The BRITNeY Suite: A Platform for Experiments

M. Westergaard

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

Abstract. This paper describes a platform, the BRITNeY Suite, for experimenting with Coloured Petri nets. The BRITNeY Suite provides access to data-structures and a simulator for Coloured Petri nets via a powerful scripting language and plug-in-mechanism, thereby making it easy to perform customized simulations and visualizations of Coloured Petri net models. Additionally it is possible to make elaborate extensions building on top of well-designed interfaces to Coloured Petri nets created using CPN Tools.

1 Introduction

The Coloured Petri nets formalism [15] (CP-nets or CPNs) has been successfully used to model a lot of different systems, including network protocols [9, 10, 17, 18, 21], work-flows [2, 16], etc. The formalism is supported by CPN Tools [6, 22], a tool for editing and simulating CP-nets. As the CP-nets formalism has a lot of power, both computationally (in general, the question of whether a transition is enabled is not computable) and from a modeling standing point, a lot of research is made to improve analysis of the formalism (in particular experiments with different state space reduction techniques) and to use the formalism in different settings.

While CPN Tools is a great editor for CP-nets, it is not, in general, well-suited for experiments with the formalism, e.g. to test new language constructs or to integrate CPN models easily with other formalism or programs, for several reasons. Firstly, CPN Tools is closed source, which makes it difficult for people other than the developers to make modifications and extensions to the tool. Secondly, CPN Tools is written in the Beta programming language [20], which is not widely known, which makes it difficult to get started for an average programmer even if he had access to the CPN Tools source code as well as impossible to obtain off-the-shelf components for common tasks, e.g. chart-drawing, and programmers therefore have to build such components themselves. Finally, CPN Tools has no plug-in mechanism, which, in combination with the previous problems, makes it virtually impossible to experiment with CP-nets created using CPN Tools without great effort.

The BRITNeY Suite [23, 25] alleviates all of these problems by providing a Java-based, open source (GNU Public License, GPL), pluggable platform for

experimenting with CP-nets created in CPN Tools. Additionally the BRITNeY Suite comes equipped with a scripting language, which makes it easy to perform simple experiments. Earlier versions of the BRITNeY Suite have already been used in various projects, some of these are described in [16, 18, 19].

The contribution of this paper is not so much theoretical as it is a description of a platform. The reader is expected to have knowledge of CP-nets as implemented by CPN Tools as well as object-oriented programming, preferably in Java. The reader is assumed to be familiar with CPN Tools and the BRITNeY Suite (or have access to the BRITNeY Suite and try out the examples while reading).

In the next section we shall take a look at the architecture of the BRITNeY Suite and how it has evolved from a simple animation tool to a full bodied platform for experimenting with CP-nets. In Sect. 3 we shall take a look at how to create a simple script for running a customized simulation of a model downloaded from the Internet. In Sect. 4 we will take a look at how to extend the BRITNeY Suite with completely new functionality using the plug-in-mechanism, and, finally, in Sect. 5 we conclude.

2 A Brief History of the Architecture of the BRITNeY Suite

In order to fully understand the architecture of the BRITNeY Suite and how it can be used to perform experiments with CP-nets, it is beneficial to take a brief look at how the BRITNeY Suite evolved from a simple animation library to a full-blown platform for interacting with CP-nets. The walk through is historical, but also provides a top-down look at the current structure of the BRITNeY Suite, from the most abstract level to a more concrete level.

In Fig. 1 we see an abstract view of the internals of CPN Tools. We see that CPN Tools is in fact composed of two components, the CPN Tools GUI, to the left, and the CPN simulator, to the right. The GUI is responsible for editing CP-nets and sending them to the simulator, where they are checked for correctness and code is generated to simulate the model. The GUI communicates with the simulator using a proprietary binary protocol, which may change with new versions of CPN Tools. The uni-directional arrow indicates that the CPN Tools GUI must initiate any action (but data can be transferred in the other direction as responses to requests, e.g. the GUI may check a CP-net, and the simulator sends back data about how the check went). This architecture may seem a bit strange at first, but it has several advantages: Firstly, it is possible to write the editor in a language suitable for writing graphical user interfaces, and we can write the simulator in a language well-suited for implementing compilers. Secondly, we are able to run the editor on a single desktop PC while running the simulator on a server with a lot of memory and a powerful processor.

The BRITNeY Suite started as a simple animation library for use with CPN Tools. The library supported drawing and moving simple geometric figures on the screen. Originally the library was a simple application which exposed functions,

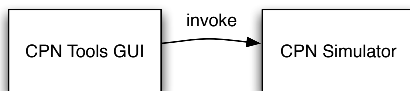


Fig. 1. The architecture of CPN Tools.

which could be called using a simple Remote Procedure Call (RPC) mechanism [5, Chap 5.3] based on the COMMS/CPN library [7]. A bit of code had to be loaded into the simulator in order for the simulator to be able to execute the remote procedures of The BRITNeY Suite. This architecture can be seen in Fig. 2, where we have the CPN Tools GUI and simulator, as before, on the left. These two comprise CPN Tools. Loaded into the simulator, we find some RPC stubs, which are able to call the procedures in the animation tool on the right. Here the CPN Tools GUI must initiate actions in the simulator (simulate a CP-net), and the simulator can then initiate actions in the BRITNeY Suite (animate objects). The animation library was written as a separate application in order to be able to use the powerful standard library of Java.

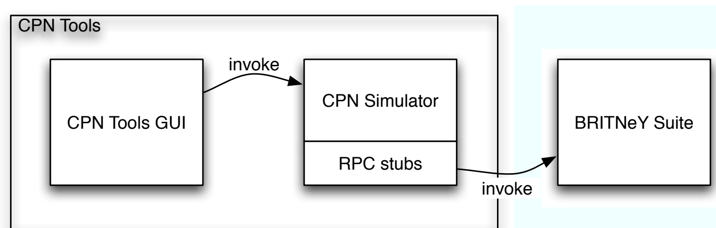


Fig. 2. The architecture of the first version of the BRITNeY Suite.

The next thing to happen was the introduction of so-called *animation objects*, objects representing an animation. By adding a new animation object, it is possible to extend the functionality of the animation tool, to add, e.g., functionality for drawing Gantt-charts. In order for CPN Tools to be able to communicate with the newly added animation objects, the user needed to generate stubs manually and load them into the simulator. This architecture can be seen in Fig. 3. The only change from Fig. 2 is that the tool is no longer locked to a single kind of animation, but contains interfaces, which can be used to extend the functionality.

After adding animation objects, the next step was to add *animation plug-ins* which are, like animation objects, responsible for creating an animation. The main difference between animation objects and animation plug-ins is that animation plug-ins are detached from the tool, can be developed outside the development tree of the BRITNeY Suite itself, and can be automatically downloaded from the Internet and loaded on runtime. Now the exposed features of the BRITNeY Suite are truly dynamic, and the old method of manually generating

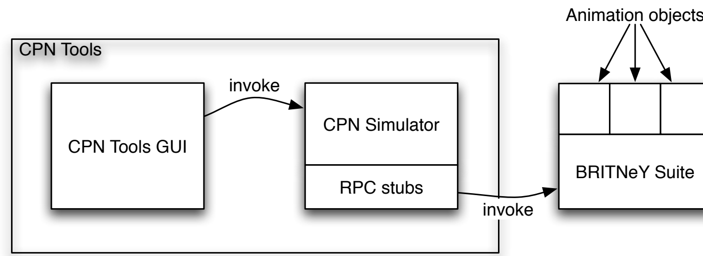


Fig. 3. The architecture of the BRITNeY Suite with animation objects.

and loading RPC stubs becomes infeasible in practice. Therefore an automatic stub-generator was introduced. The responsibility of this component is to generate and load the stubs into the CPN simulator. In order to facilitate this, the BRITNeY Suite needs to communicate with the simulator (and not just the other way around). Unfortunately, only one process can communicate with the simulator at any time. Therefore the BRITNeY Suite acts as a proxy for the communication from CPN Tools. The architecture can be seen in Fig. 4. Here the CPN Tools GUI is on the left, the BRITNeY Suite in the middle and the simulator on the right. The CPN Tools GUI no longer communicates directly with the simulator, but communicates with the *Simulator proxy* component of the BRITNeY Suite. Above the BRITNeY Suite main component, completely detached, lie the animation plug-ins. The *Stub generator* component uses Java reflection [12] to automatically generate stub code for each registered animation plug-in. This code is then loaded into the simulator whenever the CPN Tools requests the creation of a new simulator.

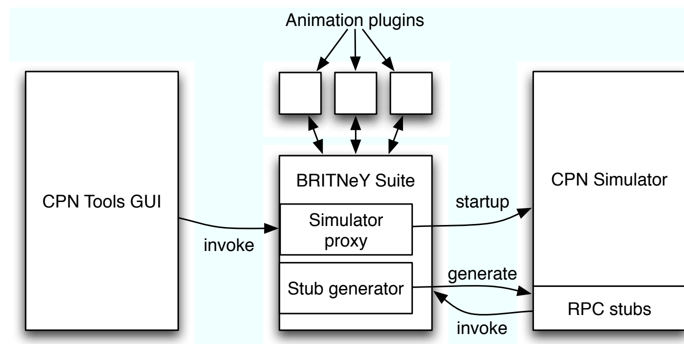


Fig. 4. The architecture of the BRITNeY Suite with animation plug-ins.

Having a very flexible plug-in architecture in place, a lot of refactoring took place. Firstly, the program was split up into a number of different plug-ins to make it easier to maintain and to provide versions that only support the required functionality. Secondly, the home-made RPC protocol was thrown away in favor

of a standardized open RPC protocol, XML-RPC [26]. These two changes made the tool much more open, as extensions can be written as simple plug-ins, and other tools can interface with the animation plug-ins using a simple standardized protocol. In Fig. 5 this can be seen. Here the BRITNeY Suite, in the middle, is split up into a *Simulator* plug-in, a *Simulator proxy* plug-in and an *Animation* plug-in with the Stub generator[1]. The plug-ins may depend on and communicate with other plug-ins, as indicated by the unlabeled bidirectional arcs (a plug-in depends on plug-ins below it connected by arcs). Animation plug-ins are normal plug-ins communicating with the plug-in Animation. They have been renamed to *Extension plug-ins*, as they can be used for more than just animation, e.g. a plug-in has been created to facilitate data-storage.

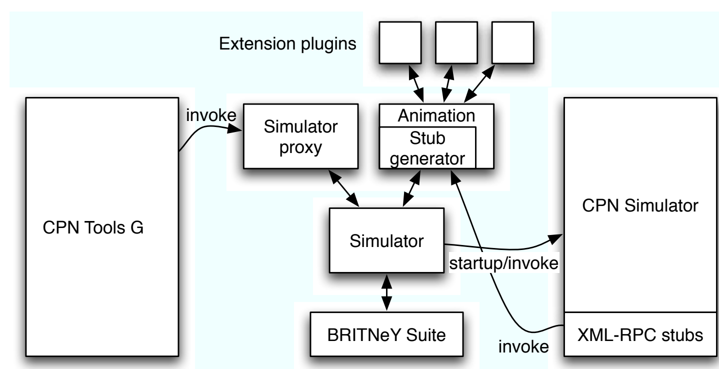


Fig. 5. The architecture of the BRITNeY Suite with general plug-ins.

The next thing to happen is that the BRITNeY Suite learns more about CP-nets. Originally, the tool had no need to know anything about CP-nets, as it was just a dumb RPC-server. In order to define extensions at a higher level, a simple CP-net model was introduced into the BRITNeY Suite. The model can be created by either snooping on the channel of the Simulator proxy (in order to syntax check and simulate a CPN model, all details of the model are transferred from CPN Tools, via the BRITNeY Suite, to the CPN Simulator) or by loading a net description from a PNML [11] file. Nets created with CPN Tools can also be loaded by translating them into PNML first (this is done automatically and transparently to the user). The net model can be used for various things. For example the model provides a high-level abstraction of Fusion places and transitions of the model, allowing extension plug-in writers to use a simple interface to the state and actions of the model. This can be used to create simple views of the model using Java.

[1] This is a simplification of the plug-in architecture of the BRITNeY Suite; in fact the BRITNeY Suite consists of nearly 20 plug-ins.

To sum up, the BRITNeY Suite provides a platform for experimenting with CP-nets created using CPN Tools for the following reasons:

- it is distributed under an open source license (GPL),
- it is written in a well-known language (Java), enabling the reuse of off-the-shelf components,
- it has a pluggable architecture,
- it has abstractions of the CPN simulator on several levels (from direct access to the simulator's interface up to an executable CPN object model).

3 Scripting Engine

Even though the BRITNeY Suite comes equipped with a powerful plug-in system, the full power of this is often not needed. In order to allow simple tasks to be performed, the BRITNeY Suite employs a simple scripting engine, which, while simple, is powerful, as most of the internals of the BRITNeY Suite are exposed through this engine.

The scripting engine used is BeanShell [1], which allows programmers to use a Java-like syntax to create macros (in fact Java syntax is allowed, but so is a more relaxed version). A complete tutorial on using BeanShell is out of scope of this paper, but the provided example should be a good starting point for writing your own macros.

The example assumes we have created a CPN model with a nice animation using the approach described in [24], which shows how to create CPN models, using animations created for the BRITNeY Suite, which can be executed without the help of CPN Tools. Basically, you have to do all initialization of the animation using an init-transition, and use the Save simulator as-tool from the Simulator-menu, to export a simulator-image, which we have uploaded to our web-server, `http://www.example.org/`, as `simulator.x86-win32`. We now want is to write a script which allows us to run a simulation a number of times. To make the simulation more personal to the user, we will ask him for his name and use this during the simulation. The created such that we need to execute 100 steps of the simulation with 100 ms delay between steps. We want the user to be able to configure the number of runs using the built in option pane in the BRITNeY Suite. To make execution easy, we will add a new menu item to start the simulation. In order to do this, we will use some exported objects and some utility classes provided by the BRITNeY Suite. Exported objects allow the programmer to manipulate the internal data-structures of the BRITNeY Suite and utility classes allow the programmer to add entries to the menu for evaluating scripts.

The rest of this section is structured as follows: First we will take a brief look at some of the exported objects and their use (thereby adding the functionality described above) and then we will look at some of the utility classes and how to use them (thereby adding a menu item for our new feature).

3.1 Exported Objects

The BRITNeY Suite, at the time of writing, exports 6 objects for use in the scripting language. The objects are Options, MessageDisplay, ToolModel, SimulatorService, Tool, and IndexModel. In addition to this, all commands of the tool (i.e. actions that can be selected from the menus) are exported as well.

In this example, we shall use the first four exported objects (the two other are not used that often) and one of the exported commands. The Options exported object can be used to add new pages to the options facility of the tool, MessageDisplay can be used to display messages and progress-bars in the status-line. ToolModel can be used to get access to low-level parts of the tool, in particular a reference to the main window, which can be used to create modal dialog windows. The SimulatorService can be used to instantiate or load CP-net simulators. The IndexModel exported object can be used to remove items from the menu of the applications and the Tool object gives access to the internal objects of the BRITNeY Suite, but most of these are already exported into the BeanShell engine, and Tool is therefore rarely used.

The code in Listings 1 and 2 implements the new feature. Listing 2 contains the actual implementation and Listing 1 contains standard Java code defining a new class, Runs, with a single field (count) and two accessor methods, getCount and setCount, coded using the guide lines in [13, Chap. 7: Properties] (basically a property, foo must have getter and setter methods called `getFoo` and `setFoo`).

Listing 1 BeanShell code to implement the Runs class.

```
1  public class Runs {
2    int count = 3;
3
4    public int getCount() {
5      return count;
6    }
7
8    public void setCount(int count) {
9      this.count = count;
10   }
11 }
```

Line 1 in Listing 2 creates an instance of our newly created class. Note that in the BeanShell language, we do not need to specify the type of the variable. In line 2 we use a method of the Options object to add a new managed object for options. The first parameter is a descriptive name, the second is a reference to the object we want to be managed. The last parameter is a so-called stop-class, which can be used to restrict what variables are shown in the options pane. Normally Object.class is fine[2].

[2] Check the documentation for `java.beans.Introspector#getBeanInfo(java.lang.Class,java.lang.Class)` for more information.

Listing 2 BeanShell code to implement multiple runs of an animation.

```
runs = new Runs();
Options.addManagedObject("Runs", runs, Object.class);

void runSimulation() {
  MessageDisplay.showProgress("Loading", 0, 2);
  mainWindow = ToolModel.getMainWindow();
  MessageDisplay.showProgress("Loading", 1, 2);
  name = javax.swing.JOptionPane.showInputDialog(mainWindow,
    "What is your name");
  MessageDisplay.showProgress("Loading", 2, 2);
  MessageDisplay.showMessage("Hello " + name, 0);

  s = SimulatorService.getNewSimulator(
    new java.net.URL("http://www.example.org/simulator.x86-win32"));
  hs = s.getHighLevelSimulator();

  oldsteps = Play.steps;
  olddelay = Play.delay;
  Play.steps = 100;
  Play.delay = 100;

  hs.evaluate("name := \"" + name + "\"");

  for (int i = 0; i < runs.getCount(); i++) {
    hs.initialState();

    event = new java.awt.event.ActionEvent(hs, 0, "");
    Play.execute(event);
  }

  Play.steps = oldsteps;
  Play.delay = olddelay;
}
```

The rest of the code is the actual implementation of the new functionality, and consists of a single method. We notice that in BeanShell methods can exist outside of a class. Lines 5, 7, and 10 provide a short progress-bar using MessageDisplay. The showProgress method takes 3 parameters: a descriptive text, how much has been completed, and how much do we need to complete. Lines 6, 8, and 9 take care of showing a dialog box querying the user for his name. In line 6 we obtain a reference to the main window of the application, which is used to display a modal dialog to the user using JOptionPane in lines 8 and 9. In line 11 we show a friendly greeting to the user in the status bar using the showMessage method of MessageDisplay. The method takes two arguments, the text to display, and a message type. The message type is no longer used and should just be set to 0.

In lines 13–15 we obtain a reference to the simulator that we have uploaded to our web-server using SimulatorService. First we call getNewSimulator, which takes as parameter a URL pointing to the location of the simulator, and then we use the obtained simulator to obtain a HighLevelSimulator, which encapsulates a lot of functionality.

Lines 17–20 will be explained later. In line 22 we store the name of the user, which we have saved in the name variable in the simulator. We simply evaluate ML-code corresponding to assigning the value of the BeanShell variable to a ML reference. This code assumes that we have added a declaration `val name = ref "dummy";` in our CPN model.

In line 24 we start the actual loop. This is a standard Java for loop, which makes use of the getCount accessor method of the runs variable we declared in Listing 1. If the user changes the setting in the options pane, this will automatically be reflected, and the loop will run more than the default 3 iterations.

In line 25 we reset the simulator to the initial state. This is performed by calling a method on the HighLevelSimulator we obtained earlier.

Lines 27 and 28 take care of executing the simulation using the Play command. First, in line 27, we create an event to send to the command. The ActionEvent class takes as first parameter the source of the event. The Play command works on HighLevelSimulators, so we give this as parameter. The two last parameters are not used and should just be set as in the example. Finally, in line 28, we execute the Play command. In order to run the simulation with the desired settings (100 steps, at least 100 ms delay between transitions), we prepare our use of the Play command in lines 17–20. We need to set the options of this command, but as these are persistent, we first save the old values (which we restore later in lines 31–32), and then set the new values to execute 100 steps with a minimum delay between transitions of 100 ms.

After evaluating the code from Listings 1 and 2 in the Script Console, we can change the value of the runs parameter in the Options dialog (see Fig. 6) and start the simulation by evaluating `runSimulation()` in the Script Console. This procedure may not be very user-friendly, so let us immediately turn to creating menu items for our new feature using some utility classes.

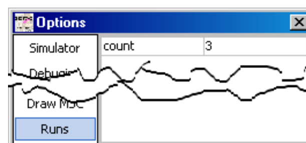


Fig. 6. The Options dialog for our newly created option, Runs. The middle of the dialog has been cut out.

3.2 Important Utility Classes

Using the BeanShell scripting engine, users have unlimited access to all classes available in Java and the BRITNeY Suite. Here we shall only treat two of them (actually we have already used a couple of other classes provided by both the BRITNeY suite and Java earlier in this section, namely the Simulator, HighLevelSimulator, JOptionPane, ActionEvent, and URL classes). The two classes we shall need are the ScriptingCommand and ScriptingTools classes. These classes allow us, from the BeanShell, to create new commands, which can be added as items in new menus.

In the BRITNeY Suite menus are represented using tool boxes. One implementation of a tool box is provided by the ScriptingTools class. Tool boxes contains commands, which can be executed. The ScriptingCommand provides one such implementation, which is able to execute BeanShell code. We shall use this to add a "Demo" menu to our application. This menu shall have one entry, namely "Start simulation", which will call the method we defined earlier in this section.

In Listing 3 we see how to create a ScriptingCommand in lines 1–4. The first parameter is a short descriptive name, which is shown in the menu. The second parameter is a longer description, which is shown as tool-tip when the command is shown in the tool-bar. The third and final parameter is the code to execute. Any BeanShell code can appear here, but quotes (") must be escaped (as \"), so it will often be easier to write a method for the feature and then just call the method from the ScriptingCommand, just like we have done in this example.

Listing 3 BeanShell code to add a new menu with an item to execute the method defined in Listing 2.

```
1 sc = new dk.klafbang.tincpn.scripting.ScriptingCommand(
2   "Start simulation",
3   "Runs the number of simulations specified in the options",
4   "runSimulation()");
5 tb = new dk.klafbang.tincpn.scripting.ScriptingTools("Demo");
6 tb.add(sc);
```

In Fig.7 the resulting menu and menu-item is shown. When the item is selected, the code from Listing 2 is called, starting by asking the user for his name and then running (in this case) 3 simulations of the model.

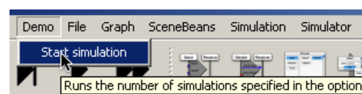


Fig. 7. A menu generated using the script code in Listing 3.

3.3 Other Possibilities

In this section we have seen a simple example of how to run a customized simulation of a CPN model with an associated animation. The example uses a lot of the features in the BRITNeY Suite. There is, however, an important feature, we have not touched at all, namely the ability to inspect and control the simulation of a CPN model. This can be used for a lot of things, such as altering the state of the model or controlling how/in what order transitions are executed.

As a simple example, let us see how we can implement prioritized transitions with the BRITNeY Suite. This can of course be done by altering the model, but for real models, this is not very easy and unnecessarily clutters the model. We will look the net in Fig. 8. Here we want to transport the single token of type UNIT from the Start place to the End place, via the P1 and P2 places. Transport can be done either by the transitions A# or B# where $\# \in \{1, 2, 3\}$. We furthermore put a token on the A (resp. B) place whenever a A# (resp. B#) transition fires. We want A# transitions to have priority over B# transitions (i.e. when we are done, we want two tokens on A and one token on B).

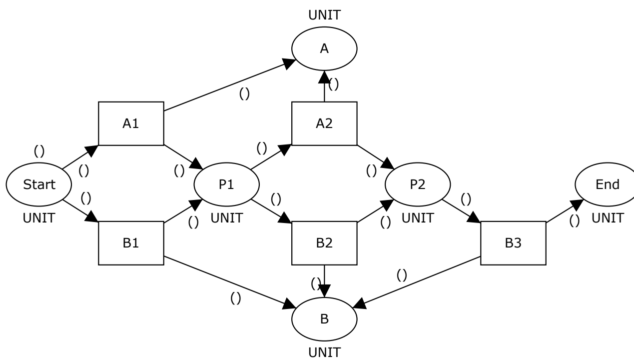


Fig. 8. A simple CP-net. We want all A transitions to have priority over B transitions.

When we simulate the net in Fig. 8 in CPN Tools, we will often get more than one token on B, indicating that A# transitions do not have priority over B# transitions. We can fix this problem with a bit of scripting in the BRITNeY Suite. Basically, we have to write our own scheduler, which prefers A# transitions over B# transitions. Although this sounds like a complicated task, it is quite simple as can be seen by the code in Listing 4. The code first loads the net (ll. 1–2), extracts an abstract description of the net (l. 4). In addition to the abstract description of the net (the netmodel), the triple returned in line 1 also contains an abstract description of how the net is displayed (the second component, a viewmodel, which which can be used to change how the net is presented to the user—we will not use this feature in this example to keep things simple), and a CPN checker process (the third component, a netchecker), from which we can obtain a CPN simulator object.

After loading the model, two lists are constructed, one for high-priority transitions, and one for low-priority transitions (ll. 6–19). The lists are constructed by traversing all instances (l. 10) of all transitions (l. 9) on all pages of the net (l. 8), and if the transition name starts with an A (l. 11) add it to the list of high-priority transitions (l. 12) and otherwise add it to the list of low-priority transitions (l. 14). After that we obtain the simulator for the net (ll. 19–20), reset the simulator (l. 22), initializes the step counter and the maximum number of steps to take (l. 23) and starts the simulation (ll. 24–29). The simulation updates the progress bar (l. 25), delays (l. 26), and calculates whether it was possible to execute any transitions (l. 27). This calculation is performed by executing an A# transition if one is enabled and otherwise a B# transition (the right-hand side of the or expression is not evaluated if the left-hand side evaluates to true). Then the step counter is incremented (l. 28). When the simulation is done, the progress-bar is reset (l. 30). Even though the script is simple, it will do the task, and it even keeps track of progress and reacts to changes to the global options of the BRITNeY Suite.

We have now seen how we can use inspection and control over the execution of CPN models to support priority of transitions by gathering all high-priority transitions in one list and all other transitions in another, and then execute transitions from the first list before transitions from the other list. In a similar way, we can also simulate inhibitor arcs [3] by creating arcs with the inscription empty where we intend inhibitor arcs. This arc will never prevent the transition from being enabled, so if we interpret the arc as an inhibitor arc, it will be a restriction. When we search for a transition to execute, we will simply omit all transitions with "inhibitor"-arcs if any place in the other end is marked with tokens. The final application of this, we will look at, is transition fusion [4] (or synchronous channels) with no data-transfer. Here a transition, e.g., A!, is enabled iff another transition, A?, is enabled. If we ignore all !-transitions where no corresponding ?-transition is enabled (and vice versa), and always execute an ?-transition immediately after a !-transition, we have implemented transition fusion with no data-transfer. We can implement transition fusion with data-transfer by also inspecting the enabled bindings, but this is more complicated.

We can use the ability to alter the state to communicate with external processes in an asynchronous way, for example we can implement fusion places between two independent nets by running the simulations in parallel and synchronizing the tokens on all shared places between transitions. We can also combine the place inspection features with the transition selection features to implement bounded places (by inspecting the effect of executing a binding element on all bounded places and disallowing the execution if it violates the bound) and FIFO (first-in-first-out) places (by keeping track of when tokens arrive, and only allow the execution of a binding element if it consumes tokens in the correct order).

These examples are just appetizers of what can be accomplished relatively easily, using few lines of scripting code. The fact that well-known concepts can be realized with such ease, even though the interface was not designed for that purpose initially, makes it believable that completely new concepts can be real-

Listing 4 Code implementing a scheduler, which prefers A# transitions over B# transitions.

```
triple = LoadNet.loadNet(new java.io.File("c:/contention.cpn"));
LoadNet.registerIndexNode(triple);

netmodel = triple.getFirst();

As = new ArrayList();
Bs = new ArrayList();
for (page : netmodel.getPages()) {
  for (transition : page.getTransitions()) {
    for (instance : transition.instances()) {
      if (transition.getName().startsWith("A"))
        As.add(instance);
      else
        Bs.add(instance);
    }
  }
}

netchecker = triple.getThird();
simulator = netchecker.getSimulator();

simulator.initialState();
i = 0; max = Play.steps;
do {
  MessageDisplay.showProgress("Executing", i, max);
  Thread.sleep(Play.delay);
  executed = simulator.fireAny(As) || simulator.fireAny(Bs);
  i++;
} while (executed && i < max);
MessageDisplay.showProgress("Executing", max, max);
```

ized relatively easily as well. As already mentioned, these features can of course be realized directly by changing the model, but the idea behind introducing more elaborate constructs is to simplify the model. Using the BRITNeY Suite, we are able to experiment with new constructs before they are integrated into CPN Tools.

4 Extension Plug-ins

In this section, we will look at an example of how to create a simple extension to the BRITNeY Suite. This consists of three steps: First we write one or more Java classes implementing the new feature, then we write a plug-in-descriptor describing how the plug-in should be loaded, and finally we package our plug-in and deploy it.

Throughout this section we will develop a simple plug-in that allows us to play sounds. We will add a new menu, Sound, to the GUI of the BRITNeY Suite add a menu item which will play a sound through the computer's speakers, and register a new extension plug-in, which can be used to allow CPN models to play sound during the execution. This example also illustrates the reason for renaming animation plug-ins to extension plug-ins, as a sound player is not a visualization, but just a new feature, we can add to the toolbox available from CPN Tools.

4.1 Writing a Java Implementation

Writing Java classes is out of scope of this paper, but it may be instructional to know of a few good hooks into the BRITNeY Suite. For more detailed information, refer to the on-line source code documentation, which can be found at `http://www.daimi.au.dk/~mw/local/tincpn/doc/`. In the following we will refer to packages and classes within this documentation.

We will start by creating a class for playing sounds. This is just regular Java code, and has nothing to do with the BRITNeY Suite. The code relies on the Java Applet code to play sounds (even though a better way has found its way into Java in later versions, but this is much more verbose). The class is able to load sound clips and store them for later playback (the loadSound method) as well as playing back previously loaded sound clips (the playSound method). The code can be seen in Listing 5.

The most important hook is probably the ToolBox plug-in, which contains classes that make it possible to add new menus, menu items and corresponding actions. The plug-in is implemented in the dk.klafbang.tincpn.tools package, and contains among other items the classes Command and AbstractToolBox, which can be used as a basis for adding new commands that can be executed when an item is selected from a menu and for adding new menus respectably.

The implementation of the command for playing sounds can be seen in Listing 6. We start by importing the classes we will need (ll. 1–4). We then create a new class inheriting from the Command class (l. 6). The constructor (ll. 9–12) calls the constructor with a name to show in the menu and a tool tip. The super class constructor can optionally take a third parameter, the location of an icon for the command. We also create an instance of the SoundPlayer class we created above (ll. 7 and 11). In the execute method (ll. 14–20), the actual action takes place. We start by calling the execute method of the super class (l. 15). We then use the built-in class, JFileChooser, to show a file dialog, which allows the user to select a file to play. If the user selects a file and presses Ok, the sound is played using the SoundPlayer instance (l. 18–23) by first converting the the file selected by the user into a string representation of the URL pointing to the file (l. 19), loading the sound clip (l. 20), and finally playing back the clip (l. 21). As all of the methods may fail, we have encapsulated the sound playing code in an exception handler (ll. 18,22–23).

We would also like to be able to show the newly created command in a new menu. We do this by sub-classing the AbstractToolBox as seen in Listing 7. The

Listing 5 A simple class for playing back sound in Java.

```
import java.applet.*;
import java.net.URL;
import java.util.*;

public class SoundPlayer {
  Map<Integer, AudioClip> clips;
  int counter = 0;

  public SoundPlayer(final String name) {
    clips = new HashMap<Integer, AudioClip>();
  }

  public synchronized int loadSound(final String url) throws Exception {
    final URL location = new URL(url);
    final AudioClip audioClip = Applet.newAudioClip(location);
    clips.put(counter, audioClip);
    return counter++;
  }

  public void playSound(final int key) throws Exception {
    clips.get(key).play();
  }
}
```

code is very simple. The constructor calls the super class constructor with the name of the new menu (l. 5) and adds a new SoundPlayerCommand (l. 6). We can add as many commands as we want.

Other hooks that may be interesting include the net model (in the package dk.klafbang.tincpn.nets.model, in the plug-in NetModel) and the simulator interface (the classes Simulator, HighLevelSimulator, and SimulatorService in the package dk.klafbang.tincpn.simulator in the plug-in Simulator). We have already seen how to use these from the scripting language in the previous section, but we can of course also use them from plug-ins.

4.2 Writing a Plug-in-descriptor

In order to be able to use our newly created plug-in, we need to tell the BRITNeY Suite how to use it. To do this, we will need to create a plug-in-descriptor. The descriptor describes where the plug-in code can be found, what plug-ins the plug-in relies on, and, most importantly, how the plug-in should be integrated.

Plug-ins in the BRITNeY Suite are created using the Java Plug-in Framework [14]. This means a plug-in-descriptor has to be written using an XML-file. The complete specification of the plug-in-descriptor format can be found at `jpf.sourceforge.net/dtd.html`, but it should be quite easy to infer the general format from this example.

Listing 6 A command for playing sounds.

```
import java.awt.event.ActionEvent;
import javax.swing.JFileChooser;
import static javax.swing.JFileChooser.APPROVE_OPTION;
import dk.klafbang.tincpn.tools.Command;

public class SoundPlayerCommand extends Command {
  SoundPlayer player;

  public SoundPlayerCommand() {
    super("Play sound", "Plays a sound");
    player = new SoundPlayer();
  }

  public void execute(final ActionEvent event) {
    super.execute(event);
    JFileChooser chooser = new JFileChooser();
    if (chooser.showOpenDialog(null) == APPROVE_OPTION) {
      try {
        final String url = chooser.getSelectedFile().toURL().toString();
        final int key = player.loadSound(url);
        player.playSound(key);
      } catch (final Exception e) {
      }
    }
  }
}
```

Listing 7 A toolbox for playing sounds.

```
import dk.klafbang.tincpn.tools.AbstractToolBox;

public class SoundTools extends AbstractToolBox {
  public SoundTools() {
    super("Sound");
    add(new SoundPlayerCommand());
  }
}
```

The entire plug-in-descriptor for our sound plug-in can be seen in Listing 8. The first four lines states that this is a plug-in-descriptor.

Line 4 identifies this plug-in to the surroundings. The id must be unique and the version should reflect the version of the plug-in.

Next, in lines 5–8 follow the dependencies section. We state that we depend on the ToolBox plug-in (to create Commands and AbstractToolBoxes) and the AnimationTools plug-in (to register our extension plug-in).

This is followed by the runtime section (ll. 10–17), which describes where to find the code. In this case the runtime consists of two locations, a code and a

resources location. Each library declaration consists of an id (a unique descriptor of the library) a path where the code can be found. In this case this is just ".", i.e. the code can be found in the same location as the plug-in descriptor itself. The path can also point to another directory, a jar-file or even a web-location. You can have more than one library declaration for each plug-in. This allows you to use third party jar-files in your plug-in.

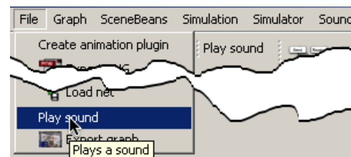
Listing 8 The plug-in-descriptor for the Sound plug-in.

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 0.3"
  "http://jpf.sourceforge.net/plugin_0_3.dtd">
<plugin id="Sound" version="1.0.0">
  <requires>
    <import plugin-id="ToolBox"/>
    <import plugin-id="AnimationTools"/>
  </requires>

  <runtime>
    <library id="Sound.code" path="." type="code">
      <export prefix="*"/>
    </library>
    <library id="Sound.res" path="." type="resources">
      <export prefix="*"/>
    </library>
  </runtime>

  <extension plugin-id="ToolBox" point-id="ToolBox"
      id="SoundTools">
    <parameter id="class" value="SoundTools"/>
  </extension>
  <extension plugin-id="ToolBox" point-id="NewCommand"
      id="SoundPlayerNewCommand">
    <parameter id="class" value="SoundPlayerCommand"/>
  </extension>
  <extension plugin-id="ToolBox" point-id="LoadCommand"
      id="SoundPlayerLoadCommand">
    <parameter id="class" value="SoundPlayerCommand"/>
  </extension>
  <extension plugin-id="ToolBox" point-id="AdditionalCommand"
      id="SoundPlayerCommand">
    <parameter id="toolbox" value="Files"/>
    <parameter id="class" value="SoundPlayerCommand"/>
  </extension>
  <extension plugin-id="AnimationTools" point-id="Animation"
      id="SoundTools">
    <parameter id="class" value="SoundPlayer"/>
  </extension>
</plugin>
```

After this follows the section where your plug-in is tied to the BRITNeY Suite. This is done using so-called *extension points*. The BRITNeY Suite comes pre-configured with a number of extension points. This plug-in extends all five currently available extension points to show how it is done. Each extension point is specified using a plugin-id, the id of the plug-in specifying the extension point, a point-id, the id of the point to extend, and an unique id for the extension. The first extension (ll. 19–22) adds a new tool box (menu) to the tool. It takes a single parameter, namely the class implementing the tool box. In this case this is the SoundTools class. The next two extensions (ll. 23–30) are related and add commands to the marking menu on the desktop. The first adds a "new"-command, i.e. a command for creating new things (e.g. creating a new net or starting a new simulator etc.). The second extension adds a "load"-command, i.e. a command for loading things (e.g. loading a net or loading a simulator). The fourth extension (ll. 31–35) is for smaller plug-ins, which do not require their own top-level menu, but which wants to add a command to an existing menu, here the Files menu. The final extension (ll. 36–39) adds a new extension plug-in. We just add our utility-class as the extension plug-in, and nothing extra needs to be done. Now, if we start the BRITNeY Suite and CPN Tools, we will see an extra menu for playing sounds in the BRITNeY Suite (Fig. 9(a) at the top-right corner), we will see an extra item in the Files menu (Fig. 9(a) to the left) and extra entries in the Load (Fig. 9(b)) and New (Fig. 9(c)) marking menus. Furthermore we can use our extension plug-in in CPN Tools, e.g. as in Listing 9.



(a) The modified menu bar. The middle of the figure has been cut out.

(b) The modified Load marking menu.

(c) The modified New marking menu.

Fig. 9. A menu and two marking menus generated by the Sound plug-in.

This example has shown how to add completely new functionality to the BRITNeY Suite and tie it into the tool in various places. Often we will not tie it into the tool in as many places as we have done here, but e.g. only register a class as an animation extension (thereby creating an extension plug-in), or only

Listing 9 Code to instantiate and use our newly created extension plug-in

```
1 structure sound = SoundPlayer(val name = "Ignored");
2 val _ = sound.playSound "c:/sound.au"
```

add an entry to the menu, e.g. to create new editing facilities. It is also possible to add new extension points to the tool, but this is out of the scope of this paper. As already mentioned, most of the BRITNeY Suite is actually implemented as plug-ins in a very small core application; for example the visualization facility, the previous core functionality of the BRITNeY Suite, is just a plug-in, and can be removed if desired.

5 Conclusion

In this paper we have seen how the architecture of the BRITNeY Suite supports experimentation with CP-nets for people without access to the source of CPN Tools. The BRITNeY Suite is distributed under an *open source license*, it is *written in a well-known programming language*, namely Java. The BRITNeY Suite has a *pluggable architecture*, which provides *abstractions of the CPN simulator* as well as a powerful *scripting engine*.

We have seen how to create simple extensions using the scripting engine and how to provide completely new functionality by creating a new extension plug-in.

Future work includes making the CPN abstraction more Java-like, and e.g. provide an *Observable* [8][pp. 293-304] abstraction of places, so it is possible to be notified whenever the marking on a place or a fusion group changes. Other than that, current and future research include experimentation with variations of CP-nets using the abstractions provided by the BRITNeY Suite.

References

1. The BeanShell Scripting Language. Java Specification Requests, JSR: 274, `http://jcp.org/en/jsr/detail?id=274`.
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. G. Chiola, S. Donatelli, and G. Franceschinis. Priorities, Inhibitor Arcs and Concurrency in P/T nets. In *Proc. of ICATPN 1991*, pages 182–205, 1991.
4. S. Christensen and N.D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. In *Proc. of ICATPN 1994*, volume 815 of *LNCS*, pages 159–178. Springer, 1994.
5. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Weslay, 3rd edition, 2001.
6. CPN Tools. `www.daimi.au.dk/CPNTools`.
7. 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.

8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
9. S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proc. of ICATPN'02*, volume 2360 of *LNCS*, pages 182–202. Springer-Verlag, 2002.
10. B. Han and J. Billington. Formalising the TCP Symmetrical Connection Management Service. In *Proc. of Design, Analysis, and Simulation of Distributed Syste ms*, pages 178–184. SCS, 2003.
11. Combined WD Circulation, CD Registration and CD Ballot, WD 15909-2, Software and Systems Engineering, High-level Petri Nets – Part 2: Transfer Format. version 0.9, `wwwcs.uni-paderborn.de/cs/kindler/publications/copies/ISO-IEC15909-2-WD0.9.0.Ballot.pdf`, 2005.
12. The Java^{TM} Tutorial: The Reflection API. `java.sun.com/docs/books/tutorial/reflect/`.
13. JavaBeans 1.01 specification. `java.sun.com/products/javabeans/docs/spec.html`.
14. Java Plug-in Framework. `jpf.sourceforge.net/`.
15. K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts*. Springer-Verlag, 1992.
16. J.B. Jørgensen and K.B. Lassen. Aligning Work Processes and the Adviser Portal Bank System. In *REBNITA05*, 2005.
17. 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.
18. L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. In *Proc. of Fifth International Conference on Integrated Formal Methods*, volume 3771 of *LNCS*, pages 266–286. Springer-Verlag, 2005.
19. R.J. Machado, K.B. Lassen, S. Oliveira, M. Couto, and P. Pinto. Execution of UML Models with CPN Tools for Workflow Requirements Validation. In *Proc. of Sixth Workshop and Tutorial om Practical Use of Coloured Petri Nets and the CPN Tools*, volume PB-576 of *DAIMI*, pages 231–250, 2005.
20. O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison Wesley, 1993.
21. C. Ouyang and J. Billington. On Verifying the Internet Open Trading Protocol. In *Proc. of 4th International Conference on Electronic Commerce and Web Technologies*, volume 2738 of *LNCS*, pages 292–302. Springer-Verlag, 2003.
22. A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ICATPN 2003*, volume 2679 of *LNCS*, pages 450–462. Springer-Verlag, 2003.
23. M. Westergaard. BRITNeY Suite website. `wiki.daimi.au.dk/britney/`.
24. M. Westergaard and K.B. Lassen. Building and Deploying Visualizations of Coloured Petri Net Models Using BRITNeY animation and CPN Tools. In *Proc. of Sixth Workshop and Tutorial om Practical Use of Coloured Petri Nets and the CPN Tools*, volume PB-576 of *DAIMI*, pages 119–136, 2005.
25. M. Westergaard and K.B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ICATPN 2006*, volume 4024 of *LNCS*, pages 431–440. Springer-Verlag, 2006.
26. D. Winer. XML-RPC Specification. `www.xmlrpc.org/spec`.