JFern Manual

Mariusz Nowostawski

Version 4.0.0 August, 2009

1 What is a JFern kernel

Java Fern (Microsorum pteropus), is a beautiful and commonly grown plant. Originating from Southeast Asia, it reproduces easily, and the reproduction is quite interesting. New plantlets are born on the edge of the leaves of the mother plant and grow right there, clinging on it with their little roots. However, JFern or JFern kernel is also a working name for a Java-based, object-oriented Petri net simulation framework. Fernlets are nets which are JFern-based, designed and generated by that package.

The package has the following features:

- small footprint; the kernel itself, without GUI or external dependencies, contains only 41 classes, and is 130kB bytecode size, perfect for Personal Java and PDAs;
- compiled, native network representation, which provides really fast (and we mean as good as one can get in Java) Petri net execution;
- ability to embed arbitrary `Java` (kawa, clojure and bsh) code inside guards and multiset expressions;
- built-in six major Petri-net design patterns
- separation of the net representation from its visualization and from the marking; this provides easy net evaluation with different initial markings; it also allows better debugging, breakpointing and intermediate net states dumping
- XML read and dynamic net generation support for a network structure
- integration with the graphical package for editing, simulation visualizations and nets debugging
- implementation of simulated time, timed simulator and timed tokens

The package will include in the future releases following features:

- more CPN design patterns;
- XML-based storage support for a marking;
- more efficient internal occurrence set representation and more efficient unification algorithm for even better performance and better memory efficiency.

2 Introduction

2.1 Version

The current version is marked as 4.0.0, and this is fourth major stable release of JFern package. It can be used in research projects and in production systems as a runtime coloured Petri net engine. Release 3 added graphical support and in its current stage is suitable for net modelling and debugging purposes. Version 4 added new features to the graphical editor, and provided inscription capabilities in a number of scripting dynamic languages: bsh (BeanShell), Kawa (Scheme variant) and Clojure (Lisp variant).

The version of the documentation always relates to the appropriate version of the JFern kernel package. The versions of this package are organized according to Major.Minor.Age versioning schema. The first number represents the major, the public API version number, the second is the patch number to the current API version, which does not introduced any new public API features and the third number is the age, of how long a given version is backward compatible. Current version 4.0.0 due to major rework on the public API level, and additional features is not compatible with the previous line 2.xx or 3.xx, thus the last, third number in the version got reset to 0.

2.2 License

The software and the documentation is released under the LGPL license together with the sources, and is copyrighted by the appropriate parties. You may use the package free of charge for commercial and non-commercial purposes. You can also redistribute it only if an appropriate copyright notice and disclaimer are provided. Please refer to the distribution package for the actual license agreement. All changes you do to the codebase shall be reported back to the original authors and copyright holders. You may simply send them to the project maintainer.

2.3 Feedback

Please send all your queries related to the project directly to mariusz@nowostawski.org. For a discussion forum related to the JFern kernel project subscribe to the JFern mailing lists on JFern SourceForge page

We will to read and incorporate all suggestions related to the documentation, and we will be very pleased to receive any feedback and bug reports.

Please, help us to make this documentation more useful and the package more robust by sending any documentation related comments and package suggestions directly to package maintainer.

2.4 Credits

Please check `AUTHORS` and `THANKS` files for names and e-mail addresses of all helpers and contributors. This project would not be possible without your hard work and help. Keep it up!

Special thanks to Martin Fleurke for his awesome work on NetToolbox and JFern GUI.

3 Quick start

To startup the JFern NetToolbox, you can simply type:
`java -jar rakiura-jfern.jar`
in your command line. Alternatively, you can place all classes from the `lib` directory into your `CLASSPATH` and call:
`java org.rakiura.cpn.gui.NetToolboxApplication`.

4 Modular architecture

JFern is compliant with Java Modules Management System called RAK. Please refer to the RAK system documentation at RAK SourceForge page.

To install a JFern module one needs to have `tools.jar`, which are distributed with the Sun JDK, and are usually placed inside `lib` directory of your JDK distribution. You should place `tools.jar` in the extentions (`ext`) directory of your JDK, or add it to your CLASSPATH. Additionally, JFern module relies on the Graham Kirby dynamic compiler package. When installed via RAK system the compiler will be installed automatically, otherwise the appropriate JAR file must be downloaded and placed in the CLASSPATH. graham-kirby jar file is part of the standard JFern distribution. If you plan to use the GUI and visualisation module, you have to obtain also the copy of the Rakiura JRio, a drawing package. You can get it at http://sf.net/projects/cirrus - download JRio module. rakiura-jrio jar file is also part of standard JFern distribution.

Three additional jar files are needed if one of the following languages is to be used for the inscriptions: kawa (Scheme), Clojure (Lisp variant) or BeanShell (bsh). All these languages are supported in the inscriptions (with varying level of testing), and appropriate jar files are distributed together with the JFern package.

Note: RAK system releases the user from the need of manual CLASSPATH setting and separate dependent JARs downloads, and we encourage the installation of the JFern module via the RAK module management system. The setup of the rakiura-jfern package is done in such a way that all dependencies must be placed in the same directory as the main executable rakiura-jfern jar file.

5 Petri net model

5.1 Java integration

The `Java` Fernlet engine emerged from the need of a Petri net tool which is fully integrated with the `Java` programming language. The idea was to provide `Java` programmers from different domains with a compact and small framework to create and reuse some of the patterns and models expressed in the form of Petri nets. It is not `Java` which got adopted to a general Petri net model, it is a Petri net model which got fully integrated with `Java` programming language.

All the construction, code segments, multiset expressions, guards, connections and simulations can be done programatically with pure `Java` syntax, and without a single click of the mouse on the graphical representation. Especially handy operations on multisets and tokens within guards and arc expressions are fully integrated with the `Java` programming language. Sometimes it makes the syntactical expressions longer than equivalent in some other simulation packages based on modified ML syntax, however most of the time it make things much more natural for `Java` programmer and much easier to express in a `Java` programming language.

Example: Imagine that you have to express a guard on an input arc which checks if there is a single (one and only one) token in the input place, and checks if the token has an appropriate value. With a Java-based net there is no real difficulty in expressing such a condition:

```
public boolean guard() {
  //check the arity of the multiset
  if(getMultiset().size() != 1) return false;
  //check the condition
  Number t = (Number) m.getAny();
  if(num.intValue() == 10)
      return true;
    else
      return false;
}
```

The method call `getMultiset()` is being delegated from the input arc guard to the current transition context (see `org.rakiura.CpnContext`), and returns the multiset of the input place.

The above Java code is quite long, therefore in the current release of JFern we have provided support for other (dynamic) languages that simplify the inscriptions substantially. The above condition, given that the token is bound to variable "x" in the input arc expression:

```
var("x");
```

can be simply expressed in Clojure as:

```
(and (number? x) (= x 10))
```

The user is free to declare which language should be used for a given Petri net. The choice of currently supported languages is: Java, BeanShell, Kawa and Clojure.

It is worth noting that there is already implemented support for graphical manipulation of the net, and graphical representation of some of the multiset expressions and guards. There is also support for visualisation of the network, simulation, debugging, and paging.

The user should keep in mind, that the JFern framework is primarily targeted towards runtime support for Petri nets, and integration of already designed Petri nets into Java-based software packages, and this is where that package is suited and can expose all its advantages.

5.2 Net Structure

Network structure and network model being used is based on the traditional model of hierarchical coloured Petri nets [1–3]. There are some minor terminology issues, and JFern package is considerably different than purely GUI-based tools like Design/CPN or Renew (see http://www.renew.de) and DesignCPN [1].

One of the major differences from the formal Coloured Petri nets is the abstraction of *object-based tokens*. With tokens values being generic Objects (in particular in JFern it means java.lang.Object), there is natural to introduce *colourless places* which can contain tokens with arbitrary `Java` Objects as values. The generic Place in JFern framework can contain any generic tokens, which means that it can contain any arbitrary `Java` Object. This high level of flexibility is similar to a programming languages which does not have typing system. Although the typing system is still present and can be explored by a keen Petri net programmer, the generic mechanism can be used without type control. This means for a Petri net designer that different logical token types can, if needed, be placed inside the same, generic place. Of course, this can be easily extended for example by extending `org.rakiura.cpn.Place` class to provide type checking, however, we feel that properly designed Petri net will never need such a runtime type checking.

As of version 2.0 closer integration with `Java` Collections framework was introduced, and in the most cases Multiset can be used with any other `Java` Collection type interchangeably. In version 3.0 this model remains unchanged. In version 4.0 close integration of Clojure collection has been added, so lists and arrays can be treated as multisets in the context of output arc expressions.

5.3 Simulated Time

JFern version 3.0 implements simple TimedSimulator which enables the user to perform simulations with time. Note, that this short description of the simulated time implementation is not intended as a tutorial of time simulations with Coloured Petri nets – we expect the user to be familiar with basic notions of simulated time from other Petri net related literature.

Simulated time has nothing to do with the external (wall clock) time and the units of simulated time do not represent any particular absolute time units. Users may interpret the simulated time steps as they wish, according to a particular domain of simulation.

Not all simulations will utilize the notion of time, thus to avoid adding overhead to such nets, to use timing one has to use a special simulator, namely `org.rakiura.cpn.TimedSimulator`.

The timestamps of tokens are normal numbers of type `long`, and the simulated clock is a single number of type `long`. TimedSimulator clock starts from `0`. The simulated clock does not automatically increment – it's value remains constant until the internal occurrence checking mechanism forces it to be incremented. The simulated clock does not increment by constant amount, say `1`. It increments by a particular interval calculated by the internal occurrence checking mechanisms. For example, suppose a system includes a token which has particular timestamp, and that nothing else can happen in the system until that timed token is activated (i.e. the simulated time matches the timed token timestamp). One would not want a model of the system to stand around ticking a simulated time one unit at a time – instead, one would want to jump the simulated clock immediately to the time when the activity is complete and proceed with executing the net. This is exactly what JFern implementation does. If during a step there is no enabled transitions in the whole net, the internal mechanism will try to find next available timed token, and will increment the simulated time. During the following step the occurrence set will be calculated and appropriate transition will be fired.

`TimedSimulator` includes some utility methods to set and check timestamps of tokens, and to check the current value of the simulated time. There are also two special utility methods for users who like to manually test and fire transitions, and manually move tokens around. These are `removeDisabledTokens()` and `addDisabledTokens()`. Please consult `JavaDoc` generated documentation for details.

6 Programming Model

The package `org.rakiura.cpn` contains the main JFern framework classes. This package also contains test suites for appropriate classes. Source files for tests classes are kept in a seperate directory structure, under `src/test`, whereas all the core sources are stored under `src/main` in the package directory structure. You can simply use all the classes by putting the `lib/rakiura-jfern.jar` into your `CLASSPATH`, as the compiled jar file contains both, test and non-test classes all together.

We will guide you now through the main classes and all the framework classes which build up the JFern kernel. Please, refer to the provided JavaDoc API documentations for more details[1].

6.1 Coding conventions

The source code follow the Sun Microsystems `Java` coding conventions [4]. Where the class or interface implementation can be used in graphical-based tools the Java Bean frameworks conventions for getter and setter methods is being used. However, for entities which are not likely to be used as Java Beans, simpler and names have been chosen.

6.2 Multiset

The very basic unit of information within the model is, not surprisingly, a token. Tokens in JFern are represented by Java `Objects`. The token can be initialized with any given arbitrary user-defined object type, and as such represents very generic *colour*. This, in fact, distorts the idea of the type of the token in traditional sense. The support for type checking is left up to the developer, and the framework operates on this very generic and abstract level of Object-based tokens.

The bag of tokens is traditionally called a *Multiset* and is represented by a java.util.Collection interface, which is implemented as `org.rakiura.cpn.Multiset` class. There is close dependency on JFern multisets and Java 2 Collections API, which provides most flexible API for the Java programmer.

Example: For example creation of a simple multiset and adding a single integer token to it can be accomplished via the following `Java` code:

[1] In case of a mismatch between this document and JavaDoc API the latter should be treated as the correct and most up to date reference.

Figure 1: JFern framework structure - core classes and interfaces

```
Multiset m = new Multiset(new Integer(10));
```

6.3 Place

Place is net element which is a container for tokens. It is represented as `org.rakiura.cpn.Place`. Place has set of incoming arcs, which are of type OutputArc, and set of outgoing arcs, which are of type InputArc. This arc naming convention is transition-centric, and relates to the role a given arc plays relative to a transition (not with relation to the place). Places can be accessed via name or id from within a given net object.

6.4 Transition

A transition is a stateless entity which can be enabled or disabled. Enabled transition can be fired by the Petri net simulator. It then will remove some of the tokens from the input places, according to the input arc expressions, and introduce new tokens to the output places according to the output arc expressions. As a side effect the transition will perform an `action`. The transition is represented as `org.rakiura.cpn.Transition` class. The transition action is represented as `org.rakiura.cpn.Transition.Action`.

It is important to note, that adding input and output arcs to the transition is not performed by the programmer manually, but this is done automatically by the framework when appropriate arcs are being created. Programmer should not directly use addInput/addOutput methods, which are accessed via the arcs internal mechanisms.

Transition Guard represents a boolean function which decides if the transition is enabled or not.

6.5 Input Arc

Arcs are divided explicitly into input and output arcs. Input arcs represent the mechanism to manipulate the connection between input places and a transition. Input arcs are represented as `org.rakiura.cpn.InputArc` objects.

The input arc expression is used to select a set of tokens from the input place. The selection is being achieved by the utility methods from the transition context, which is represented as CpnContext object. The user can use named variables and/or anonymous variables for selecting the tokens.

The input arc guard evaluates to true or false, notifying the simulator whether this particular binding for this single arc is valid or not. The selection, guards and unification mechanism is being discussed further in this manual.

Note: input arc guards can be written in any of the supported languages, however, the input arc declarations must use Java API variable declarations in a form:

```
var("variableName");
```

6.6 Output Arc

Second type of arcs are output arcs. Output arcs are represented as `org.rakiura.cpn.OutputArc`. This arcs are used to generate output tokens which are being placed into output places.

The OutputArc Expression simply generates a multiset, which can be empty, or which can contain a bag of tokens which will be placed inside the output place of this arc. The user is free to use existing tokens from the input places, or generate completely new tokens.

7 Context and unification mechanism

7.1 Context

As of JFern 2.0 there is implemented a notion of context and variable handling, which takes a burden of work from a programmer directly into Petri net kernel. There is a simple notion of context, which is a variable space for a single transition, spanning all input and output arcs and the transition itself. All arc expressions, guards and transition action can use and manipulate variables within a scope of a single context. In version 3.0 all the core kernel mechanisms are the same as in version 2.0.

7.2 Input arc variables

In the context of programming inscriptions, the user will use named variables declared in the input arc expressions.

However, in the context of Java inscriptions, where the Java code has access to the entire multiset of a given input place, not naming the variables may be appropriate. There are therefore two types of variables which are being used in input arc expression to select tokens: named variables, and anonymous variables.

Named variables are like any ordinary variable in Java programming language. They are identified by an identifier, and, they are bound to a single Object, in our case, a single token.

Anonymous variables are like named variables but they do not have the name, thus, the programmer cannot refer to them or identify them by name. These are useful in situations where you do not need a seperate individual references to each of the tokens you have selected from the input place (note, this can only be useful for Java inscriptions). The values which are bound to anonymous variables can be accessed by the call to the `getMultiset()` method in a given context, and it will return all bound variables values.

Anonymous variables is a tool for the programmer to handle transitions with many inputs and many tokens in each input place. When there is dozens of input places and hundreds of tokens in each input place, partial matching of all enabling bindings leads to exponential growth of computational complexity. In such situations the programmer is free to use anonymous variables instead, which leads to more linear growth of complexity. Lets take the example of selecting two tokens from the given input place. To use named variables the programmer writes in the input arc expression for example:

```
var("x");
var("y");
```

This semantically simply mean: select two tokens and bind them to variables `x` and `y` respectively. In case of anonymous variables the code looks like this:

```
var(2);
```

This means: select two tokens, and I do not care what variable names they have.

7.3 Guards and partial matching

In JFern there are two types of guards: input arc guards, and transition guards. Guards are simply a boolean methods which return `true` if a given arc or transition is enabled. For the transition to be enabled all its input arcs must be enabled and the actual transition guard must evaluate to true.

To speed-up a process of evaluating whether a given transition is enabled or not it is important to place the appropriate conditions in the most narrow scope. For example, if we are to select a single token with particular properties from an input place, it is much better to place the condition on the token properties in the input arc guard instead of transition guard. By putting the conditions in the input arc guards we reduce the search space for valid bindings, and by doing that we can speed-up the network simulations dramatically, sometimes by factor of 1000 or more.

In the transition guard we should place only these conditions which span multiple input places, like comparison of tokens from separate input places.

7.4 Tokens generation and side-effects

The tokens generation takes place in the output arc expressions, which simply returns the multiset of tokens. These tokens got inserted into the appropriate output place by the end of the transition firing process.

All side-effects should be done in the action code, which is guaranteed to fire only once during a successful transition firing process. The evaluation of output tokens can be performed more than once in a single step of the simulator, therefore any side-effects placed in the output arc expression may be executed more than once in a single step. This is very `bad` and the user should never put any side-effects into output arc expressions.

7.5 Fusion places

JFern supports fusion places. Two or more places can be displayed on the net diagram, and they will "share" same tokens. It means, a token entering one place will appear automatically in all fused places, and token removed from one place, will disappear automatically from all the other fused places.

With complex nets there is a lot of arcs, and very often one can't prevent arcs from crossing each other. Crossing arcs make the layout of the net chaotic and difficult to comprehend. To solve this problem, you can fuse two or more places. Fused places still appear as different places, but they share the same tokens, and it may make the visualisation easier to comprehend.

8 XML File format

8.1 Overview

JFern does have a support for an XML-based network structure description. The XML fileformat is derived and very close to the one proposed by Olaf Kummer in his Renew system. There are some differences however.

Our file format for the net structure and marking completely lacks the graphics, layout and GUI descriptions. We believe that the net structure and the graphical rendering of the net structure are completely seperate aspects, and should be kept in seperate files. In particular it will simplified exchange of the net structure between different tools. In particular, JFern does not have a graphical representation, thus the layout is of no concern.

Our net declaration section is beeing split into two seperate blocks, one containing all the import statements (import block), and one containing the preamble to the generated net class (declaration block). User can use the declaration block to define her own global fields, variables, and perform some global initialization of the class representing the given Petri net.

The parser and network generator are implemented by the `org.rakiura.cpn.NetGenerator` class. The parser will ignore all additional attributes, and tags, and will process the tags described in the following section. This should make it compatible with richer file formats, like Renew for example.

8.2 XML tags

```
<!ELEMENT net (place*, transition*, arc*, annotation*)>
<!ATTLIST net id ID #REQUIRED type CDATA #IMPLIED>

<!ELEMENT place (annotation*)>
<!ATTLIST place id ID #REQUIRED type CDATA #IMPLIED>

<!ELEMENT transition (annotation*)>
<!ATTLIST transition id ID #REQUIRED type CDATA #IMPLIED>

<!ELEMENT arc (annotation*)>
<!ATTLIST arc
   id ID #REQUIRED
   source IDREF #REQUIRED
   target IDREF #REQUIRED
   type CDATA #IMPLIED >

<!ELEMENT annotation (text)>
<!ATTLIST annotation id ID #REQUIRED type CDATA #IMPLIED>
<!ELEMENT text (#PCDATA)>
```

The net tag contains nested network structure description together with additional annotations for the global network settings.

Annotation for the net can be:

- **name** – the name of the net. Must be a valid Java identifier.
- **import** – sequence of ordinary import statements. Can be preceded by the package declaration.
- **declaration** – sequence of ordinary Java class top level declarations (fields, static code blocks, additional methods, etc);

Annotation for the transition can be:

- **guard** – the code of the transition guard. In the scope of this block there is given **Marking marking** variable where the current net marking is being passed. This block must ends with the return statement with the boolean result.
- **action** – sequence of ordinary Java statements defining the transition action. In the scope of this block there is given **Multisets multisets** variable where the current set of input multisets is being passed.

Annotation for the input arcs can be:

- **guard** – the code of the input arc guard. In the scope of this block there is given **Multiset multiset** variable where the current input place marking is being passed. This block must ends with the return statement with the boolean result.
- **expression** – sequence of ordinary Java statements defining the arc expression. In the scope of this block there is given **Multiset multiset** variable where the current input place marking is being passed. This block must ends with the return statement with the Multiset result.

9 XML Parsing and Net generation

Once the net is designed and a XML file prepared (for example by Renew), the XML can be turned into Java source, which is then compiled into a native Java class file. Such class, placed inside a classpath can be used to instantiate copies of the given net type.

9.1 Static mode

In this mode user generates a Java file, with the Java source of the net. The user than compiles it together with the rest of his classes and uses it as a normal Java class.

Parsing and generating the net can be done from within the API (see methods `generateNetSource()` from `org.rakiura.cpn.NetGenerator`, or it can be done from the command line as:

```
java org.rakiura.cpn.NetGenerator inputfile.xml
```

9.2 Dynamic mode

In this mode user generates an instance of the net dynamically, directly from the XML description, without any intermediate stages and without generating any files into the filesystem. The Java source and compilation process are done internally by JFern, and the user simply receives the instance of the generated net class. To achieve that the user uses the methods `generateNet()` from `org.rakiura.cpn.NetGenerator`.

10 Event/Listener model

As of version 2.0 there is a small event-listener model built-in. The design follows closely typical Java based event models. All necessary tools, events and listeners are defined in `org.rakiura.cpn.event` package. Version 3.0 is identical in that respect to its predecessor.

11 Visualization

As of version 3.0 there is implemented net editor, called JFern Toolbox, which contains the editor, vizualiser, debbuger, and on-line visual simulator. The implementation is based on JRio (http://sf.net/projects/cirrus) package, which in turn is derived from JHotDraw package. To run the graphical editor, JFern Toolbox, one needs to have rakiura-jrio.jar installed. JRio is distributed together with JFern package.

11.1 NetViewer

To display a standalone view with the given net, simply write:

```
import org.rakiura.cpn.gui.NetViewer;

//  ....

Net net = // obtain the reference to the net
NetViewer viewer = new NetViewer(net);
viewer.getFrame().show();
```

One can embedd the NetViewer as any other AWT or Swing compontent inside a custom GUI. Or one can use it as a standalone JFrame as in the example above.

11.2 JFern Toolbox

The JFern Toolbox is a tool for creating, simulating and editing CPNs. We will describe the different objects of the main window, and we will describe some controls how to use the toolbox.

The JFern toolbox window consists of the following parts:

- The net tree. This is a tree representation of all information about the net.
- The attribute-value list. This displays the different attributes and values of the last selected object.
- The value panel. This displays the value of the last selected attribute from the attribute value list.
- The net panel. This displays the layout of the net you are editing. The net is represented as a box with rounded edges. Places are displayed as circles. Transitions are displayed as rectangles. Arcs are displayed as arrows. Subnets are displayed as rectangles with double lines. Each object has certain properties that can be shown or hidden in the net panel by double clicking on a figure like it s name, the A for the action code, the G for the guard code, the E for the expression code, the I for the imports code, the for the implements code and the # for the declaration code.
- The token panel. This displays the tokens that are in the last selected object. You can use this for simulation.
- An empty panel. This shows your favourite TV show, or plays randomly selected episode of “The Simpsons”.

11.3 Key bindings

When you edit your net, the following keyboard controls and menu commands might be of help.

- delete a figure (only in editor, not in viewer): <backspace> or <del>
- Select all figures in the drawing: <ctrl> + a
- Add/remove figure to selection: ¡shift¿ + mouseclick on figure
- Move the selection: arrow keys
- Move selection to front (= change the order of the selected elements so the elements get in front other objects): <ctrl> + [

- Move to back (=change the order of the selected elements so the elements get behind other objects) <ctrl> +]
- Group selected objects Menu Edit> group
- Ungroup selected objects Menu Edit > ungroup
- update the net tree (the net tree is the tree in the top left panel) click on the net-figure
- Change the display of the figure (like: show annotations/name or change text of a CPN element) doubleclick on a figure
- Move all annotations back to their original offset <alt> + doubleclick on a figure

Warning: the JFern Toolbox does not have an undo function. So be very careful when you edit or delete a value of a place attribute.

11.4 Attributes

Net attributes:

- Figure The type of the object; this always is CPN net . This attribute cannot be edited.
- ID The unique ID of the net. This attribute cannot be edited.
- Name The name of the net. The user can use this for its own use. We advice you to create unique and meaningful names
- Type The type of the net. The user can use this for its own purposes. The default value is: hlnet (high level net)
- Imports Sequence of ordinary import statements that are necessary for the execution of the net. It can be preceded by the package declaration.
- Implements The java package that the CPN implements
- Declaration Sequence of ordinary Java class top level declarations that are necessary for the execution of the net like fields, static code blocks, additional methods

Verification check When the Imports has been set, a special symbol in a form of a vertical bar: — is shown in the net object, in the net layout panel. When the Implements has been set, a is shown in the net object, in the net layout panel. When the Declaration has been set, a # is shown in the net object, in the net layout panel. To verify if the values you have entered are correct, you can (re)create a net-object. The JFern compiler then tries to compile a net with the specification of the net you are editing. It tries to create the adequate net object that has the proper code for all expressions. Note that in the toolbox you only change the text-representation of the annotations, not the actual computer code. This makes sure that the code is updated as well. You can do this as follows: Go to the menu net ¿ resync nethandle . If the resync was successful, an inform message will be displayed. Otherwise a window appears with the compilation error(s).

Places:

- Figure The type of the object; this always is CPN place . This attribute cannot be edited.
- ID The unique ID of the place. This attribute cannot be edited.
- Name The name of the place. The user can use this for its own use. We advice you to create unique and meaningful names

Transitions:

- Figure The type of the object; this always is CPN transition . This attribute cannot be edited.
- ID The unique ID of the transition. This attribute cannot be edited.
- Name The name of the transition. The user can use this for its own use. We advice you to create unique and meaningful names
- Type The type of the transition. The following values have a special display, according to Van der Aalst s example: **task**: a task is displayed with an extra line at the bottom; **routing**: depending on the value of the specification field.
- Specification The specification of the type of transition. The following specifications have a special display when the type has been set to routing , according to Van der Aalst s example : **AND-split** **AND-join**, **OR-split**, **AND/OR-split**. Note that these values don't have in influence on the actual working of the transition; the real kind of routing depends on the net layout.
- Guard A boolean function that needs to be true, together with the guards of the input arcs, to fire the transition
- Action The action code that is executed when the transition is fired

When the Guard has been set, a G is shown in the transition object, in the net layout panel. When the Action has been set, a A is shown in the transition object, in the net layout panel.

11.5 Subnets

JFern supports hierarchical Petri nets, and the Toolbox editor can help you to create and manipulate subnets. You can set the attributes of the subnet, and you can set the layout of the subnet. Subnets are useful to manage complex nets so the overall design and structure are comprehensible. When a part of the whole main net is placed in a subnet, the main net looks easier to comprehend.

You can set the layout of the subnet. To do this, double-click on the subnet. Now a window appears that shows the subnet. For every place in the main net that is connected to the subnet, a ghost place appears in the subnet layout. These ghost places cannot be edited. To edit the corresponding place, go to the main net and edit the corresponding place there. If you haven't specified the layout of the subnet, before you connect places to the subnet in the main net, then one transition is generated automatically and the ghost places are connected to that transition. The accompanying arcs cannot be moved. So we advise you to first design the actual subnet before connecting that subnet to the rest of the main net. After you designed the subnet, you can go back to the main net and connect the subnet object there. When you draw an arc between a place and the subnet, you will be prompted to select the appropriate transition. The selected transition will be connected to the place. In the subnet a ghost place is created that represents the real place. This ghost place act as the same place from the main net.

12 Hints and suggestions

1. The methods var(int, class) and var(string, class) are not implemented yet. Don't use these methods.
2. Multisets can have multiple references to the same token. So if you add the same token twice, you'll have two tokens in your place.

3. The expression in an arc is not only executed if the transition fires, but it can also be executed if the simulator checks if a transition is enabled. Therefore you should not enter any side-effect generating code inside the expressions. Only use code that selects tokens.

4. The toolbox does not have an undo function. So be very careful when you delete an old value of an arc attribute, or whatever.

5. You can't edit ghost arcs (arcs going to/from subnets). If you want to set properties, you need to go to the layout of the subnet and edit the appropriate arc.

13 Examples

13.1 Maximum value

This example network is graphically presented on Figure 2. This network in each step of the simulation is taking two tokens from the input place, and returning back one, which is the maximum of the two tokens. The tokens color is the abstract Comparable which means a generic ordering relation exist between tokens. For example, objects of type java.lang.Integer can be used.

Figure 2: Maximum Value - an example net

For the net to work correctly in JFern the input arc expression should read:

```
// select two tokens
var(2);
```

the input arc guard should read:

```
// check if we have two or more
// tokens in input place
return getMultiset().size() > 1;
```

and the output arc expression should read:

```
final Multiset result = new Multiset();
final List list = new ArrayList(getMultiset());
final Comparable t1 = (Comparable) list.get(0);
final Comparable t2 = (Comparable) list.get(1);
if (t1.compareTo(t2) > 0) {
        result.add(t1);
} else {
        result.add(t2);
}
```

In this example we have used anonymous variables for efficiency purposes. The same version with the use of named variables would look like in the following code snapshot. For the input arc expression we use two variables `x` and `y`:

```
// select two tokens
var("x");
var("y");
```

and the output arc expression should read:

```
final Multiset result = new Multiset();
final Comparable t1 = (Comparable) get("x");
final Comparable t2 = (Comparable) get("y");
if (t1.compareTo(t2) > 0) {
        result.add(t1);
} else {
        result.add(t2);
}
```

As we can see, the use of anonymous variables does not change much the way tokens are being used, and there is little overhead with them. We encourage the use of anonymous variables in appropriate situations, especially when there is many input places with many tokens in these places.

References

[1] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts. Springer-Verlag, Berlin, 1992.

[2] J. l. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, 1981.

[3] W. Reisig. *Petri Nets. An Introduction*, volume 4 of *EATCS Monographs on theoretical Computer Science*. Springer-Verlag, 1985.

[4] Sun Microsystems. Code Conventions for the Java Programming Language. http://java.sun.com/docs/codeconv/index.html.