Building and Deploying Visualizations of Coloured Petri Net Models Using BRITNeY Animation and CPN Tools

Michael Westergaard and Kristian Bisgaard Lassen

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {mw,k.b.lassen}@daimi.au.dk

Abstract. The contribution of this paper is a tutorial in the use of BRITNeY animation tool together with CPN Tools to make different views on Coloured Petri Nets. Examples of such views are message-sequence charts, gantt-charts, or SceneBeans animations showing the state of the model. In this paper we will describe how to generate message-sequence charts from executions of Coloured Petri Nets and how to create SceneBeans animations.

Keywords: Model-driven prototyping; visualization; Coloured Petri nets; CPN Tools.

1 Introduction

Formal models have proved their usefulness in modeling and understanding of complex systems [2,12,14,18], e.g., for verification of existing behavior or requirements engineering of needed behavior. However, when using a formal model such as Colored Petri nets (CP-nets or CPN) [8,11], it is only people familiar with the formalism who are truly able to understand the model of the system. A domain-user may understand the formalism used but is not capable, as the expert, to fully understand the model. Therefore formal models of systems are prone to be erroneous if they can not be fully understood and validated by a user with domain knowledge.

In this paper we present the tool BRITNeY[1] animation [23] which introduce an animation layer for CPNs. By using BRITNeY animation to animate CP-nets of systems the above problems of formal methods, being hard to understand, is alleviated. BRITNeY animation provides a uniform way to implement, integrate, and deploy visualizations of CPN models and has a pluggable architecture which make it possible to write customized plug-ins to animate the model but it also have more than a dozen predefined plug-ins for message sequence charts, SceneBeans [16, 19], plot different kinds of graphs and more. BRITNeY animation has already been used successfully to animate a network protocol [13] and

[1] Originally an abbreviation for Basic Real-time Interactive Tools for Net-based animation.

to animate the work flow of a bank work process for the purpose of requirements engineering [9].

We will describe methods to develop an animation on top of a CPN model by using an example of two runners. We describe the methods at the level where it is possible to replicate our example and use the methods in other projects.

The paper is structured as follows: In Sect. 2 we give a short overview of the architecture of the BRITNeY animation package. In Sect. 3 we briefly describe the example used throughout the paper, and go on to describe how to visualize the execution of the model using message sequence charts and a graphical animation. In Sect. 4 we mention related work and outline some of the planned new features of BRITNeY animation.

2 Architectural Overview

A well-known design pattern from object oriented software, is the model-view-controller (MVC) design pattern [5]. In the MVC design pattern, three participants collaborate to provide the implementation of an application, namely a model, a view, and a controller, see Fig. 1. The model models the state of the system, the view is a (graphical) representation of the current state of the model, and the controller implements the behavior of the system. The view may initiate actions in the controller.

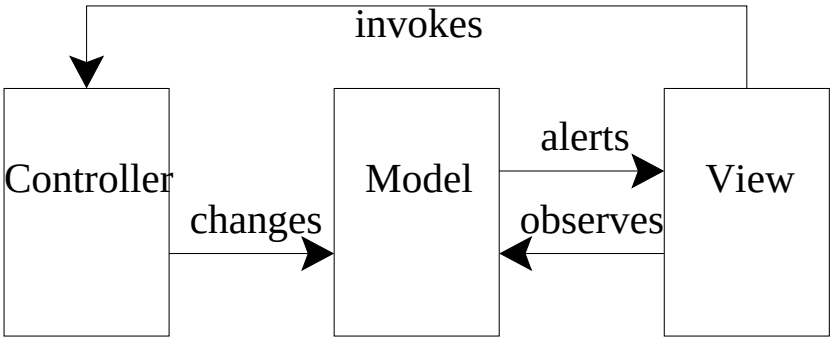

Fig. 1. Architectural overview of the model-view-controller design pattern.

The idea behind the BRITNeY animation package is to use a CPN model to model the state and behavior of the system (the model and controller), and use BRITNeY animation to model the view of the system. For more information on the architecture of BRITNeY animation, please refer to [22, Chap. 4].

3 Building Visualizations

In this section we will describe how to create two different graphical views of a CPN model. In Sect. 3.1, we give a brief description of how to work with the animation tools and the CPN model that will be visualized. In Sect. 3.2 we will describe the model we intend to create views of. In Sect. 3.3, we will show how to generate message sequence charts from the execution and finally in Sect. 3.4, we will describe how to create a domain specific animation. We will assume the reader knows CPN Tools, but we assume no prior knowledge of BRITNeY animation.

3.1 Working with BRITNeY

The animation tool can be obtained from the home-page [23] either as a downloadable version or as a Java Webstart [7] application.

In order to design animations that are easy to deploy, a certain design pattern must be used when adding animation to your model. The main idea of the design pattern is to add an init-transition, which is the only transition enabled in the initial step. All initialization of the animation can then be done in the action part of this transition. The reason we need to do this is that we will need to set up the animation for each simulation, but for efficiency, declarations are only evaluated once in CPN Tools. An overview of the points we have to go through is in Listing 1.1. In rest of this section, we will describe and exemplify each of these points in more detail.

Listing 1.1. The steps required to create an animation from an existing CPN-model.

1. Add declarations of structures for each animation plug-in you will need
2. Add an init-transition
3. Add code needed to set up the animation in an action part on the init-transition
4. Tie the execution of the model to the animation using action parts
5. (Optionally) deploy your animation on the web

3.2 Model

In this paper we will use a simple example to show the different aspects of animating a CP-net. In the example, two runners are competing to win a race. Initially the runners are at the start of the race, and they can then continue to the end of the race. During the race, the runners pass a drink stand. When the first runner passes the finish-line, he is declared the winner of the race, and a flag is lowered to celebrate. When the other runner crosses the finish-line, he is declared the loser of the race.

The CPN model in Fig. 2 captures the behavior of the example. In the CP-net, the two runners are modeled by tokens runner(1) and runner(2), both initially

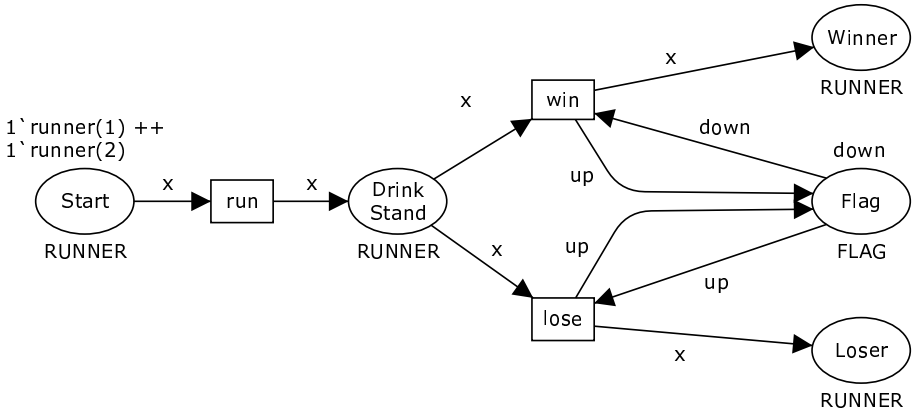

Fig. 2. CPN model of runner example.

placed on the Start-place. And the Flag is up. The run-transition moves the token representing a runner from the Start-place to the Drink Stand-place. When the flag is up, a runner-token on the Drink Stand-place is moved to the Winner-place and the flag is moved down by the win-transition. When the flag is down, the lose transition moved runner-tokens from the Drink Stand-place to the Loser-place.

3.3 Message-Sequence Charts

Message sequence charts (MSC) are well-known to protocol engineers, and it is therefore a good idea to be able to present the execution of a CPN model as an MSC. Using BRITNeY animation, it is very easy to create MSCs from CP-net executions. In this section, we will describe the process of adding an animation view to your model using MSCs as example. An example of an MSC that is generated from the model can be seen in Fig. 3.

The main points of this section are presented in the listings, which can later be followed in order to add animation to another CPN-model.

BRITNeY animation has a pluggable structure, meaning it can support different animation plug-ins. Out of the box, BRITNeY animation supports more than 10 different plug-ins. In order to be able to use an animation plug-in in our model, we must set up a connection to it using a declaration (as we would normally do in CPN Tools to declare a color-set or a CPN-variable). An example of a declaration of a connection to an animation plug-in can be seen in Listing 1.2. Here we declare a new connection called msc to an MSC animation plug-in. We initialize the animation plug-in with a user-friendly name, which will displayed to the user, namely Runners. All declarations of connections to animation objects are of this kind. We can interchange MSC with the name of the desired animation plug-in, we can use any identifier instead of msc, and we can give any string as the user-friendly name instead of Runners. The operations supported by the MSC animation plug-in can be seen in Listing 1.3. The plug-in allows

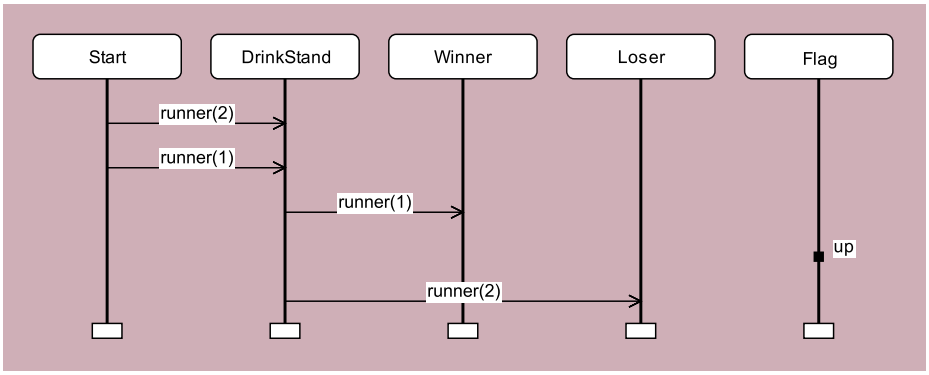

Fig. 3. Example of an MSC generated by the model in Fig. 2

us to create processes, create events between 2 processes and to create internal events inside a single process. A more detailed description of the MSC animation plug-in interface along with documentation of all animation plug-ins can be seen on the BRITNeY animation home-page [23].

Listing 1.2. The declaration of a structure that can be used to communicate with the animation plug-in for drawing MSCs.

```
1 structure msc = MSC(val name = "Runners");
```

Listing 1.3. The interface of the MSC animation plug-in.

```
1 void addProcess(String name)
2 void addEvent(String from, String to, String name)
3 void addInternalEvent(String process, String name)
```

The next step is to create an init-transition, a new transition that is always executed before all other transitions, while allowing the rest of the execution of the model to proceed as before adding the init-transition. One way to do this is outlined in Listing 1.4. Concrete examples of the application of this procedure can be seen by comparing the net in Fig. 2 with the nets in Figs. 4 and 6 where init-transitions have been added at the top left.

Listing 1.4. Adding an init-transition.

1. Create a transition named init
2. Create a place named Not inited of type UNIT with initial marking ()
3. Create a place named Inited of type UNIT with no initial marking

4. Create an arc from the Not inited-place to the init-transition with inscription () and an arc from the init-transition to the Inited-place
5. (If you have a hierarchical model) add copies of the Not inited-place on all pages, and fuse all together in a single fusion-set
6. Create a double-arc[2] with inscription () between the Inited-place and all transitions enabled in the inital step (except for the init-transition)

When we have created an init-transition, we need to add our initialization code in the action part of the init-transition. In our case, we will create 5 processes in the MSC, one for each of the places. The action part of the init-transition can be seen in Listing 1.5. Here we simply make 5 calls to the MSC animation plug-in, each call creating a new process.

Listing 1.5. The action part used to set up the animation.

```
INPUT ();
OUTPUT ();
ACTION
msc.addProcess("Start");
msc.addProcess("DrinkStand");
msc.addProcess("Winner");
msc.addProcess("Loser");
msc.addProcess("Flag")
```

Now, we have set up the entire animation and only need to update it during the animation. This is currently done by adding action parts to the model. In this example, we will create an action part for each of the transitions, which adds events to the MSC. For example, the action part for the run-transition can be seen in Listing 1.6. Whenever the run-transition occurs, an event is added from the Start process to the DrinkStand process, with a label describing which runner moved from the start of the race to the drink stand. Similar action parts are added to the other transitions.

Listing 1.6. The action part of the run-transition which updates the animation.

```
INPUT x;
OUTPUT ();
ACTION
msc.addEvent("Start", "DrinkStand", RUNNER.mkstr x)
```

The net we obtain after adding an init-transition, set-up code and action parts to all transitions can be seen in Fig. 4. The changes created to the original model from Fig. 2 are highlighted.

[2] CPN Tools uses interleaving semantics for simulation. In tools using partial order semantics, we would use a test-arc rather than a double-arc to preserve the semantics of the model.

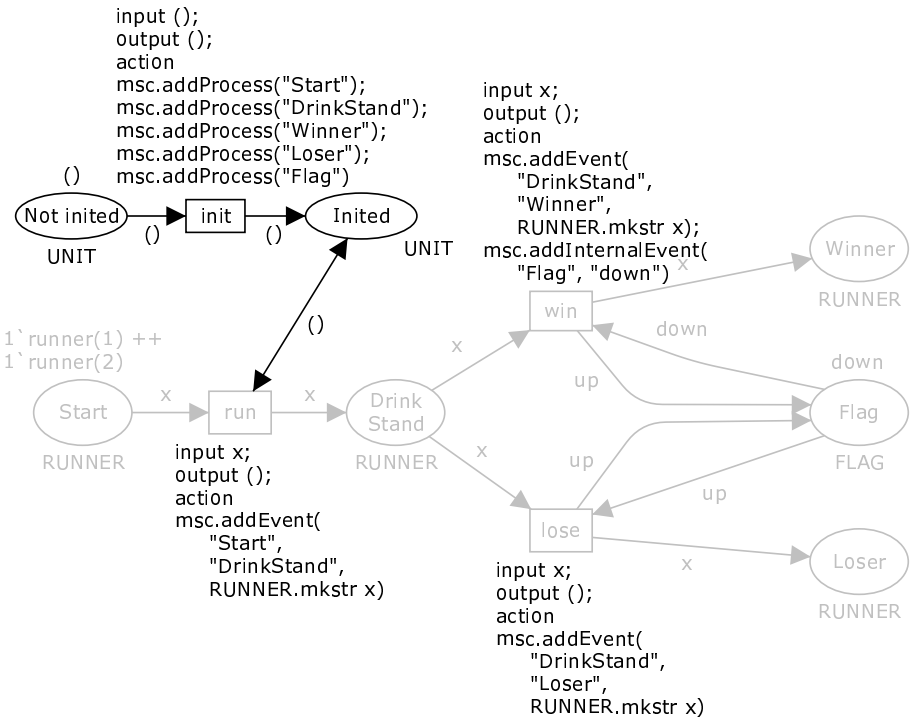

Fig. 4. CPN model of runner example which is able to generate an MSC of the execution.

We can now simulate the model in CPN Tools, and will notice that the animation is updated. We can also quit CPN Tools, and just simulate the model in BRITNeY animation. We will no longer be able to see how the model is updated, but we are still able to see that the animation is updated. If we want to see how the model is updated, the easiest way to deploy the animation is to simply require that users have CPN Tools and BRITNeY animation installed, and distribute the model saved by CPN Tools.

If we want to distribute a prototype to several users, who have little or no knowledge of CP-nets, we have an easier option, however. We can combine BRITNeY animation's ability to dump a simulator for a net with BRITNeY animation's powerful scripting engine and Java Webstart to deploy a visualization of a CP-net that can be started by clicking on a single link on a web-page. The steps we have to go through are outlined in Listing 1.7. We basically have to dump a simulator for our net, create a script to load a simulator and create a webstart loader for our simulator.

Listing 1.7. How to deploy a visualization of a CPN model.

1. Save a simulator for your net
2. Create a script to load and start your simulator
3. Download the offline version of BRITNeY animation
4. Modify `webstart/britney.jnlp` to suit your needs
5. Copy the files to a web-server

Saving a simulator for your net is easy:

Listing 1.8. Saving a simulator for a CP-net.

1. Load your net in CPN Tools
2. Ensure you are at the initial state
3. In BRITNeY animation find the simulator console corresponding to your net
4. Right click on the background of the simulator console
5. Select "Save simulator as" from the marking menu
6. Select a suitable name and location for the simulator

Now we need to create a script to load our simulator. BRITNeY allows us to create scripts using the full power of Java (but a more relaxed syntax if we desire so). We shall not go into detail about the scripting facilities, but just mention that we have access to an object, SimulatorService, which takes care of starting CPN simulators. A simple script to start a simulator can be seen in Listing 1.9. The first two lines of the script downloads and loads a simulator from the location `http://www.daimi.au.dk/~mw/local/tincpn/simulator.x86-win32`. You will obviously need to change the location of the simulator. The resulting simulator is very low-level, and only used to construct a more high-level simulator. Line 3 of the script obtains this high-level simulator. One feature of the high-level simulator is the ability to simulate nets. In line 4 of the script, we start a new simulation of 10 steps in 1 step increments. We pause for 1000 ms after each simulation step and we show a progress-bar at the bottom of the tool. You will probably need to adjust these parameters, at least the total number of steps and the amount of time to pause. Save the script as something reasonable at the same location you saved your simulator.

Listing 1.9. A simple script to download and start a simulator.

```
1 s = SimulatorService.getNewSimulator(
2     new java.net.URL("http://www.daimi.au.dk/~mw/local/tincpn/sim.x86-win32"));
3 hs = s.getHighLevelSimulator();
4 hs.startSimulation(10, 1, 1000, true);
```

The offline version of BRITNeY animation can be obtained from the homepage [23], and when you unzip the archive, you will find three jar-files and a number of directories. You should edit the file `webstart/britney.jnlp`. You need to make the changes:

Listing 1.10. Changes to make to `webstart/britney.jnlp`.

1. Change `http://www.daimi.au.dk/~mw/local/tincpn/` to location you intend to deploy your visualization everywhere in the file
2. Uncomment the line containing "¡property name="tincpn.script.load"...."
3. Change the location of the configuration script to the location you intend to store your configuration script
4. (Optionally, for experienced users only) remove any plug-ins, you do not need

Finally you need to copy the files to your web-server. The files, you will need are:

Listing 1.11. The files you need to make available to deploy an animation.

1. All files and directories from the offline version of BRITNeY animation
2. Your edited copy of `britney.jnlp` (should live at the top with `britney.jar`)
3. Your configuration script
4. Your simulator

Now you can simply point your users to the location of `britney.jnlp` on your web-server. A nice way to do that is to create a web-page with some information about the visualization and some use-scenarios and link to `britney.jnlp` from that page.

3.4 SceneBeans Animation

Fig. 5. Animation of runner example.

This section describes how to build a SceneBeans animation. We will reuse our example and show two runners go from start to finish crossing the drink stand on the way. Fig. 5 shows a snapshot of the animation.

We describe the process by incrementally changing our CP-net from Fig. 2 and a SceneBeans XML description to achieve our goal. We will use the procedure from Listing 1.1, which we used to create an MSC view of the model.

The first thing that needs to be done is to set up a connection to the SceneBeans animation plug-in in BRITNeY. This is done using the declaration as in Listing 1.12.

Listing 1.12. The declaration of a structure that can be used to communicate with the animation plug-in for SceneBeans.

```
1 structure runners = SceneBeans(val name = "Runners")
```

Listing 1.13 shows the interface accessible through the structure.

Listing 1.13. The interface of the SceneBeans animation plug-in.

```
1 String  getNextEvent()
2 boolean hasMoreEvents()
3 void    invokeCommand(String name)
4 String  peekNextEvent()
5 void    setAnimation(String filename)
```

We then need to create an initialization transition as described in Listing 1.4.

SceneBeans descriptions are written in an XML file. To specify the location of this XML file use the expression as in Listing 1.14 on the initialization transition, as in Fig. 6.

Listing 1.14. Setting the location of the SceneBeans specification.

```
1 runners.setAnimation "path-to-XML-file"
```

In Listing 1.14 the path-to-XML-file can e.g. be c:/animation.xml. When the simulator has evaluated the expression, BRITNeY animation shows the animation as it will look before any behavior is executed.

We tie the CP-net to the animation by specifying in code segments what should happen when a transition occurs. In Fig. 6 we have annotated our example net with the needed code segments.

The two functions which are called in the code segments have similar definitions. In the following we will only look at the runToDrinkStand function and how it is hooked up to the animation. The function is defined as:

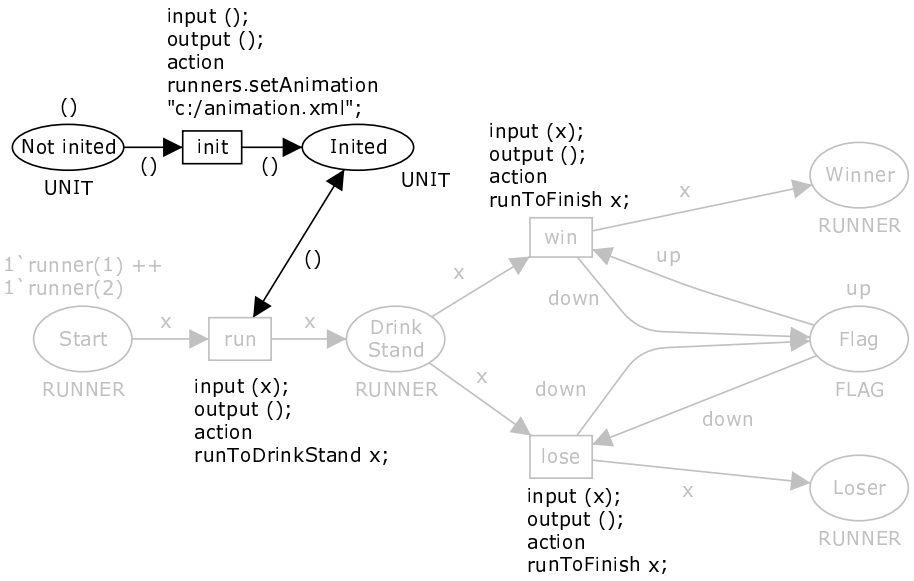

Fig. 6. Runner CP-net annotated for the SceneBeans animation.

```
fun runToDrinkStand x =
 let val _ = if x = runner(1)
             then runners.invokeCommand "runToDrinkStand-r(1)"
             else runners.invokeCommand "runToDrinkStand-r(2)"
     val _ = runners.waitForEvent "command-executed"
 in ()
 done
```

Two things are worth to notice in this function. Firstly, we use the function invokeCommand to execute a predefined command in SceneBeans; this is executed asynchronously. Secondly, we use the function waitForEvent to wait for a named event; in our case the event command-executed. The reason for why not just use invokeCommand is that it is executed asynchronously, so a situation could occur where two behaviors are executed concurrently. This might be the wanted behavior in some cases but not in our example.

We have now finished in adapting our CP-net to the SceneBeans animation. The final thing that needs to be done is to specify the commands in SceneBeans which we called from our CP-net. The XML code for this animation can be seen in Listing 1.15. For the purpose of this tutorial it is not important to read and understand the code. It is included for the reader to see the connection between the CP-net and the SceneBeans specification.

Listing 1.15. SceneBeans Specification of the Runners

```
<animation height="150" width="400">
  <forall var="i" values="1 2">
    <behaviour algorithm="move" event="command-executed"
               id="runToDrinkStand-r(${i})">
      <param name="from" value="10"/>
      <param name="to" value="150 + 30 * ${i}"/>
      <param name="duration" value="1"/>
    </behaviour>
    <event event="command-executed" object="runToDrinkStand-r(${i})">
      <announce event="command-executed"/>
    </event>
    <behaviour algorithm="move" event="command-executed"
               id="runToFinish-r(${i})">
      <param name="from" value="150 + 30 * ${i}"/>
      <param name="to" value="340 + 10 * ${i}"/>
      <param name="duration" value="1"/>
    </behaviour>
    <event event="command-executed" object="runToFinish-r(${i})">
      <announce event="command-executed"/>
    </event>
    <command name="runToDrinkStand-r(${i})">
      <reset behaviour="runToDrinkStand-r(${i})"/>
      <start behaviour="runToDrinkStand-r(${i})"/>
    </command>
    <command name="runToFinish-r(${i})">
      <reset behaviour="runToFinish-r(${i})"/>
      <start behaviour="runToFinish-r(${i})"/>
    </command>
  </forall>
  <draw>
    <forall var="i" values="1 2">
      <transform type="translate">
        <param name="translation" value="(0,50)"/>
        <animate behaviour="runToDrinkStand-r(${i})" param="x"/>
        <animate behaviour="runToFinish-r(${i})" param="x"/>
        <transform type="scale">
          <param name="x" value="0.4"/>
          <param name="y" value=".4"/>
          <primitive type="sprite">
            <param name="src"
                   value="file:///c:/Runners/runner-${i}.gif"/>
          </primitive>
        </transform>
```

```
44        </transform>
45      </forall>
46      <transform type="translate">
47        <param name="translation" value="(150,10)"/>
48        <transform type="scale">
49          <param name="x" value="0.4"/>
50          <param name="y" value=".4"/>
51          <transform type="scale">
52            <param name="y" value="0.9"/>
53            <primitive type="sprite">
54              <param name="src"
55                     value="file:///c:/Runners/drink-stand.jpg"/>
56            </primitive>
57          </transform>
58        </transform>
59      </transform>
60      <transform type="translate">
61        <param name="translation" value="(10,50)"/>
62          <transform type="scale">
63          <param name="x" value="0.4"/>
64          <param name="y" value=".4"/>
65          <primitive type="sprite">
66            <param name="src"
67                   value="file:///c:/Runners/start.JPG"/>
68          </primitive>
69        </transform>
70      </transform>
71      <transform type="translate">
72        <param name="translation" value="(350,50)"/>
73        <transform type="scale">
74          <param name="x" value="0.4"/>
75          <param name="y" value=".4"/>
76          <primitive type="sprite">
77            <param name="src"
78                   value="file:///c:/Runners/stop.JPG"/>
79          </primitive>
80        </transform>
81      </transform>
82    </draw>
83  </animation>
```

Lines 3-8 and 12-17 in the XML file describe the behaviors that exist in the animation; i.e. both runners can run from start to the drink stand and from the drink stand to finish. Lines 9-11 and 18-20 describes that events will be generated after each behavior is executed; e.g. after runner one goes from start to the drink stand an event is generated saying that the behavior has completed

to signal to the CP-net it can proceed on simulating. Lines 21-28 specify the commands which can be executed on our SceneBeans; i.e. the commands we call from the CP-net, e.g., runToDrinkStand-r(1). Lines 30-82 describes the figures that is in the animation; i.e. the two runners, start and finish signs and the drink stand. Also, this part describes how the figures are related to the defined behaviors; i.e. runner number one have the two behaviors runToDrinkStand-r(1) and runToFinish-r(1).

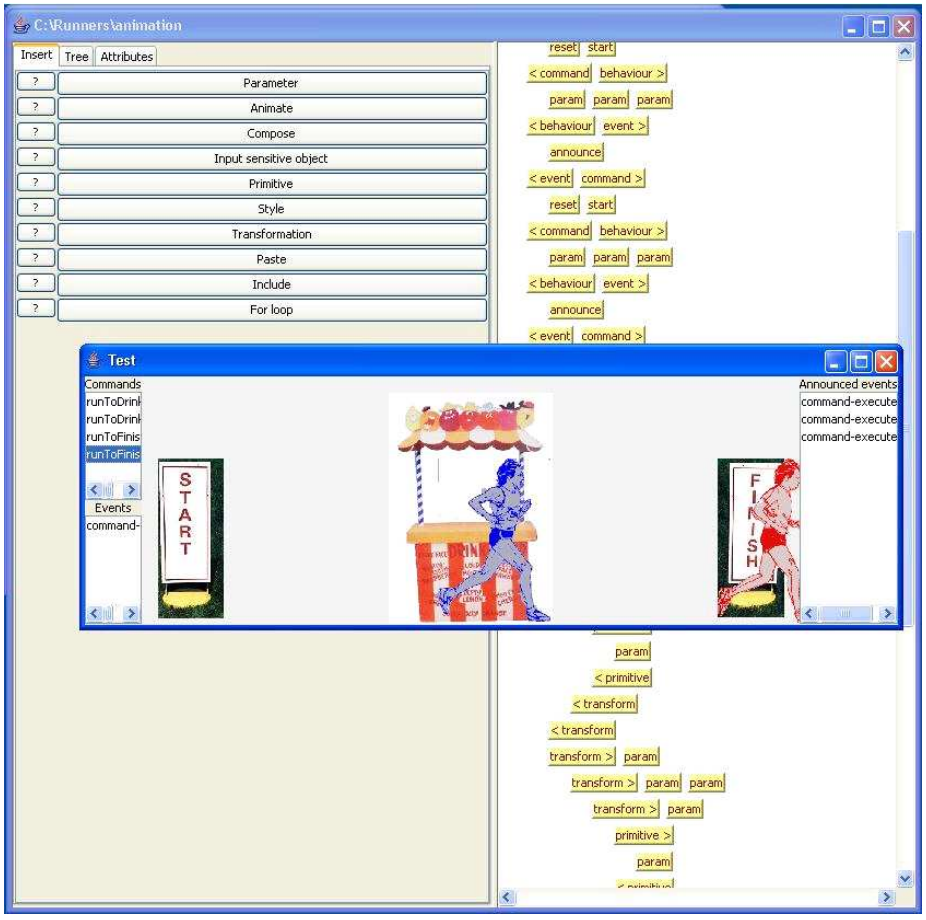

Fig. 7. SceneBeans XML editor.

The actual SceneBeans animation can be built in SClub, which is an extended version of BRITNeY animation, which provides, among other things, an XML editor for specifying SceneBeans animations. This editor interprets the XML file while it is being constructed so it is possible to preview the animation and test

commands as you go along. Figure 7 shows the runners animation being built and tested. In the figure you can see that the editor is aware of which constructs to suggest depending on the context of position of the cursor. Also, you can see the defined commands and events. These can be executed by double-clicking on them.

This is all that is needed to make a simple animation using the SceneBeans plug-in. Obviously, this is a very small example but the techniques used to make this animation are well suited for large scale animations.

4 Related Work and Future Improvements

BRITNeY animation supports adding animations to CPN models by annotating transitions with function calls, which are executed whenever the transition occurs. In the following, we outline how a number of other modeling tools allow users to use visualization.

ExSpect [20], a tool for modeling based on CP-nets, allows the user to view the state by associating widgets with the state of the model, and asynchronously interact with the model, also using simple widgets. In this way, it is easy to create simple user interfaces that support displaying information, but support for creating more elaborate animations is not easily available.

Mimic/CPN [17] makes it possible to animate models within Design/CPN [1, 3], which is another tool for modeling using CP-nets. CPN models are animated by Mimic/CPN by using function calls that are executed whenever a transition of the CP-net occurs. The animations are drawn using an application that resembles traditional drawing programs. Input from the user is possible by showing a modal dialog, where the simulation of the model is stopped while the user is expected to input information. It is also possible to make click-able regions, and the model can then query if one of these has been clicked.

LTSA [15], a tool for modeling using timed labeled transition systems, allows users to animate models using the SceneBeans library [16, 19]. In LTSA animations are tied to the models by associating each animation activity with a clock; resetting a clock corresponds to starting an animation sequence. The animation sequence or a user with his mouse can then send events which correspond to the progress of the timer.

Another approach, taken by e.g. the Comms/CPN [4] library for Design/CPN and CPN Tools, is to provide a TCP/IP abstraction, allowing the user to code the user interface in any language and use RPC to communicate with it. This approach resembles creating real programs quite a lot, but the user has to go through the hassle of implementing RPC himself, making this approach difficult to use in practice.

PNVis [10] is an add-on for the Petri Net Kernel [21], a highly modular tool for editing Petri nets. PNVis associates tokens with 3D objects and certain places with locations in a 3D world. Moving tokens corresponds to moving the associated object in the 3D world. PNVis is suitable for modeling physical systems, but not really useful for creating prototypes of software.

The Play-Engine [6] supports the developer in implementing a prototype by inputting scenarios (play-in) via an application-specific GUI, and then executes the resulting program (play-out). This makes the model implicit as the model is created indirectly via the input scenarios. In a sense, we create a prototype via direct manipulation, but as the model of the system is created indirectly via the input scenarios it may be difficult to use the model for analysis and as basis for implementation of the final system. The reason is that an implicitly created model is difficult to interpret as it is automatically generated.

We have mentioned a number of libraries, all of which support animation in different ways. Using some libraries, animation is integrated with the modeling formalism, such as the use of timers in LTSA or the ability to view or change the marking of places in ExSpect. Some libraries are easy to extend, such as animations in LTSA, as the SceneBeans library allows users to easily extend it with new animation primitives. Also, animations created using Comms/CPN can easily be extended, as the "animation" is just a custom (Java) application. Some libraries make it easy to design animations, such as ExSpect and Mimic/CPN, which both provide a graphical user interface to design animations.

The approach of the current version of BRITNeY animation resembles a combination of Mimic/CPN and Comms/CPN, as the animation is driven by function calls associated with transitions to an external application. The main feature offered by BRITNeY from a user point of view is thus compatibility with CPN Tools (rather than the discontinued Design/CPN) and platform-independence. From a developer point of view, BRITNeY provides good foundations for allowing closer integration with the model by allowing parts of the animation to inspect and modify tokens on fusion places of the CPN model, much like how widgets are associated with places in ExSpect. This is an important part of future work.

An important new feature of BRITNeY animation is that it is possible to deploy animations in a way that allows even non-technical users to download and experiment with the animation. Another part of the future work is to make this process even easier by adding a wizard to take care of all the details.

BRITNeY animation is currently useful enough to use in real projects, and has, as mentioned in the introduction, already been used in two projects as part of the development.

References

1. Design/CPN. Online `www.daimi.au.dk/designCPN`.
2. C. Bossen and J.B. Jørgensen. Context-descriptive prototypes and their application to medicine administration. In *DIS '04: Proc. of the 2004 conference on Designing interactive systems*, pages 297–306, Boston, MA, USA, 2004. ACM Press.
3. S. Christensen, J.B. Jørgensen, and L.M. Kristensen. Design/CPN—A Computer Tool for Coloured Petri Nets. In *Proc. of TACAS'97*, volume 1217 of *LNCS*, pages 209–223. Springer-Verlag, 1997.
4. G. Gallasch and L.M. Kristensen. A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of 3rd Workshop on Practical Use of*

Coloured Petri Nets and the CPN Tools, volume PB-554 of *DAIMI*, pages 79–93. Department of Computer Science, University of Aarhus, 2001.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
6. D. Harel and R. Marelly. *Come, Let's Play*. Springer-Verlag, 2003.
7. Java Network Launching Protocol and API. `http://jcp.org/en/jsr/detail?id=56`.
8. K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts*. Springer-Verlag, 1992.
9. J.B. Jørgensen and K.B. Lassen. Aligning Work Processes and the Adviser Portal Bank System. In *REBNITA05*, 2005.
10. E. Kindler and C. Páles. 3D-Visualization of Petri Net Models: Concept and Realization. In *Proc. of ICATPN 2004*, volume 3099 of *LNCS*, pages 464–473. Springer-Verlag, 2003.
11. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
12. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer-Verlag, 2004.
13. L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. Accepted for Fifth International Conference on Integrated Formal Methods, 2005.
14. L. Lorentsen, A-P Tuovinen, and J. Xu. Modelling Features and Feature Interactions of Nokia Mobile Phones Using Coloured Petri Nets. In *Proc. of ICATPN 2002*, volume 2360 of *LNCS*, pages 294–313, 2002.
15. J. Magee and J. Kramer. *Concurrency – State Models and Java Programs*. John Wiley & Sons, 1999.
16. J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical Animation of Behavior Models. In *Proc. of 22nd International Conference on Software Engineering*, pages 499–508. ACM Press, 2000.
17. J.L. Rasmussen and M. Singh. *Mimic/CPN. A Graphical Simulation Utility for Design/CPN. User's Manual*. `www.daimi.au.dk/designCPN`.
18. J.L. Rasmussen and M. Singh. Designing a Security System by Means of Coloured Petri Nets. In *Proc. ICATPN 1996*, volume 1091 of *LNCS*, pages 400–419. Springer-Verlag, 1996.
19. SceneBeans. Online `www-dse.doc.ic.ac.uk/Software/SceneBeans`.
20. The ExSpect tool. `www.exspect.com`.
21. M. Weber and E. Kindler. The Petri Net Kernel. In *Petri Net Technologies for Modeling Communication Based Systems*, volume 2472 of *LNCS*, pages 109–123. Springer-Verlag, 2003.
22. M. Westergaard. Building Verifiable Software Prototypes using Coloured Petri Nets. Progress report, Department of Computer Science, University of Aarhus.
23. M. Westergaard. TIN-CPN, BRITNeY animation, and SClub website. Online `wiki.daimi.au.dk/tincpn`.