3D-Visualization of Petri Net Models: A concept

Ekkart Kindler and Csaba Páles

University of Paderborn, Department of Computer Science
`kindler|cpales@upb.de`

Abstract. We present a simple concept for a 3D-visualization of systems that are modelled as a Petri net. To this end, the Petri net is equipped with some information on the physical objects corresponding to the tokens on the places.

1 Introduction

Petri nets are a well-accepted formalism for modelling concurrent and distributed systems in various application areas: Workflow management, embedded systems, production systems, and traffic control are but a few examples. The main advantages of Petri nets are their graphical notation, their simple semantics, and the rich theory for analyzing their behaviour.

In spite of their graphical nature, getting an understanding of a complex system just from studying the Petri net model itself is quite hard – if not impossible. In particular, this applies to experts from some application area who, typically, are not experts in Petri nets. 'Playing the token-game' is not enough for understanding the behaviour of a complex system. Using suggestive icons for transitions and places in order to indicate the corresponding action or document in the application area is but a poor solution.

Therefore, there have been different approaches that try to visualize the behaviour of a Petri net in a way understandable for experts in the application area. At best, there will be an animation of the model using icons and graphical features from the corresponding application area. *ExSpect* [9], for example, uses the concept of *dashboards* in order to visualize the dynamic behaviour of a system in a way that is familiar to the experts in the application area (e. g. by using flow meters, flashing lights, etc. as used in typical control panels). It is even possible to interact with the simulation via this dashboard. Another example is the Mimic library of Design/CPN [4, 8], which allows a Design/CPN simulation to manipulate graphical objects, and the user can interact with the simulation via these graphical elements. This way, one can get a good impression of the 'look and feel' of the final product. A good example is the model of a mobile phone [6]. Another approach for visualizing Petri nets is based on graph transformations and their animation: GenGED [1, 2]

In the *PNVis* project, we take the next step: The graphical objects manipulated by the simulation are no longer considered to be artifacts for visualizing information; rather we consider them as a part of the system. Actually, they are considered as the *physical part* of the system. Though simple, this step has several benefits: First, it makes the interaction between the control system and the physical world explicit. Second, the physical part can be used for a realistic 3D-visualization of the dynamic behaviour by using the shape and the dynamic properties of the physical components. Third, the properties of the physical objects can now be used for analysis and verification purposes. For example, we can exploit the fact that two physical components cannot be at the same place at a time.

In this paper, we discuss some first ideas on how to enrich a Petri net model with information on physical objects, and we present a tool which uses this information for a 3D-visualization of the physical objects. Here, the focus is on those aspects of the physical objects that are necessary for visualization: basically the shape of the objects. Currently, the tool is restricted to low-level Petri nets, P/T-Systems to be precise. The PNVis project, however, has a much wider scope. For example, we would like to use some physical properties of the objects such as their weight for analysis purposes. Moreover, PNVis will support high-level Petri nets as a more powerful modelling mechanism. And PNVis will provide concepts for constructing a system from components in a structured way.

PEP [7] was one of the first Petri Net tools that came up with a 3D-visualization of Petri net models: *SimPep* [5]. Basically, *SimPep* triggers animations in a VRML model while simulating the underlying Petri net. But, this simulation imposes a sever restriction on the animations: There is only one animation at a time; concurrent animations of independently moving objects are impossible. In this paper, we will present concepts that allow us to have concurrent animations of independent objects.

Of course, there are many toolkits and frameworks for building 3D-visualizations of systems, which cannot be discussed in this paper. Most of them are much more powerful than PNVis and have much more features. PNVis is not intended to compete with such toolkits. The focus of PNVis is on the Petri net model, which can be easily equipped with some more information in order to get a simple 3D-visualization. What is more, no programming or knowledge of 3D-visualization techniques are needed for adding this information.

2 Concepts

In order to animate the behaviour of a Petri net in a 3D-visualization, the Petri net must be equipped with some information on the physical objects. Moreover, the behaviour of the physical objects must be related to the dynamic behaviour of the Petri net. In the follwoing, we will discuss how to add this information to a Petri net.

Geometry, shapes and animation functions. In a first step, we distinguish those places of a Petri net that correspond to physical objects. We call these *anima-*

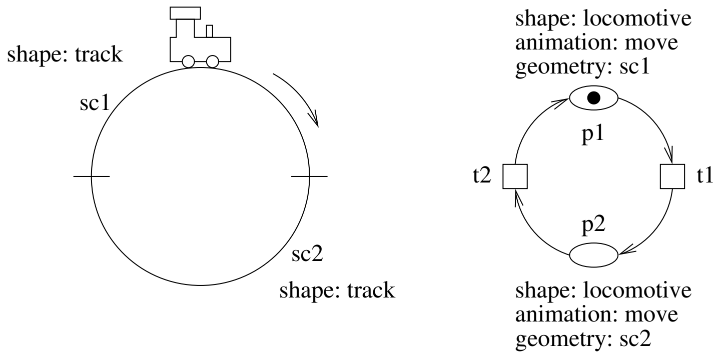

Fig. 1. A toy-train

tion places. The idea is that each token on an animation[1] place corresponds to a physical object with its individual appearance and behaviour. In order to animate a physical object, we need two pieces of information: its shape and its behaviour.

Defining the *shape* of the object is easy: Each animation place is associated with a *3D-model* (e. g. a VRML file) that defines the shape of all tokens on this place[2]. Defining the *behaviour* of an object is similar: Each animation place is associated with an *animation function*, where the animation function is constructed from some predefined animation functions. When a token is produced on an animation place, an object with the corresponding shape appears and behaves according to the animation function. For example, the object could *move* along a predefined line, the object could *rotate*, or the object could simply *appear* at some position.

In order to illustrate these concepts, let us consider a simple example: a toy-train. Figure 1 shows the layout of a toy-train, which consists of two semicircle tracks *sc1* and *sc2*, which are composed to a full circle. We call this layout the underlying *geometry*. For defining such a geometry, there is a set of predefined geometrical objects such as lines, circle segments, and Beziér curves. In our example, there is one toy-locomotive moving clockwise on this circle. The right-hand side of Fig. 1 shows the corresponding Petri net model, where both places *p1* and *p2* are animation places. In this example, the correspondence between the Petri

[1] Actually, these places could be called *physical* places because the tokens correspond to physical objects. Since we are concentrating on the visualization of the physical models in this paper, we call them *animation* places.

[2] Note that this means that all tokens on the same animation place have the same shape. This corresponds to the fact that we use low-level Petri nets in which the different tokens on a place cannot be distinguished. When the concept is lifted to high-level Petri nets, we can have objects of different shape on the same place: the shape can depend on the value of the token. Moreover, we will discuss a way of having tokens with different shapes on the same place even for low-level Petri nets later in this section.

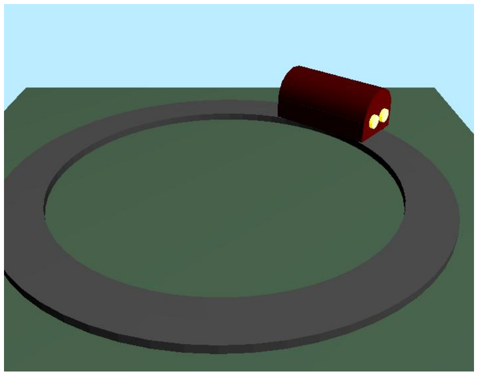

Fig. 2. Screenshot of an animation of the toy-train

net model and the physical model is clear from the similar layout. Formally, this correspondence is defined by annotating each place with a reference to the corresponding element in the geometry. As expected, the annotation *shape* defines the shape of the objects. In our example, it is a locomotive for both places[3], where the details of the definition of the shape will be discussed in Sect. 3. For now, you can think of it as the reference to some VRML model of a toy locomotive. The annotation *animation* defines the behaviour of the object, which is started when a token is added to the place. In our example, it is a *move* animation. Note that, without additional parameters, each animation function refers to the geometry object corresponding to that place. So, a locomotive corresponding to a token on place *p1* moves on track *sc1*, and a locomotive corresponding to a token on place *p2* moves on track *sc2*.

In order to make our example complete, we must also give some information on how to visualize the geometry objects themselves. To this end, each geometry object can have an annotation *shape*, too. In our example, the semicircles should be visualized as tracks (see Sect. 3 for details). Once we have provided this information, we can start a 3D-visualization of this system. Figure 2 shows a screen-shot of the animation of our example, where the locomotive on place *p1* has almost reached the end of its move animation on *sc1*.

Object identities. Up to now, we assumed that the shapes corresponding to the tokens on the two places *p1* and *p2* are completely independent of each other. When transition *t1* fires, the object corresponding to the token on place *p1* is deleted and a new object corresponding to the token on place *p2* is created and the move animation is started. Clearly, this is not what happens in reality. In reality, the same physical object, the locomotive, moves from track *sc1* to track

[3] In principle, we could have different shapes for both places. In that case, the locomotive would appear to change its shape when it changes the track. Using the same shape gives the impression that the same locomotive is moving on. We will see soon that there is an even better way to keep the *identity* of the locomotive.

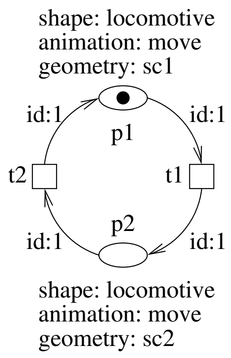

Fig. 3. The toy-train equipped with identities

sc2. In order to keep the identity of a physical object when a token is moved from one place to another, we equip the arcs of the Petri net with annotations of the form *id:n*, where n is some identifier[4]. We call n the *identity* of that arc. By assigning the same identity to an in-coming arc and an out-going arc of a transition, we express that the corresponding object is moved between those two places. In order not to clone a physical object, we require that there is a one-to-one *correspondence* between the identities of the in-coming and out-going arcs of a transition. Therefore, we require for each transition that each identity occurs exactly once in all in-coming arcs and exactly once in all out-going arcs. Figure 3 shows the toy-train example equipped with such identities[5].

Animation results. Next, we consider the relation of the behaviour of the Petri net and the animations of the objects corresponding to the tokens in more detail. When a token is added to an animation place by firing a transition, the animation for the corresponding object is started. But, what will happen, if a token is removed before the animation is terminated? In our example, this does not make much sense – the locomotive would jump from some intermediate position of the track to the start of the next track. Assuming that transition firing does not take any time, this behaviour is physically impossible. But, there are other examples in which a transitions could fire while an animation is running. Whether a transition may remove or must not remove a token with a corresponding animation running must be explicitly defined in the Petri net model. When the transition should wait until the animation of a token has terminated before removing it, we add the annotation *result:* {..} to the corresponding arc[6]. If there is no such

[4] In the current implementation, it must be some natural number.

[5] Both transitions have only one in-coming and out-going arc. Therefore, the examples is not very interesting. We will see more interesting examples, soon.

[6] Actually, an animation function has a return value, and the annotation *result* says for which return value of the animation the corresponding transition may fire. The set {..} stands for all possible return values. So, *return:* {..} means that the animation must be terminated – with any return value.

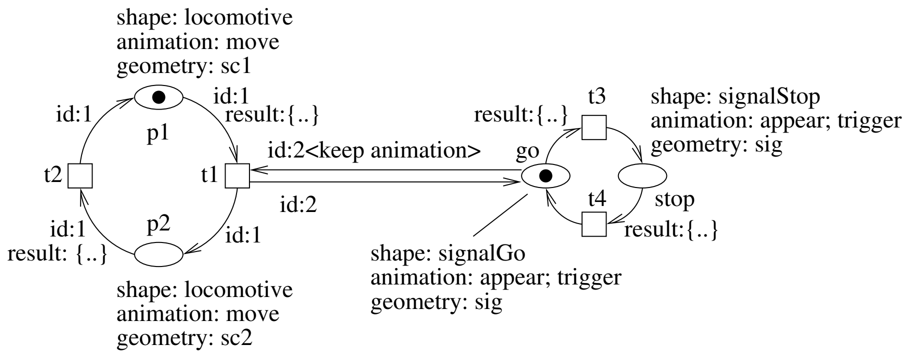

Fig. 4. A toy-train with a signal

annotation at the arc, the transition need not wait until the animation of the corresponding object is terminated – when fired, the transition simply stops the animation[7].

In order to illustrate these new concepts, we extend our example. We assume that there is a signal at the end of track *sc1*. When the signal is in state *stop*, the locomotive stops at the end of track *sc1*; when the signal is in state *go*, the locomotive may enter track *sc2*. In order to have a position for the signal in the layout, the geometry is extended by a point *sig* at the end of semicircle *sc1*. Figure 4 shows the Petri net model of this extended system. The two places *p1* and *p2* as well as the transitions *t1* and *t2* are the same as before. The arcs are equipped with identities in order to keep the same object, the locomotive, on the tracks. The annotation *result:*{..} guarantees that the transitions wait until the move animation of the locomotive has come to an end (i. e. the locomotive has reached the end of the track). Next we consider the signal: The two states of the signal are represented by the places *stop* and *go*. The object corresponding to a token on place *stop* is a signal with its red light on: *SignalRed*. The object corresponding to a token on place *go* is a signal with its green light on: *SignalRed*. These objects will appear at the point sig of the geometry (somewhere at the end of *sc1*). Due to the loop between place *go* and transition *t1*, transition *t1* can only fire, when the signal is in state *go*. The interesting parts of this model are the identities of transition *t1*; when transition *t1* is fired, the object of the signal from place *go* stays on this place. Moreover, the animation is not restarted, because the identity is equipped with the *keep animation tag*.

Another interesting issue is the animation of the signal. The animation function is composed from two predefined animation functions: *appear; trigger*. The meaning is that these animations are started sequentially. When the first animation function has finished, the second is started. So, in both cases the signal

[7] If the corresponding arc has an identity, there is an option *keep animation* that does not stop the current animation on the object, but continues the animation while the token is on another place.

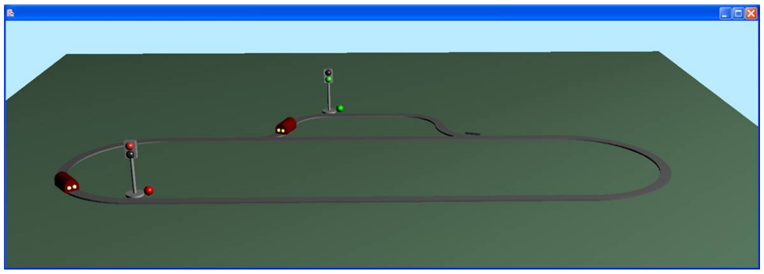

Fig. 5. Screenshot of a more complex toy-train

appears at position sig; then, it behaves as a trigger. A *trigger* is an animation function that simply waits for a user to click on that object. When this happens, the animation terminates (where the result depends on where the user clicked). In combination with the annotations *result:*{..} at the in-coming arcs of transitions *t3* and *t4*, the user can toggle the state of the signal by clicking on the signal.

Collisions. In our previous examples, there was only one locomotive. Figure 5 shows a screenshot of a more complex example, where there are two locomotives, two signals, and two switches. All objects are animated independently of each other. In particular, the signals and the switches can be toggled by the user by clicking on the corresponding objects. In this scenario, it could well happen that two locomotives move on the same track. On the one hand, the animations of these locomotives should be independent; on the other hand, the locomotives are solid objects such that they cannot be at the same place at the same time. Consider the situation shown in Fig. 5 again. Suppose that the first locomotive stops in front of the stop signal. Eventually, the second locomotive will approach the first locomotive. Then, the move animation of the second locomotive will be suspended. When the signal is switched to go by the user, the first locomotive will start moving; then, the second locomotive will resume its movement again and, eventually, will finish its animation. This way, the animation reflects the fact that objects are solid. Currently, we avoid collisions of solid objects, by suspending the corresponding animations when there is another object in front of it. This behaviour was inspired by material flow systems in which collisions of shuttles are avoided by infra-red detectors. But, we could also model other behaviour; for example, we could also stop the animation of objects, when they collide and return a special result value. This way, the Petri net model would be aware of a collision. More detailed concepts for reacting on collisions, however, need further investigations.

Parameters. The shapes and the animation functions for the places could take some parameters. Examples, are the size of an object or the speed of a movement. Moreover, there is an implicit parameter for the animations: the corresponding geometry object. Here, we do not deal with these parameters because we use low-level Petri nets only. For high-level Petri nets, the parameters for the shape and the animation function can be taken from the token itself.

3 Realization

The above concepts have been implemented in a prototype tool, which is based on the *Petri Net Kernel* (PNK) [10]. In the following, we discuss how the additional information is provided to this tool.

There are three types of information that must be provided to the tool: the annotations of the Petri net, the geometry, and the 3D-models for the animated objects. The annotations for the Petri net can be easily added as extensions to the Petri, by defining a new Petri net type. Here, we do not discuss the definition of such a Petri net type. Basically, there is a list of new annotations for each element of a Petri net, which is similar to the concept of annotations in PNML [3]. Moreover, the Petri net will have two global annotations: a reference to a *geometry file* and to a *models file*, which are discussed below.

The *geometry file* defines the layout of the system, i. e. it lists all the geometry objects in some XML syntax. For our toy-train example with one signal, the geometry file looks as follows:

```
<geometry>
   <circle id="sc1" shape="track" cx="0" cy="0" cz="0"
                                  sx="-10" sy="0" sz="0"
                                  angle="180" />
   <circle id="sc2" shape="track" cx="0" cy="0" cz="0"
                                  sx="10" sy="0" sz="0"
                                  angle="180"  />
   <point  id="sig" x="13" y="0" z="0" />
</geometry>
```

Basically, the XML file consist of a list of predefined elementary geometry objects. Currently, there are *points*, *lines*, *circles*, and *Bezièr curves*. Moreover, a geometry object could be composed from many elementary geometry objects. We call such geometry objects an *compound objects*. The attribute *id* is the unique identifier of the corresponding geometry object. This identifier will be used in the geometry annotations of the places of the Petri net in order to establish the correspondence between the Petri net and the geometry. The attribute *shape* defines its graphical appearance; it is a reference to a 3D-model, which will be discussed below. The other attributes depend on the chosen geometry object. For example, the attributes *cx*, *cy*, and *cz* define the center of a circle segment, attributes *sx*, *sy*, and *sz* define the start point of a circle segment, and attribute *angle* defines the angle of the circle segment (in clockwise orientation).

The *models file* defines the graphical appearance of the shapes used in the geometry file and the Petri net model. We call the shapes of the geometry file *static models*, because they are not animated. The shapes from the Petri net file are called *dynamic models*, because they are animated according to the animation functions given in the Petri net model. For our toy-train, the models file looks as follows:

```
<models>
   <static>
      <model id="track">
          <profile> <rectangle height="1.5" width="3" /> </profile>
          <texture name="track.jpg" />
      </model>
   </static>
   <dynamic>
      <model id="locomotive">
         <file name="locomotive.wrl" />
      </model>
      <model id="signalGo">
         <file name="lampRed.wrl" />
      </model>
      <model id="signalStop">
         <file name="lampGreen.wrl" />
      </model>
   </dynamic>
</models>
```

In the *static* section, we have the definition of tracks, which define the graphical appearance of the geometry objects in the visualization. It is defined by giving a profile and a texture. This way, the same definition can be used for all geometry objects. In our example, the profile is a rectangle[8] and the texture is some JPEG file. In the *dynamic* section, we define several 3D-models (one for each model referred to in the Petri net). Here, we refer to some VRML models.

In fact, the Petri net from Fig. 4 along with the above geometry file and the model file are sufficient for visualizing the Petri net model with our tool. The separation of the geometry file and the model file allows us to easily exchange the underlying layout as well as the graphical appearance of a model. This way, we have a clear separation between the dynamic behaviour which is modelled in the Petri net, the underlying layout, which is defined in the geometry file, and the graphical appearance, which is defined in the models file.

4 Conclusion

In this paper, we have introduced some simple concepts that allow us to easily equip a Petri net with a 3D-visualization. What is more, for obtaining a visualization, no programming is necessary. We only need to provide some 3D-models, a geometry, and some animation functions from a set of predefined animation functions.

[8] Currently, we have implemented the profile rectangle only. But, in future version, there can be other predefined profiles or even user defined profiles.

One of the principles underlying these concepts is *separation of concerns*. The 3D-visualization part should be as independent as possible from the Petri net itself. This way, it is possible to analyse the behaviour of the system without considering the details of the physical model. Actually, this is not always possible. For example, collisions of objects could result in deadlocks that are not present in the Petri net model itself. The investigation of such problems and the definition of some sufficient conditions for independence is one of the future research directions.

The implementation of our prototype demonstrates that the concepts work. Clearly, there could be much more features for obtaining more realistic animations. Such features will be added in a future version of the tool, which will also support high-level Petri nets. Which features are necessary and, on the other hand, which features do not compromise analysis results for the Petri net itself is another direction of future research.

References

1. R. Bardohl, C. Ermel, and L. Ribeiro. Towards visual specification and animation of Petri net based models. In *Workshop on Graph Transformation Systems (GRATRA '00)*, pages 22–31, March 2000.
2. Roswitha Bardohl, Claudia Ermel, and Julia Padberg. Formal relationship between Petri nets and graph grammars as basis for animation views in GenGED. In *Integrated Design and Process Technology IDPT 2002*, Society for Design and Process Science, June 2002.
3. Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, technology, and tools. In W. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003, 24th International Conference*, *LNCS* 2679 , pages 483–505. Springer, June 2003.
4. Design/CPN. `http://www.daimi.au.dk/designCPN/`. 2001/09/21.
5. Michael Kater. SimPEP: 3D-Visualisierung und Animation paralleler Prozesse. Masters thesis, Universität Hildesheim, April 1998.
6. Loise Lorentsen, Antti-Pekka Tuovinen, and Jianli Xu. Modelling of features and feature interactions in Nokia mobile phones using coloured Petri nets. In J. Esparza and C. Lakos, editors, *Application and Theory of Petri Nets 2002, 23rd International Conference*, *LNCS* 2360, pages 294–313. Springer, June 2002.
7. The PEP Tool. `http://parsys.informatik.uni-oldenburg.de/~pep`. 2002/07/29.
8. Jens Linneberg Rasmusen and Mejar Singh. *Mimic/CPN: A Graphical Animation Utility for Design/CPN.* Computer Science Department, Aarhus University, Aahrhus Denmark, December 1995.
9. Eric Verbeek. ExSpect 6.4x product infromation. In K. H. Mortensen, editor, *Petri Nets 2000: Tool Demonstrations*, pages 39–41, June 2000.
10. M. Weber and E. Kindler. The Petri Net Kernel. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technologies for Modeling Communication Based Systems*, LNCS. Springer, to appear.