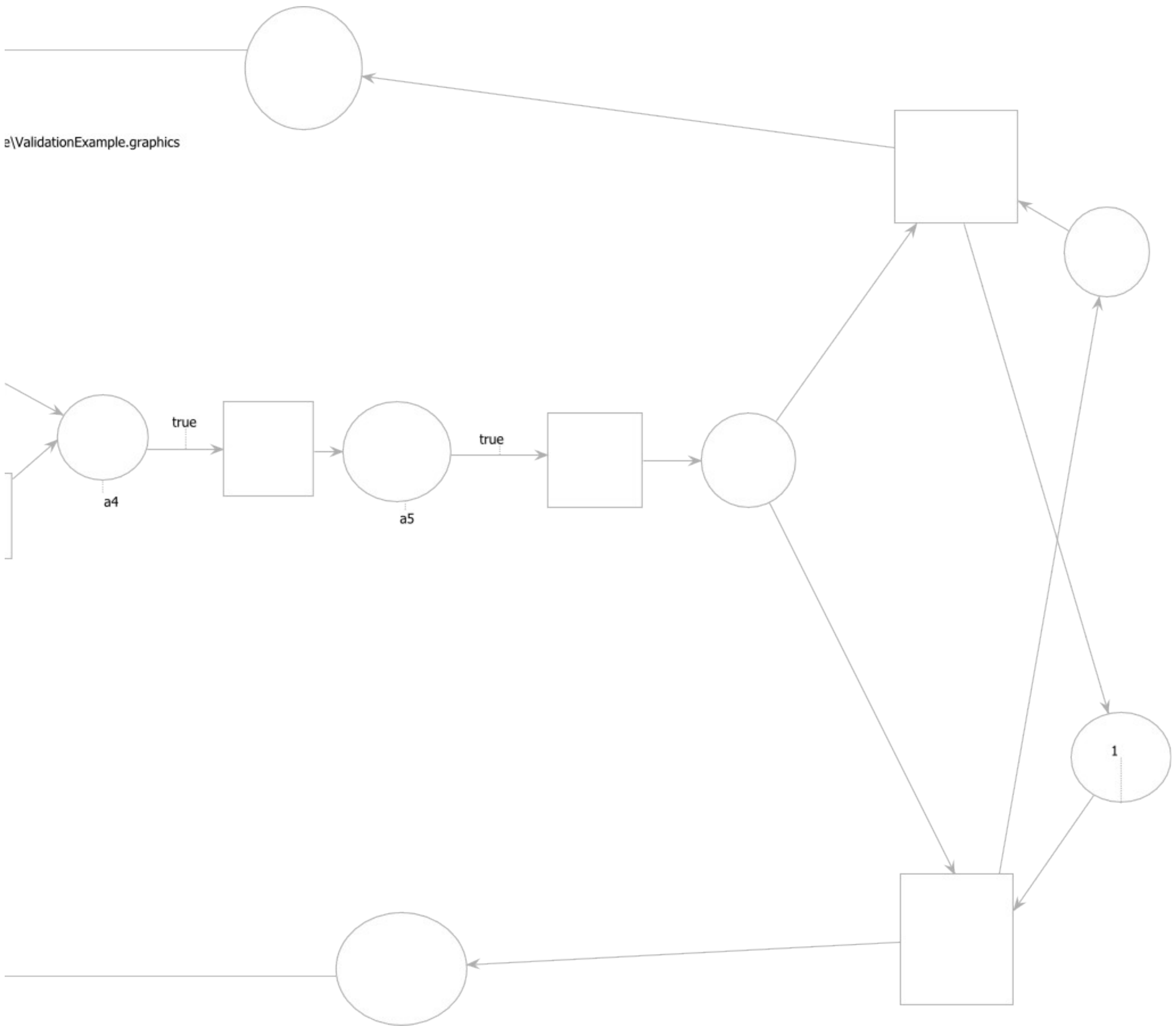


Figure 7-15 Bunnies factory Petri Net model part2

On Figures 7-3, 7-4, 7-5 different stages of simulation on above model can be seen.

Figure 7-3 presents simulation of one bunny type before packing.

Figure 7-4 presents the same simulation few seconds later, when bunny was packed (different model3D)

Figure 7-5 presents two different bunnies represented by different colors.

Bunnies used in simulation are using Stanford bunny model [4].

Sometimes for unknown reasons application throws exception related to Java3D behavior class. It is possible that it is an error in the prototype. However it could also be error in Java3D itself. As for now, I cannot state that for sure.

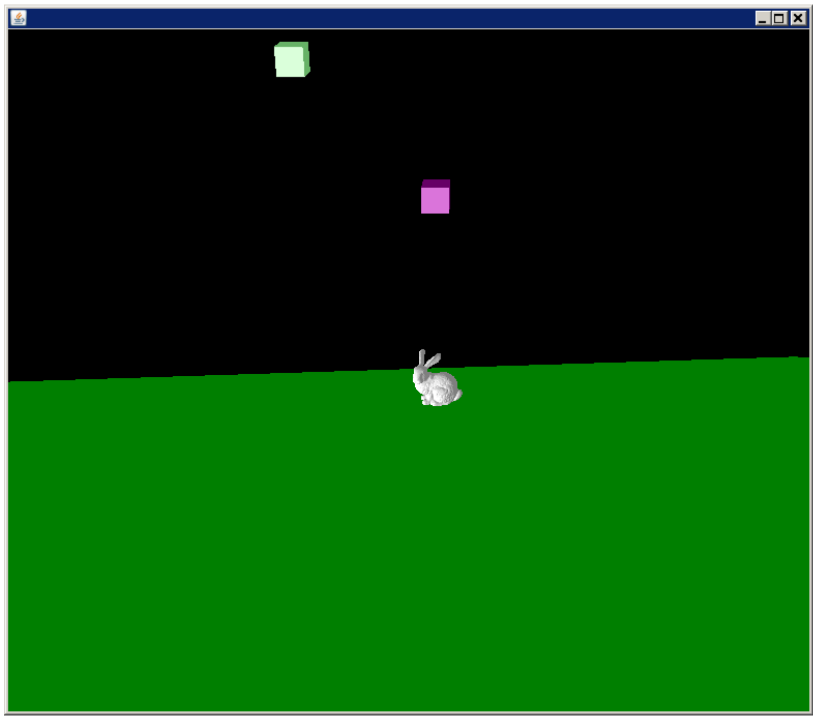


Figure 7-16 Simulation with one bunny type

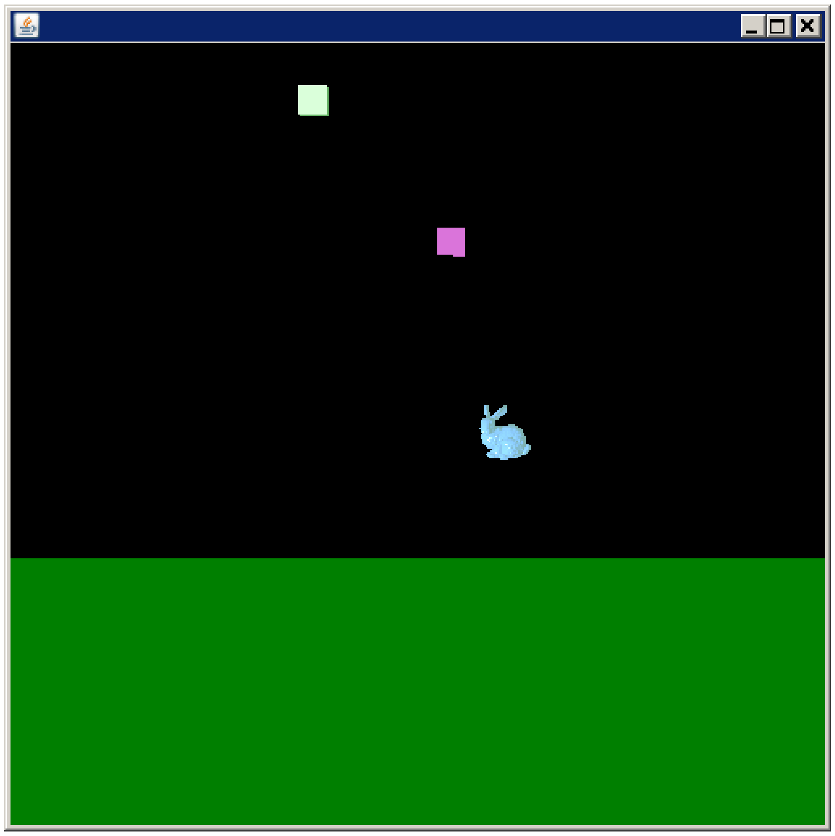


Figure 7-17 Simulation with one bunny type- later stage

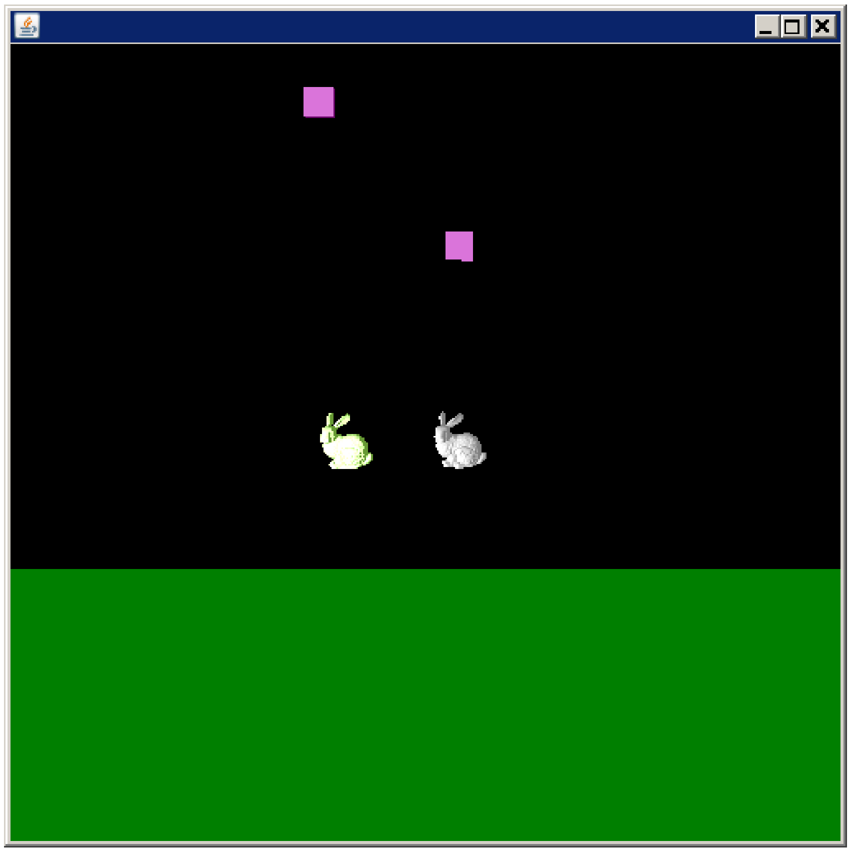


Figure 7-18 Simulation of two bunnies types

8. Limits/Potential extensions

As for prototype version, the system has currently few limitations. They might and should be eliminated in the future version of the application:

- Only two types of animation: movement and trigger
- Models are assumed to contain only one group of polygons
- Collision detection resolve only case when object hits another from behind
- No support of many animations for one place in Petri Net simulator
- Not optimal use of Java3D (requires greater knowledge of library)

Additional potential extensions could be made in next versions of the framework:

- Add support for GLSL shaders

9. Conclusions

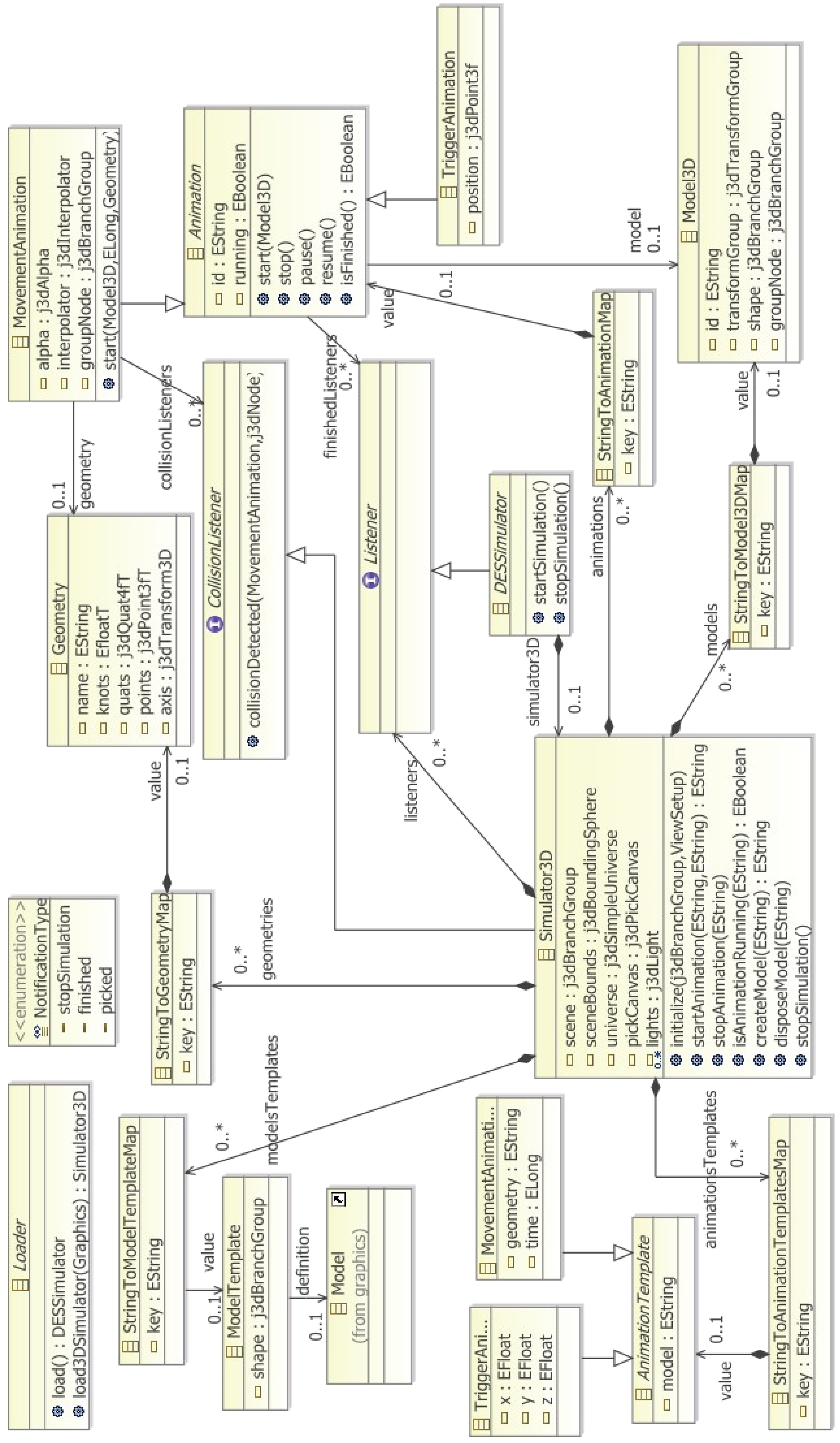
Presented paper provides concepts for creating a 3D-visualization system for DES models. In addition, the design and implementation of one concrete example of DES were made during the thesis. It would be better to provide more implementation of different DES notation, but it was impossible because of the time limitation. Nevertheless, from the design of the framework can be seen that it is fairly easy to extend the framework with new DES Simulator. Given prototype proofs all concepts given in section 2. It also proofs that framework is able to provide a 3D-visualization for DES notation, in that case Petri Net.

10. References:

1. E. Kindler, C. Páles 3D-Visualisation of petri Net Models: Concept and Realization
2. http://www2.imm.dtu.dk/~eki/projects/ePNK/
3. http://www.pnml.org/
4. http://graphics.stanford.edu/data/3Dscanrep/

11. Appendix

11.1 Ecore diagram for DES3dv project


MovementAnimation
alpha : j3dAlpha
interpolator : j3dInterpolator
groupNode : j3dBranchGroup
start(Model3D,ELong,Geometry)
Animation
id : EString
running : EBoolean
start(Model3D)
stop()
pause()
resume()
isFinished() : EBoolean
TriggerAnimation
position : j3dPoint3f
Model3D
id : EString
transformGroup : j3dTransformGroup
shape : j3dBranchGroup
groupNode : j3dBranchGroup
model
0..1
value
0..1
StringToAnimationMap
key : EString
finishedListeners
0..*
collisionListeners
0..*
CollisionListener
collisionDetected(MovementAnimation,j3dNode)
Listener
geometry
0..1
Geometry
name : EString
knots : EfloatT
quats : j3dQuat4fT
points : j3dPoint3fT
axis : j3dTransform3D
DESSimulator
startSimulation()
stopSimulation()
animations
0..*
StringToModel3DMap
key : EString
models
0..*
simulator3D
0..1
listeners
0..*
Simulator3D
scene : j3dBranchGroup
sceneBounds : j3dBoundingSphere
universe : j3dSimpleUniverse
pickCanvas : j3dPickCanvas
lights : j3dLight
initialize(j3dBranchGroup,ViewSetup)
startAnimation(EString,EString) : EString
stopAnimation(EString)
isAnimationRunning(EString) : EBoolean
createModel(EString) : EString
disposeModel(EString)
stopSimulation()
«enumeration»
NotificationType
stopSimulation
finished
picked
StringToGeometryMap
key : EString
geometries
0..*
Loader
load() : DESSimulator
load3DSimulator(Graphics) : Simulator3D
StringToModelTemplateMap
key : EString
modelsTemplates
0..*
ModelTemplate
shape : j3dBranchGroup
definition
0..1
Model
(from graphics)
MovementAnimati...
geometry : EString
time : ELong
TriggerAni...
x : EFloat
y : EFloat
z : EFloat
AnimationTemplate
model : EString
animationsTemplates
0..*
StringToAnimationTemplatesMap
key : EString