Petri Nets and the Real World

Ekkart Kindler and Frank Nillies
Department of Computer Science
University of Paderborn
D-33098 Paderborn, Germany
[kindler|frank]@upb.de

***Abstract*— Two years ago, we extended Petri nets by a simple but powerful concept for interactively animating systems as a 3D-visualization, which is called *PNVis*. The basic idea of PNVis is to equip the Petri net with information on how a token on some place corresponds to a physical object and how this object behaves. PNVis associates the simulation of tokens of the Petri net with these objects in the virtual 3D-world.**

In this paper, we take the next step and use the concepts of PNVis and associate the tokens of a Petri net with objects of the real world. This way, a Petri net can be used as a controller of some plant. In principle, this idea works for any kind of hardware; for simplicity, however, we demonstrate this idea by a Petri net for controlling a simple toy-train.

What is more, we show that the control and the 3D-visualization can be synchronized so that the visualization, basically, shows the behaviour of the real world. Altogether, this demonstrates that the concepts of PNVis are a powerful means for designing, prototyping, and validating controllers.

I. Introduction

Petri nets are a well-accepted formalism for modelling concurrent and distributed systems. The main advantages of Petri nets are their graphical notation, their simple semantics, and the rich theory for analyzing and verifying 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 application areas who are not experts in Petri nets. 'Playing the token-game' is not enough for understanding the behaviour of a complex system. The concepts of *PNVis* improve this situation by providing a simple mechanism for animating the behaviour of a system modelled as a Petri net in a 3D-visualization. The extensions of Petri nets that are necessary for such a 3D-visualization are remarkably simple [2], [3]: the tokens of the Petri net are associated with objects of a virtual world and with a behaviour. A simple feedback mechanism allows the 3D-visualization to have an effect on the behaviour of the Petri net.

The interaction between the actual *Petri net simulator* (*PNSim*) and the *visualization* (*PNVis*) is realized by a simple protocol. It turned out that this protocol can be used for visualizing systems in other formalisms than Petri nets by replacing *PNSim* by a simulator for some other formalism. Likewise, we can use *PNSim* with some other visualization tool such as a simple control panel, which allows users to interact with the Petri net simulation; this will result in a tool similar to *ExSpect* with its dash boards [5].

Even more interestingly, the interface to the virtual world of the visualization can be replaced by an interface to the real world, e. g. a machinery or plant. This allows us to control the plant directly by a Petri net (respectively by *PNSim* simulating the Petri net). In this paper, we present the basic concepts that allow us to control a plant by a Petri net. The concepts are, basically, the same as for visualizing a Petri net; the controller and visualization can even be used synchronously, which allows us to visualize the real behaviour of a plant while running. Since these concepts are quite simple and compatible with the standard Petri net semantics, these concepts seem to be universal for relating the behaviour and the analysis results of Petri nets to the real world.

II. PNVis

In this section, we give a brief overview of the extensions needed for visualizing Petri nets by the help of PNVis. To this end, a Petri net is equipped with some information on the shape and the dynamic behaviour of the objects corresponding to tokens on some places.

Shapes and animation functions: In a first step, we distinguish those places of a Petri net that correspond to virtual objects. We call them *animation places*. The idea is that each token on such a place corresponds to an object with its individual appearance and behaviour. In order to visualize and to animate a physical object, we need two pieces of information: its shape and its behaviour.

It is easy to define the *shape* of the object associated with a token on a place: Each animation place is associated with a *3D-model* (e. g. a VRML file) that defines the shape of all tokens on this place. Defining the *behaviour* of an object is similar: Each animation place is associated with an *animation function*. This 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, 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

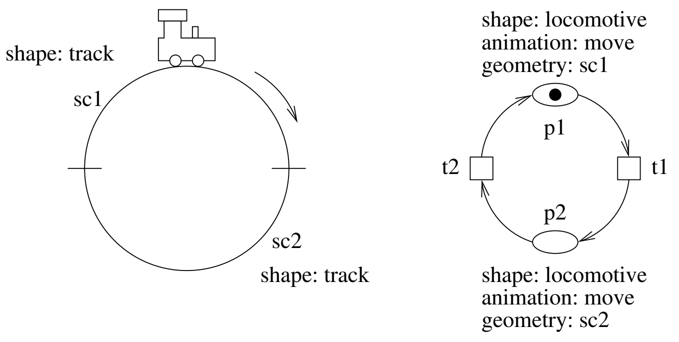

Fig. 1. A toy-train

segments, and Beziér curves. In our example, there is one toy-train 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 similar layout. Formally, this correspondence is defined by annotating each place with a reference to the corresponding element in the geometry. The annotation *shape* defines the shape of the objects. In our example, it is a toy-train, actually a *locomotive* only, for both places, where the details of the definition of the shape are discussed in [2], [3]. Here, we can think of it as the reference to some VRML model of a 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. Without additional parameters, each animation function refers to the geometry object corresponding to that place. Therefore, a toy-train corresponding to a token on place *p1* moves on track *sc1*, and a toy-train corresponding to a token on place *p2* moves on track *sc2*.

In order to make our example complete, we must provide some graphical information for visualizing the geometry objects. To this end, each geometry object can have an annotation *shape*, too. In our example, the semicircles *sc1* and *sc1* are visualized as tracks (see [2], [3] for details). Once we have provided this information, we can start PNVis for visualizing this system. Figure 2 shows a screen-shot of the 3D-animation of our example, where there is a toy-train moving on track *sc2*, which corresponds to a token on place *p2*.

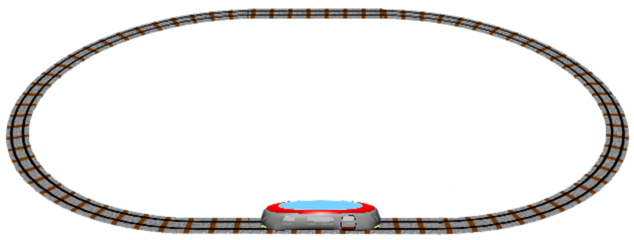

Fig. 2. Screen-shot of the visualization

Object identities: Up to now, the objects and the shapes corresponding to the tokens on the two places *p1* and *p2* are completely independent of each other. When transition *t1* fires, an object corresponding to the token on place *p1* is deleted and a new object corresponding to the new token on place *p2* is created and the move animation is started. Clearly, this is not what happens in reality. In reality, the same object, the toy-train, moves from track *sc1* to track *sc2*. In order to keep the identity of an 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 number. 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 an 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 identity of a transition 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[1].

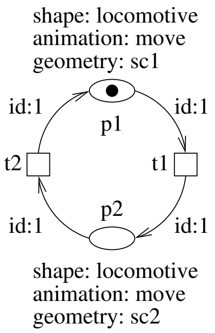

Fig. 3. The model with identities

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 toy-train would jump from its current position on the track to the start of the next track. Assuming that firing a transition 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. Therefore, we must explicitly define in the Petri net model whether a transition may or may not remove a token while an animation is still running on the corresponding object. When we want a transition to wait until the animation of a token has terminated before removing the token, we add the annotation *result:* {..} to the corresponding arc. Actually, an animation function has a return value, and the annotation *result* says for which return values of the animation the corresponding transition may fire. The set {..} stands for all possible return values. Altogether, *return:* {..} means that

[1]Both transitions have only one in-coming and out-going arc. Therefore, the example does not show the full power of identities. We will see a more exciting examples, soon.

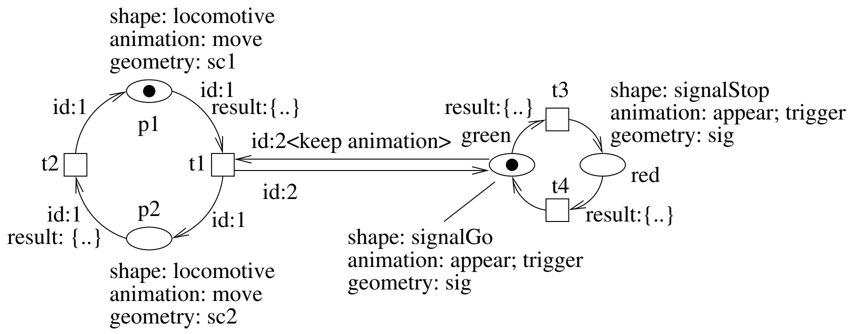

Fig. 4. A toy-train with a signal

the animation must terminate – with any return value – before the transition can fire. If there is no such annotation at the arc, the transition does not need to wait until the animation of the corresponding object terminates – when fired, the transition simply stops the animation[2].

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 *red*, the toy-train stops at the end of track *sc1*; when the signal is in state *green*, the toy-train 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 toy-train, on the tracks. The annotation *result:*{..} guarantees that the transitions wait until the move animation of the toy-train has come to an end (i. e. the toy-train has reached the end of the track). Next we consider the signal: The two states of the signal are represented by the places *red* and *green*. The object corresponding to a token on place *red* is a signal with its red light on: *SignalStop*. The object corresponding to a token on place *green* is a signal with its green light on: *SignalGo*. These objects will appear at the point *sig* of the geometry (somewhere at the end of *sc1*). Due to the loop between place *green* and transition *t1*, transition *t1* can fire only when the signal is in state *green*. The interesting parts of this model are the identities of transition *t1*; when transition *t1* is fired, the object of the signal from place *green* 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 of two predefined animation functions: *appear; trigger*. The meaning is that these animations are started sequentially. When the first animation function finishes, the second starts. In both cases, the signal appears at position *sig*; then, it behaves as a trigger. A *trigger* is an animation function that waits for a user to click on that object. When this happens, the animation terminates and a result value is assigned to the token; the assigned value depends on the part of the object on which 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 in the 3D-visualization.

[2]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.

III. Controlling Plants

In this section, we show how the concepts of PNVis can be used for controlling a plant via the same interface as the visualization; i. e. the Petri net simulator does not interact with the 3D-visualization, but with the hardware. In our implementation, we used a Märklin toy-train, which has an interface to a Linux workstation in order to interact with it. A picture of the toy-train system is shown in Fig. 5.

Actions and events: When the simulator interacts with the visualization, the simulator starts an animation of an object when a token is added to a place. Likewise, the simulator interacting with the hardware issues some *action* when a token is added to a place. Such an action can be the switching of some actuator of the hardware. Likewise, the simulator interacting with the hardware can issue an action when a token is removed from a place. The details on how to define actions, and how to associate them with a place of a Petri net will be discussed later.

In order to give the simulator feedback on the behaviour of the hardware, we will use *events*. An example for an event could be a rising edge of the position sensor at the end of some track – indicating that the toy-train has reached the end of this track. Then the token on the place representing the train on that track could be removed and added to the place representing the next track. When the event occurs, a result value is assigned to a token on the corresponding place (where the result value depends on the type of the event). This way, the Petri net simulator knows that the token can be removed. In our example, we have one sensor at the end of each track segment. The details on how to define events and how to relate them to a place will be discussed later.

In a nutshell, the actions on the hardware correspond to starting an animation in the visualization, and the events correspond to the termination of an animation function in the visualization. This allows the Petri net simulator to use the same interface to interact with the visualization and with

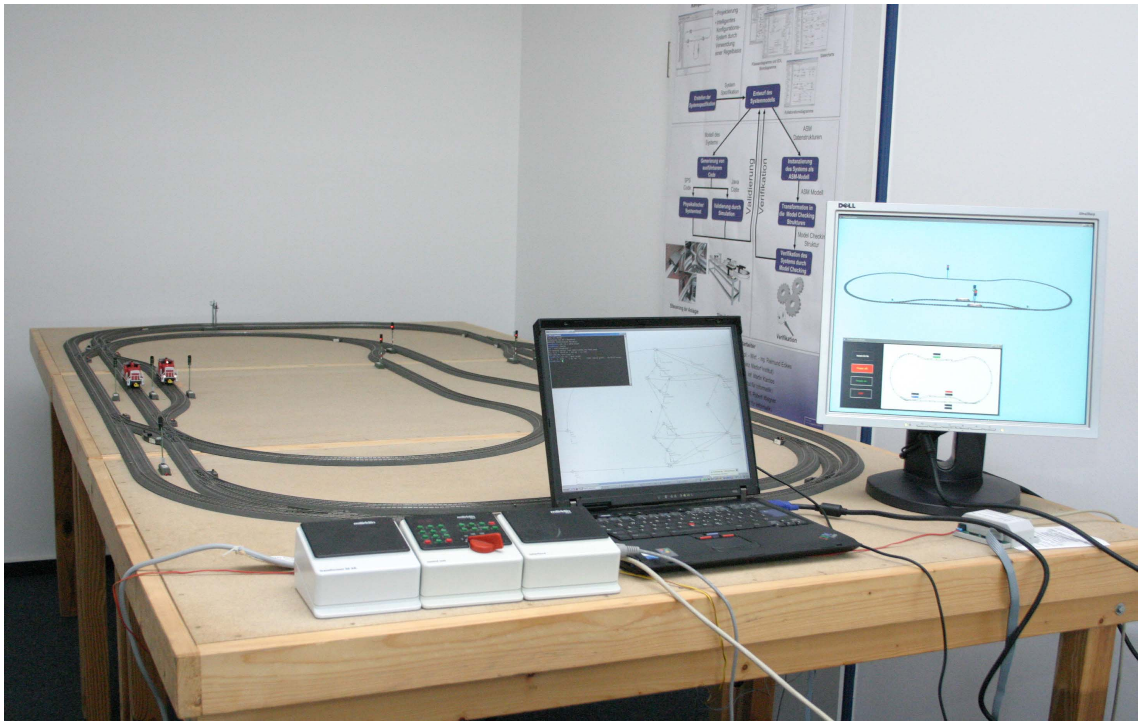

Fig. 5. The hardware: A Märklin toy-train

the hardware; we call this interface the *interaction handler* interface. Here, we do not go into the technical details of this interface.

Panels and buttons: Of course, the user would also like to interact with the hardware. For example, the user might want to toggle some signal to red or to green, or the user might want to toggle some switch from left to right or vice versa.

To this end, the hardware handler supports the definition of *buttons* on some control *panel*, which can be pressed by the user in order to interact with the hardware. An example panel for our toy-train example is shown in Fig. 6. The buttons can be activated and deactivated by corresponding actions, i.e. when tokens are removed and added to the corresponding places. The activated buttons will be highlighted and can be pressed by the user. Pressing an activated button, triggers an event, which in turn can be used to return a result value to some token, which enables the transition to fire.

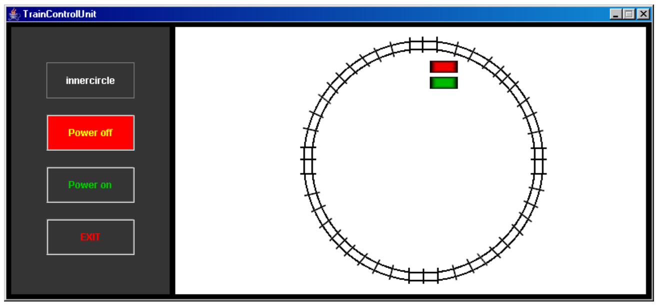

Fig. 6. Screen-shot of a simple control panel

Example: The actions and events as well as the panel with its buttons are defined in an XML file. A simplified version of the XML file for our toy-train example of Fig. 4 is shown in Fig. 7 at the end of this article.

The first part of this file defines the initialization of the hardware, i. e. the initial setting of all actuators. In our example, the signal (with hardware address 101) is set to green. In a second part, two buttons are defined, which allow the user to interact with the system. Each button has a position, a size (dimension), and an image that appears on that button. Moreover, the two colours `acolor` and `hcolor` define the colours of the button in the activated and the deactivated state. The definition of each button implicitly defines an event, which occurs when a user presses the button. The name of this event is given in the corresponding attribute in the button definition.

Moreover, the file defines some actions and events. Each action is assigned a name, and it refers to a component in the hardware by some id; actually, this id refers to some software object representing the hardware component in the *Hardware Abstraction Layer* (HAL). The attribute *type* defines the class of this object and the attribute *perform* refers to the method to be called on this object when the action is initiated.

Likewise, the definition of an event defines the name, and it refers to some hardware id (resp. a software object representing it). The attribute *type* defines its class, and the attribute *trigger* defines the value which triggers this event – actually, it is a change to this value triggering the event. In our example, the event *endSC1* is triggered once the sensor *S1* changes its value to 1, which indicates that the toy-train has reached the end of track *sc1*.

A second XML file defines how the different actions and events are associated with the places of the Petri net. The XML file for our example is shown in Fig. 8 at the end of this article. In this file, place *p1* is assigned the end-event *endSC1*. When this event occurs, a token on place *p1* will receive 0 as its result value[3]. For each of the places green and red representing the two states of the signal, we define two actions that are invoked, when a token is added to them: the first action enables the button for switching the signal to the other state, the second action actually switches the hardware signal to the state corresponding to the place (*red* or *green*). The action associated with the removal of a token is the deactivation of the other button. Moreover, the result value 0 is assigned to the token when the button is pressed, which will allow the Petri net simulator to fire the transition from *red* to *green* or vice versa.

When the Petri net simulator and the hardware handler are started with the Petri net from Fig. 4 and the two XML files from Fig. 7 and 8, they actually control the real hardware, where the user can interact with it via the panel shown in Fig. 6.

Synchronizing interaction handlers: Actually, we have another implementation of an interaction handler, which is capable of synchronizing one *master interaction handler* with many other *slave interaction handlers*. This way, it is possible to start the Petri net simulation with the hardware handler as the master and one or more visualization handlers as the slaves. Then the visualization shows the behaviour of the real hardware (see right monitor in Fig. 5). Due to variations in speed, there might be minor mismatches: For example, the toy-train in the visualization might stop at the end of a track because the real train has not arrived there yet; or the toy-train of the visualization might jump to the next track from its current position on a track when the real toy-train reaches the end of its track first. But, the deviations will be synchronized when the transitions fire and could be minimized by dynamically adjusting the speeds.

This synchronization shows that the interface is well-designed. But, we cannot go into the details of this interface here. You will find more details in [1], [4].

IV. Conclusion

In this paper, we have shown that the simple concepts of PNVis can be used not only for visualizing the behaviour of a real system, but also for controlling it. In order to allow the Petri net simulation to interact and synchronize with the real world, the concept of result values of tokens were used. These result values are set either by the visualization or by the real hardware. By the help of the *result* labels at some arcs, a transition can fire only when the tokens have particular result values. Note that this is compatible with the traditional firing rule of low-level Petri nets, where transitions fire non-deterministically and are not even required to fire at all. The result values just make the behaviour more deterministic. Therefore, the analysis results for the low-level Petri net are still valid.

This way, a single Petri net model can be used throughout the design process of plants such as flexible manufacturing systems – including analysis and verification as well as validation. In particular the behaviour can be simulated and visualized in early stages of the design process.

[3]Actually, the result values are irrelevant in this example.

References

[1] Dennis Beck. Steuerung von Anlagen durch Petrinetzmodelle. Bachelor thesis, Department of Computer Science, University of Paderborn (in German), June 2004.

[2] Ekkart Kindler and Csaba Páles. 3D-visualization of Petri net models: A concept. In G. Juhas and R. Lorenz, editors, *Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 69–78, September 2003.

[3] Ekkart Kindler and Csaba Páles. 3D-visualization of Petri net models: Concept and realization. In J. Cortadella and W. Reisig, editors, *Application and Theory of Petri Nets 2004, 25th International Conference, LNCS* 3099, pages 464–473. Springer, June 2004.

[4] Frank Nillies. Synchronisation einer 3D-Visualisierung mit einer realen Anlage auf der Basis von Petrinetzmodellen. Bachelor thesis, Department of Computer Science, University of Paderborn (in German), May 2005.

[5] Eric Verbeek. ExSpect 6.4x product infromation. In K. H. Mortensen, editor, *Petri Nets 2000: Tool Demonstrations*, pages 39–41, June 2000.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE occurrences SYSTEM "occurrence.dtd">

<hardwaredefinition>
  <hardwareinitialisation>
    <initial type="signal" adressid="101"
             state="green"/>
  </hardwareinitialisation>

  <buttons>
    <pushbutton id="red" event="redPressed"
      position="320,295" dimension="40,12"
      acolor="#FF0000" hcolor="#4C4C4C" />
    <pushbutton id="green" event="greenPressed"
      position="320,310" dimension="40,12"
      acolor="#00C800" hcolor="#4C4C4C" />
  </buttons>

  <events>
    <event name="endSC1">
      <attributes type="sensor" id="s1"
                  trigger="1"/>
    </event>
    <event name="endSC2">
      <attributes type="sensor" id="s2"
                  trigger="1"/>
    </event>
  </events>

  <actions>
    <action name="enableGreen">
      <attributes type="button" id="green"
                  perform="settoenable"/>
    </action>
    <action name="disableGreen">
      <attributes type="button" id="green"
                  perform="settodisable"/>
    </action>
    <action name="enableRed">
      <attributes type="button" id="red"
                  perform="settoenable"/>
    </action>
    <action name="disableRed">
      <attributes type="button" id="red"
                  perform="settodisable"/>
    </action>

    <action name="fire101green">
      <attributes type="signal" id="101"
                  perform="switchToGreen"/>
    </action>
    <action name="fire101red">
      <attributes type="signal" id="101"
                  perform="switchToRed"/>
    </action>
  </actions>
</hardwaredefinition>
```

Fig. 7. Actions, events and panel definition

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE relations SYSTEM "relation.dtd">

<eventactiondefinition>

  <place name="p1">
    <endEvent>
      <event name="endSC1" result="0"/>
    </endEvent>
  </place>

  <place name="p2">
    <endEvent>
      <event name="endSC2" result="0"/>
    </endEvent>
  </place>

  <place name="green">
    <onAdd>
      <action name="enableRed"/>
      <action name="fire101green"/>
    </onAdd>
    <onRemove>
      <action name="disableRed"/>
    </onRemove>
      <endEvent>
    <event name="redPressed" result="0">
      </endEvent>
    </place>

  <place name="red">
    <onAdd>
      <action name="enableGreen"/>
      <action name="fire101red"/>
    </onAdd>
    <onRemove>
       <action name="disableGreen"/>
    </onRemove>
    <endEvent>
      <event name="greenPressed" result="0">
    </endEvent>
  </place>

</eventactiondefinition>
```

Fig. 8. Associating actions and events with the places