3D-Visualization of Petri Net Models: Concept and Realization

Ekkart Kindler and Csaba Páles

Department of Computer Science, University of Paderborn
{kindler,cpales}@upb.de

Abstract. We present a simple concept for the 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. Moreover, we discuss a prototype of a tool implementing this concept: PNVis version 0.8.0.

Keywords: Petri nets, 3D-visualization, animation.

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 of the Petri net in order to indicate the corresponding action or document in the application area is only a first step.

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* [14], for example, uses the concept of a *dashboard* 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). In ExSpect, it is even possible to interact with the simulation via this dashboard. Another example is the Mimic library of Design/CPN [4, 13], 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 [10]. Another approach for visualizing Petri nets is based on graph transformations and their animation: GenGED [1, 2].

J. Cortadella and W. Reisig (Eds.): ICATPN 2004, LNCS 3099, pp. 464–473, 2004.

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 model. 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 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 show how a Petri net can be equipped with the information on the physical objects. Moreover, we discuss the concepts and a prototype of a tool that uses this information for a 3D-visualization of the system: PNVis version 0.8.0. Here, the focus is on those aspects of the physical objects that are necessary for visualization: basically the shape of the objects. The prototype of PNVis is restricted to low-level Petri nets. 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, and it will provide concepts for constructing a system from components in a hierarchical way.

PEP [11] was one of the first Petri Net tools that came up with a 3D-visualization of Petri net models: *SimPep* [6]. Basically, *SimPep* triggers animations in a *VRML model* [5] 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 independent objects are impossible. In this paper, we will present concepts that allow us to have concurrent animations of independent objects. The trick is to associate animations with places rather than with transitions.

2 Concepts

In order to animate the behaviour of a Petri net in a 3D-visualization, the net must be equipped with some information on the physical objects. Moreover, the behaviour of the objects must be related to the dynamic behaviour of the Petri net. In the following, we will discuss how to add this information to a 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 *animation* places. The idea is that each token on an animation 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 model [5]) that defines the shape of all tokens on this place. Defining the *behaviour* of an object is similar: Each animation

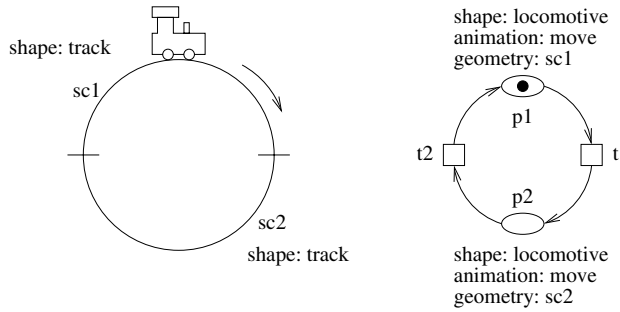

Fig. 1. A simple toy-train model

place is associated with an *animation function*, where the animation function is composed 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 or the object could *appear* at some point.

In order to illustrate these concepts, let us consider a simple example: a model of 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 *points*. 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 net model and the physical model is clear from the layout. Formally, this correspondence is defined by annotating each place with a reference to the corresponding element in the geometry. Place *p1* corresponds to the upper semicircle of the geometry *sc1* and place *p2* corresponds to the lower semicircle of the geometry *sc2*. The annotation *shape*[1] defines the shape of the physical objects corresponding to the tokens on this place. In our example, it is a locomotive for both places, where the details of the definition of the shape will be discussed in Sect. 3. For now, you can think of it as a reference to some VRML model of a toy locomotive. The annotation *animation* defines the behaviour of the object corresponding to a token on a place. This behaviour will be started when a token is added to the place. In our example, the behaviour is a *move* animation. Note that, without additional parameters, the animation function refers to the geometry object corresponding to that place. So, a locomotive corresponding to a token on place *p1* will move on semicircle *sc1*, and a locomotive corresponding to a token on place *p2* will move on semicircle *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

[1] Note that, in the implementation of the net type for the PNK, we call this extension *dynamic shape*.

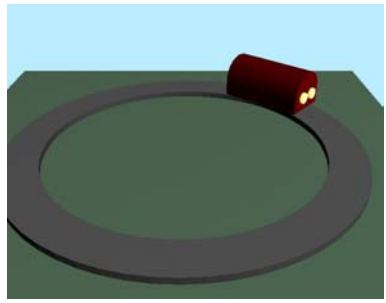

Fig. 2. Screenshot of the visualization

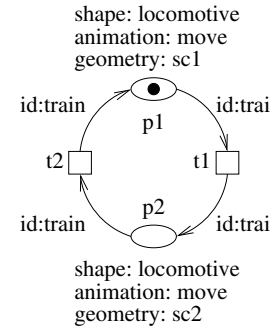

Fig. 3. The model with identities

object is equipped with an annotation *shape*, which defines the graphical appearance of the geometry object. In our example, the semicircles will be visualized as tracks. The precise definition of these tracks and their appearance will be discussed in Sect. 3. Once we have provided all this information, we can start the 3D-visualization of this system. Figure 2 shows a screenshot 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, the objects 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. Apart from the fact that this constant deletion and new creation of 3D-objects would be quite inefficient, this behaviour is not what happens in reality. In reality, the same physical object, the locomotive, moves from track *sc1* to track *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 an annotation *id*, which is some identifier n. 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; i. e. each identifier 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. Of course, we may have arcs without identity annotations. For an in-coming arc of a transition, this means that the corresponding object will be deleted. For an out-going arc of a transition, this means that a corresponding object will be created, where the shape of the newly created object is defined by the shape annotation of the place.

Finished and unfinished animations. Next, we consider the relation of the behaviour of the Petri net and the animations of the objects corresponding to the

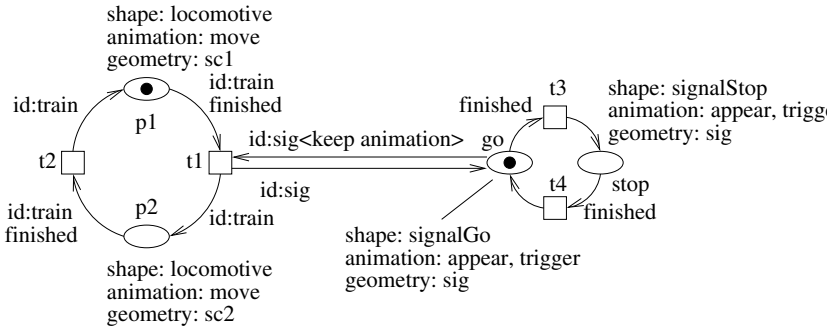

Fig. 4. A toy-train with a signal

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 from a place before the animation on its corresponding object is terminated? One idea would be to immediately stop the animation. In our example, this would not make much sense, because the locomotive would appear to 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. In our example, we would like to remove a token from a place only when the animation of the corresponding object is finished. On the other hand, there are examples in which it makes sense to remove a token from a place while an animation on the corresponding object 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 animation must be finished before the token may be removed, we add a label *finished* to the corresponding arc. If there is no such annotation, the transition need not wait until the animation of the corresponding object is terminated. In that case, there are two possibilities to proceed: Either the animation of the object is stopped or the animation is continued on the new place. When the animation should be continued for the token on the new place, the corresponding arc has an id with an additionally tag <keep animation>. When there is no such tag, the running animation is stopped and a new animation is started on the new place.

In order to illustrate these new concepts, we extend our example: We assume that there is a signal at the end of track *sc1* for which we add a position *sig* in the geometry somewhere at the end of track *sc1*. The idea is that the locomotive should stop at the end of track *sc1*, when the signal is in state *stop*; when the signal is in state *go*, the locomotive may enter track *sc2*. The Petri net in Fig. 4 models this behaviour. 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, i. e. the locomotive, on the tracks. The annotation *finished* 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). 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: *signalStop*. The object corresponding to a token on place *go* is a signal with its green light on: *signalGo*. These objects will appear at the point *sig* of the geometry (at the end of *sc1*). Due to the loop between place *go* and transition *t1*, transition *t1* can fire only 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 starts. So, in both cases the signal 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 in the 3D-visualization. When this happens, the animation terminates. In combination with the annotations *finished* at the in-coming arcs of transitions *t3* and *t4*, the user can toggle the state of the signal by clicking on the signal. A user's click on the signal object will finish the *trigger* animation running for this object; once the animation function is finished transition *t3* resp. *t4* will fire.

Animation results. In order to allow us more complex interactions between the Petri net model and the animations, the animation functions are equipped with a result value. The result of an animation could depend on the outcome of the animation function. For example, the outcome of the trigger animation, could depend on the part of the object the user clicked on. In some cases, we would like a transition to fire only when the animation function returns a particular result *n*. To this end, we annotate the corresponding arc with *result:*{*n*}. Actually, the annotation result may give a range of values *result:*{*0..3*} or *result:*{*0..*} or *result:*{*..10*}, where the last two annotations denote ranges that are open in one direction. The particular annotation *result:*{*..*} represents the full range of possible return values, which means that the corresponding animation must have terminated, but its value does not matter at all. Therefore, the notation *finished* introduced earlier is just a shorthand for *result:*{*..*}.

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 as well as the switches can be toggled by the user by clicking on the corresponding objects. This way, the user can control the route of the locomotives. In this scenario, it could well happen that two locomotives move on the same track. In principle, the animations of the different tokens in a Petri net are completely independent of each other. But, they may interfere, when two objects approach each other. The reason is that objects are considered to be solid. And solid objects cannot be at the same position 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

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

suspended (but not finished). So the second locomotive will stop right behind the first locomotive without finishing its animation. When the user clicks on the signal again, it is switched to go. Then, the first locomotive will be moved to the next track and a move animation on the next track will be started for this locomotive. When the first locomotive has moved a little bit, 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 can be aware of collisions. More detailed concepts for reacting on collisions, however, need further investigations.

Extensions. Here we have discussed those concepts only, that are already implemented in PNVis version 0.8.0. Future versions will support high-level Petri nets and parameterized animation functions and parameterized 3D-models [7].

3 Realization

The above concepts have been implemented in a prototype tool called PNVis, which is based on the *Petri Net Kernel* (PNK) [15] and uses Java3D for implementing the 3D-visualization. In the following, we discuss how the additional information is provided to PNVis.

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 and the geometry 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*.

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

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

Basically, the XML file consist of a list of predefined geometry objects, which are *points*, *lines*, *circles*, and *Bezièr curves*. Moreover, a geometry object could be composed from many predefined geometry objects. We call such a geometry object a *compound object*[2]. 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 the graphical appearance of the geometry object, which is a reference to a definition in the models file. The other attributes depend on the chosen geometry object. For example, 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)[3].

Models file. The *models file* defines the graphical appearance of the shapes used in the geometry file and the Petri net model. We call the shapes for the geometry file *static models*, and we call the shapes for the places of the Petri net *dynamic models*. 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>
```

[2] PNVis version 0.8.0 does not support Bezièr curves and compound geometry objects.
[3] Note that PNVis version 0.8.0 ignores the z-coordinates.

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. The idea is that the profile will be moved along the geometry object in order to defines its outline. The texture will be placed on this outline. This way, we need only one definition of a static shape for all types of geometry objects. In our example, the profile is a rectangle and the texture is some JPEG file[4].

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, the model file, and the VRML files 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 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.

One of the principles underlying these concepts is *separation of concerns*. The 3D-visualization part is quite independent from the Petri net itself. This way, the concept provides an abstraction mechanism, and it is possible to analyze the behaviour of the system without considering the details of the physical model. But, this is not always possible. For example, collisions of objects could result in deadlocks that are not present in the Petri net model alone. The investigation of such problems and the definition of sufficient conditions for the independence of the Petri net properties from the physical properties is one of the future research directions.

The implementation of *PNVis version 0.8.0* is now freely available and demonstrates that the concepts are feasible. PNVis runs on all systems on which Java and Java3D are installed. More detailed information on PNVis and its code can be found at [12]. Clearly, there could be much more features for obtaining more realistic animations. Such features will be added in a future version of PNVis; in particular, PNVis will also support high-level Petri nets. Which other features are necessary and appropriate is another direction of future research.

Acknowledgments

We would like to thank some anonymous reviewers for their comments on earlier versions of this paper.

[4] Note that PNVis version 0.8.0 supports the profile rectangle only, and it completely ignores textures.

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, 24^{th} International Conference*, *LNCS* 2679 , pages 483–505. Springer, June 2003.
4. Design/CPN. `http://www.daimi.au.dk/designCPN/`. 2004/03/12.
5. ISO/IEC International Standard. Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML) – Part 1: Functional specification and UTF-8 encoding. ISO/IEC 14772-1, 1997.
6. Michael Kater. SimPEP: 3D-Visualisierung und Animation paralleler Prozesse. Masters thesis (in German), Universität Hildesheim, April 1998.
7. Ekkart Kindler and Csaba Páles. PNVis: Documentation of version 0.8.0. PNVis homepage [12], March 2004 (evolving document).
8. Ekkart Kindler and Wolfgang Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:16–31, December 1996.
9. Ekkart Kindler and Hagen Völzer. Algebraic nets with flexible arcs. *Theoretical Computer Science*, 262:285–310, July 2001.
10. Louise 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, 23^{rd} International Conference*, *LNCS* 2360, pages 294–313. Springer, June 2002.
11. The PEP Tool. `http://parsys.informatik.uni-oldenburg.de/~pep`. 2004/03/12.
12. PNVis homepage. `http://www.upb.de/cs/kindler/research/PNVis`. 2004/03/12.
13. Jens Linneberg Rasmusen and Mejar Singh. *Mimic/CPN: A Graphical Animation Utility for Design/CPN.* Computer Science Department, Aarhus University, Aarhus Denmark, December 1995.
14. Eric Verbeek. ExSpect 6.4x product infromation. In K. H. Mortensen, editor, *Petri Nets 2000: Tool Demonstrations*, pages 39–41, June 2000.
15. Michael Weber and Ekkart 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* 2472, pages 109–123. Springer, 2003.