In Proc. International Workshop on Concurrency, Specification, and Programming (CSP 2011), Biaystok University of Technology, ISBN: 978-83-62582-06-8, pages 531–554, September 2011. http://csp2011.mimuw.edu.pl/proceedings/index.html

A Plugin System for Charlie

Jan-Thierry Wegener[1], Martin Schwarick[2], and Monika Heiner[2]

[1] Institute of Biology, Otto-von-Guericke-University, 39106 Magdeburg, Germany
jan-thierry.wegener@ovgu.de
[2] Department of Computer Science, Brandenburg University of Technology
Postbox 10 13 44, 03013 Cottbus, Germany
{ms, monika.heiner}@informatik.tu-cottbus.de

Abstract. Charlie is a tool for analyzing place/transition Petri nets. Due to its features and its transparency of the supported rule system Charlie is not only used in research, but also in teaching. The functionality of Charlie can be easily varied and extended by deploying a plugin mechanism. In this paper we present different features which can be expanded and show how one can implement a new plugin for Charlie.
Charlie is written in pure Java and thus runs on most operating systems, including, but not limited to Linux, MacOS, and Windows. It is available free of charge at http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Charlie.

1 Introduction

Charlie is an analyzing tool for place/transition Petri nets capable of performing various static and dynamic analyses of the standard body of Petri net theory. It is able to compute place and transition invariants, siphons and traps and to construct and evaluate the reachability graph, just to mention a view basic analyzers. Charlie has been greatly inspired by the tool INA (Integrated Net Analyzer) [18], which was developed by Peter H. Starke until 2003. Still today one can see the influence. For example, the abbreviations for the names of the properties in Charlie are almost the same as in INA, compare Figure 1. An overview and explanation of the abbreviations can be found in [19] or [12], and more technical details in [17].

Charlie comes with an intuitive and easy to use Graphical User Interface (GUI), see Figure 1 for a screenshot and its explanation.

Charlie reads Petri nets written in the Abstract Petri Net Notation (APNN, see e.g. [7]) and files that haven been created by INA or the tool Snoopy [13], [15]. Snoopy is a program for drawing and simulating Petri nets; it is available at [2]. Currently, Charlie ignores additional information which may be stored in Snoopy files, e.g., the firing rate of transitions of stochastic Petri nets (see e.g [14], [6]). Additionally, there is an interface for writing new readers to load Petri nets from files stored in a different file format.

The development of Charlie began in the year 2006 by Martin Schwarick [16]. Since then, Charlie has been improved by various students; e.g., Ansgar Fischer extended the capabilities by analyzing time-dependent Petri nets in his diploma thesis [9].

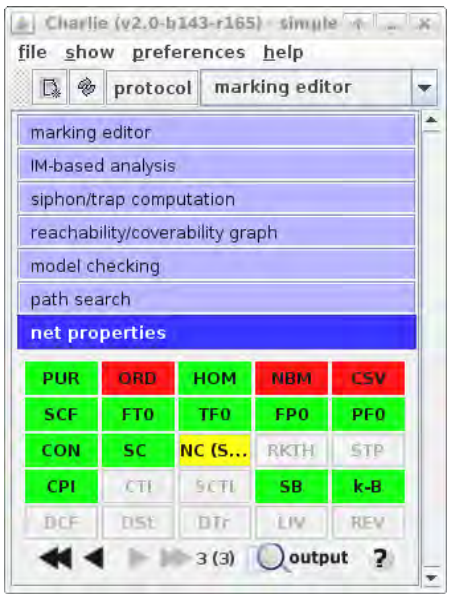

Fig. 1. A screenshot of Charlie's GUI showing the standard features without any extension. Starting from the top one sees the menu bar, followed by some buttons for a quick access to often used functions and the dialog boxes for the different analyzers. At the bottom a special dialog appears, the "net properties". This dialog shows all properties Charlie knows of. If the loaded Petri net fulfills a property then it is marked in green, if the Petri net does not fulfill a property then it is shown in red. Unknown properties, i.e., properties that have not been analyzed yet are shown in gray, while a yellow color indicates that more information than true or false are provided. Finally, there are control buttons to switch through the different results and to show the "output" and "help" windows.

In the year 2009 Andreas Franzke introduced a new GUI and redesigned the basic architecture of Charlie's code [11]. This redesign made one of our recent developments possible: a plugin system. Before the plugin system was implemented, users had only little possibilities of configuring Charlie according to their needs. Now a user can decide which analyzers are required. This improves the flexibility and configurability of the program and thus improves the general usability. For example, the analysis of time-dependent Petri nets has been reorganised as a plugin. Therefore, only users who are interested in analyzing time-dependent Petri nets have a further dialog box containing the settings for analysing them.

Plugins for Charlie are easy to use, for the user and for the software developer as well. Due to the standardization and abstraction of the basic classes, which are in need of writing new analyzers, it became very easy to implement new analyzers in Charlie. This also helps to decrease the time spent on implementing new analyzers and thus increases the overall productivity. Furthermore, users who want Charlie to be extended are now able to write their own extensions. In this paper we give the

required background knowledge and explain in detail the necessary steps to be taken to add new plugins to Charlie.

The paper is organized as follows. The next section gives a brief overview of some other tools for analyzing Petri nets. In Section 3 we describe how to implement a new analyzer for Charlie. This step is usually the most time-consuming part, but also the most important step of writing a plugin. In Section 4 we show how to implement a user dialog that can interact with the newly created analyzer. After that, i.e., in the fifth section, we give a brief description of Charlie's rule system. The section after, i.e., Section 6, gives an overview on writing new readers. A reader or parser is a class responsible for reading external file formats. Section 7 describes how a plugin file is created and how to install the new plugin within Charlie. In the penultimate section we give a brief overview of the class hierarchy of the plugin system and some additional information about its implementation. Finally we summarize our work and give an outlook of our future work.

2 Related Work

There are several other tools for performing structural analysis of Petri nets. Some also come with their own plugin system. There are two major differences in the design of plugin systems. Some programs provide a ready and fully usable framework for the developer, while others let you call your own program and thus only work as a front-end for other analyzers. Both approaches have their own advantages and disadvantages. An advantage of the first approach is that the developer does not need to implement basic classes, e.g., a representation of a Petri net, and thus saves a lot of time. On the other hand, the developer is, more or less, bound to the specific framework. Furthermore it also takes some time to get used to a framework.

Charlie itself provides a framework for writing analyzers. Another program following this approach is the "Platform Independent Petri net Editor" (PIPE) [1]. It comes with an editor for drawing Petri nets. The functionality of PIPE is close to the functionality of Charlie. However PIPE does not have a rule system (see Section 5).

The program "TIme petri Net Analyzer" (TINA) [4] allows users to include their own programs. These programs can then be called from TINA. Petri nets can be drawn with a built-in editor. Unfortunately, the functionality of TINA in the field of structural analyses is not comparable to Charlie. It also does not have a rule system like Charlie does.

3 Writing an Analyzer

In this section we explain how to write an analyzer for Charlie. As running example we realize an analyzer that makes a very simple and basic test whether a Petri net is structurally bounded (see e.g. [12]). In our simplified example we test a Petri net for structurally boundedness by analyzing its incidence matrix. If there is no positive entry in the incidence matrix (meaning, none of the transitions increases the token number), then the Petri net is structurally bounded. Although this analysis is extremely limited in its application, we have chosen this simplified scenario since it is

easy to understand, but complex enough to demonstrate the main steps of writing new analyzers.

The first step is to write a new class that extends the class `PluginAnalyzer` which is located in the package `charlie.plugin.analyzer`. This class is an abstract class and one has to overwrite seven methods.

The first method to overwrite is `getName()` which returns a string with the name of the analyzer. The returned string should be human readable since it is not used as an identifier, but it is shown in the Analyzer-Thread-Manager window during the analysis.

```
public String getName() {
  return "simple structurally bounded analyzer";
}
```

The next three methods we describe are `analyze()`, `evaluate()`, and `cleanup()`. The main work is done in the method `analyze()`, in `evaluate()` the results are summed up and given to the result manager. Any additional cleanup work, like removing temporary files, can be done in `cleanup()`.

In our running example we want to analyze the incidence matrix of the loaded Petri net. The loaded Petri net can be received by calling `getObjectToAnalyze()` on the option set, which is done in the first line in the method `analyze`. The incidence matrix itself can be received from the class `PlaceTransitionNetUtils`. This is done in the third line. Finally the actual analysis is done from line four to line seven.

```
public void analyze() {
  PlaceTransitionNet pn = (PlaceTransitionNet)getOptionSet().
      getObjectToAnalyze();
  PlaceTransitionNetUtils utils = new PlaceTransitionNetUtils(pn);
  int[][] im = utils.getIncidenceMatrix();
  for (int i = 0; i < im.length; i++) {
    for (int j = 0; j < im[i].length; j++) {
      if (im[i][j] > 0) {
        bounded = false; return;
      }
    }
  }
}
```

The class `PlaceTransitionNet` is responsible for representing a Petri net in Charlie. For creating an analyzer that is capable of analyzing, e.g., stochastic Petri nets, one can simply extend the class `PlaceTransitionNet` and add the methods and features needed. Another possibility is writing a new representation of the data. This is possible since `getObjectToAnalyze()` returns an object of the class `java.lang.Object`.

Having finished the analysis, we set our result in the method `evaluate()`. This is done by calling the method `addResult(int, Object)`.

```
protected void evaluate() {
  addResult(Results.SB, Boolean.valueOf(bounded));
}
```

In our simple running example we do not need to do any cleanup, since the memory usage is automatically freed by Java's garbage collector. Therefore the method `cleanup()` remains empty.

Next we want to provide the user with some statistics about the analysis. The initialization of this information is done in `initializeInfoStrings()`. In our example, we only inform the user about the elapsed time.

```
public void initializeInfoStrings() {
  infoStrings = new String[2];
  infoStrings[0] = "time";
  infoStrings[1] = getFormatedDuration();
}
```

Before the analysis can start, the analyzer manager creates a new analyzer object which is set up by an `OptionSet`. In our simple example we do not need any initialization of the analyzer and thus we simply return a new object. If the analyzer is of a more complex nature the initialization of the object needs to be done here.

```
public Analyzer getNewInstance(OptionSet options) {
  return new SimpleStructurallyBoundedAnalyzer();
}
```

Finally, we describe how the analyzer manager can be informed about our analyzer. This is done in the method `registerAnalyzer()` which should return true if the analyzer could be registered, and false if the registration of the analyzer failed. In order to do so we need to write an additional class, a result set for our analyzer. A result set is a class that can extend any arbitrary class and should be able to store any additional results. In our simple case we write an empty class that extends the object class. We name the class `SimpleStructurallyBoundedResultSet`. Please note, the analyzer manager decides from the object that shall be analyzed and from the result set that is given, which analyzer to be used. Therefore one should write a result set for each analyzer.

```
public boolean registerAnalyzer() {
  return AnalyzerManager.register(
    new SimpleStructurallyBoundedAnalyzer(),
    new PlaceTransitionNet(),
    new SimpleStructurallyBoundedResultSet());
}
```

4 Writing a Computational Dialog

In the preceding section we have shown how a new analyzer for Charlie can be written. Now we concentrate on the question how the user can invoke our new analyzer. This is done by writing a dialog panel to that GUI elements are added and with that the user can interact. The dialog panel is called computational dialog.

A new plugin computational dialog must extend the abstract class `PluginComputationalDialog` which can be found in the package `charlie.plugin.gui`. One

has to overwrite two methods of the abstract class. Furthermore one must provide a specific constructor.

So the basic template of our class is the following.

```
import charlie.plugin.gui.PluginComputationalDialog;

public class SimpleStructurallyBoundedComputationalDialog extends
    PluginComputationalDialog {

  public SimpleStructurallyBoundedComputationalDialog(IDirector
      director) {
    super(director);
  }

  public String getDescription() {
    return null;
  }

  public void initialize() {
  }
}
```

The method `getDescription()` returns a string containing the description or name of the dialog. The description is shown to the user and thus it should be human readable and meaningful. In our example, the method returns "simple structural analysis".

The method `initialize()` creates the GUI elements and adds these elements to the dialog panel. We assume that the reader has basic GUI programming knowledge. In the next listing it is also shown how to start our analyzer and its analysis.

First we prepare the layout using the TableLayout. The class `TableLayout` can be downloaded at [3]. It is used by Charlie and thus also included in Charlie.

```
public void initialize() {
  double size[][] = {{5,TableLayout.FILL,5},
                     {5,TableLayout.PREFERRED,5}};
  setLayout(new TableLayout(size));
  JButton computeButton = new JButton("compute");
  final Initiator thisInitiator = this;
  computeButton.addActionListener(new ActionListener() {
    AnalyzerManager.compute(getPN(),
                    new SimpleStructurallyBoundedResultSet(),
                    thisInitiator);
  } );
  add(computeButton, "1,1");
}
```

In a more complex analyzer it is also possible to specifiy some settings. This can be done by creating a class that extends the class `PluginOptionSet`. Then one can use the method `AnalyzerManager.compute(Object, Object, OptionSet, Initiator)` or the method `AnalyzerManager.compute(OptionSet)` to provide the analyzer with the settings.

5 Expanding the Set of Rules

In Charlie not every property has to be computed by an analyzer. Several well-known theorems are also implemented in Charlie, e.g. the fact that a Petri net that is covered by P-invariants (see e.g. [12]) is automatically structurally bounded. By applying theorems to already computed results one can save a great amount of computational time. The implementation of theorems is done with Charlie's rule system.

A rule consists of a set of results, the pre-conditions, and another set of results, the post-conditions. After an analyzer has finished its analysis, the rule system checks if a known rule can be applied to the results, i.e., if there exists a rule so that all pre-conditions of the rule are fulfilled. If a rule can be applied to the known results, then all results in the post-condition are set. An example for an implemented rule is the well-known theorem from above: if a Petri net is covered by P-invariants, then it is structurally bounded. In Charlie this means that if the loaded Petri net is covered by P-invariants and an analyzer sets the property "covered by P-invariants" to "true", then the mentioned rule is applied to the results and the property "structurally bounded" is set to "true" as well. This example demonstrates a big advantage of Charlie's rule system: in many cases a lot of computational work can be saved. Especially in big nets the rule system improves the speed of the net analysis drastically.

As already mentioned, Charlie is a great tool for teaching of properties of Petri nets and their applications. By default every rule that is applied to the analyzed results is shown to the user. This can be seen in Figure 2. The behavior of asking for the appliance of a rule can be set in Charlie, i.e., a rule can also be silently accepted by Charlie, making it more convinient for using the tool in the field of research.

Fig. 2. Charlie asks the user for applying a rule. As one can see, Charlie handles each property separately. Thus, there is a rule for "k-bounded", and there is a rule for "structurally bounded" as well, although the property "structurally bounded" implies the property 'k-bounded".

The set of supported rules can be expanded by a plugin. Furthermore, the set of results can also be expanded and thus it is possible to make Charlie ready to be usable for a new class of Petri nets, e.g., time-dependent Petri nets (see, e.g. [5], [20]), continuous Petri nets (see, e.g. [8]), etc.

In order to expand the set of rules one has to create a class that extends the class `PluginRuleExtender`, which can be found in the package `charlie.plugin.analyzer`. In this class there are two methods that can be overwritten. The first method is `getAdditionalRules()` which returns a list of rules that should be added to the set

of rules. The second method is `getAddionionalResults()` which returns a list of new results. These results can be used in any rule. By default the results are also shown in the property dialog.

Further information about Charlie's rule system can be found in [11].

6 Writing a Reader

By default Charlie reads a few file formats – all files created by Snoopy, APNN files, and INA files. In all these cases, any additional information are ignored, e.g., firing rates of transitions in stochastic Petri nets or the colors of tokens, places, and transitions in colored Petri nets.

It is rather straightforward to write another reader by extending the abstract class `charlie.plugin.io.PluginPlaceTransitionNetReader`. There are five methods that must be overwritten: `getLoadableExtensions()`, `getDescription()`, `read()`, `countPlaces()` and `countTransitions()`. The method `getLoadableExtensions()` returns a list of strings. Each returned string can contain several different file extensions, separated by commas, that the reader is able to read. The method `getDescription()` returns a list of strings describing the extensions the reader can read. For example, if the reader is able to read files with the extensions ".foo", ".foob" and ".foobar", where ".foo" and ".foob" are two different file extensions for the same format (like ".htm" and ".html"), then the methods `getLoadableExtensions()` and `getDescription()` would look like:

```
public List<String> getLoadableExtensions() {
  List<String> extensionList = new ArrayList<String>();
  extensionList.add(".foo, .foob");
  extensionList.add(".foobar");
  return extensionList;
}

public List<String> getDescription() {
  List<String> descriptionList = new ArrayList<String>();
  descriptionList.add("Read foo files.");
  descriptionList.add("Read foobar files.");
  return descriptionList;
}
```

Please note that the size of the list returned by `getLoadableExtensions()` must be equal to the size of the list returned by `getDescription()`.

The method `read()` actually reads the file and fills the contents of a Petri net. A Petri net object is initialized before `read()` is called. This object is of the type `charlie.pn.PlaceTransitionNet` and can be accessed by the method `getPlaceTransitionNet()`. There are three basic methods in the class `PlaceTransitionNet` with that one can add places, transitions and edges to the Petri net, namely `addPlace(Place)`, `addTransition(Transition)` and `addEdge(Node, Node, int, int)`. Further information of these methods and the classes `PlaceTransitionNet`, `Place` and `Transition` can be found in Charlie's Javadoc documents.

The two methods `countPlaces()` and `countTransitions()` return the number of places and transitions, respectively. Both methods must be implemented in a way that the number of places and transitions can be accessed without having called the method `read` before. In other words, before `read()` is called, the methods `countPlaces()` and `countTransitions()` are called and must return the correct values for the number of places and transitions in the Petri net. This can be done by parsing the file three times in total; once in `countPlaces()`, once in `countTransitions()`, and finally in `read()`. This behavior is planned to be changed in the future so that the file needs to be read only once.

7 Deploying the Plugin

Having written an analyzer and a computational dialog, the major work is usually done. Now we explain how the plugin file is created and how it can be used in Charlie.

The plugin file itself is a standard zip file that contains the analyzer class, the computational dialog class and any other additional class that is needed. Furthermore one must create a Java property file with the name `charlie.meta`. In this file one writes the names of the classes and their usage. For example, in our case the file `charlie.meta` could look like the following:

```
name=Simple Structurally Bounded Plugin

analyzer=de.tu_cottbus.analyzer.SimpleStructurallyBoundedAnalyzer
dialog=de.tu_cottbus.analyzer.
    SimpleStructurallyBoundedComputationalDialog
```

There are some more properties that can be set, e.g., to expand the rule system, to provide a reader or setting a version for the plugin. In order to expand the rule system one has to fill the property `rule`, for providing a plugin reader one has to fill the property `reader` and in order to set a version one has to fill the property `version`. While the properties `rule` and `reader` expect the name of the classes that extend the class `PluginRuleExtender` and the class `PluginPlaceTransitionNetReader`, respectively, the value given to a `version` is arbitrary.

Afterwards, the property file is simply added to the root of the zip file. The zip file itself is put into the plugin folder in the Charlie directory, e.g., if Charlie is installed in `/usr/local/lib/charlie`, then the zip file is copied to `/usr/local/lib/charlie/plugin`.

If further external libraries are needed, one can put the jar file into the plugin zip file. The class loader looks for additional jar's in the plugin file in a special folder named `lib`. For example, if the jar "foo.jar" is needed in the plugin, then the file must be put to `lib/foo.jar` inside the zip file. Please note that currently there is no mechanism to ensure that a specific library is loaded first, i.e., if there are two plugins that need the same library, but with different versions it is not ensured that their respective library is indeed loaded. This problem will be taken care of in the future.

To make life easier for developers, there is a mechanism in Charlie that builds plugins at the start-up phase. This is useful for testing plugins during the development without creating the plugin file manually. In order to let Charlie create

the plugin file one has to add a line to the file `plugin/plugin.devel.properties` containing the path to the class files of the plugin. For example, if the class files of the plugin are stored in `/home/wegener/workspace/charliePlugin/bin` (as it is done by Eclipse [10] by default), then this path is added to the file `plugin/plugin.devel.properties`. In this case please do not forget to copy the file `charlie.meta` to the directory `/home/wegener/workspace/charliePlugin/bin`, otherwise the plugin can not be loaded.

8 Overview of the Class Hierarchy

In this section we give a brief overview of the class hierarchy of the plugin system. All classes necessary for writing a basic plugin are located in sub-packages of `charlie.plugin`, namely `charlie.plugin.analyzer`, `charlie.plugin.gui` and `charlie.plugin.io`.

The package `charlie.plugin.analyzer` contains classes to write new analyzers and their set of options. Here are also the classes stored which are necessary to extend the rule system.

For writing an extension for the GUI one has to extend the classes stored in the package `charlie.plugin.gui`.

The class `PluginPlaceTransitionNetReader`, which is required for providing a reader, can be found in the package `charlie.plugin.io`.

Figure 3 gives an UML diagram with the class hierarchy of the most important classes of the package `charlie.plugin.analyzer` and their super classes, and Figure 4 shows an UML diagram with the class hierarchy of the packages `charlie.plugin.gui` and `charlie.plugin.io`. Classes that do not have a super class either extend `java.lang.Object` or, in the case of `JPanel`, another class of the standard Java library. Furthermore the abstract methods of the classes are given in the diagram, as well as some important methods and constructors.

The plugins are loaded in a non-deterministic order and so far only during the start-up phase. Currently there is no way of loading or unloading any of the plugins at runtime. The order of loaded classes is the following: first all analyzers are loaded, then all rule extenders, followed by the computational dialogs, and finally all readers.

9 Conclusion

The functionality of Charlie, a tool for analyzing Petri nets, can be extended by plugins. In this paper we have described all steps that are necessary to implement a new plugin for Charlie. The different steps comprise the implementation of a new analyzer, a new dialog box, the extension of the set of rules, and a new reader.

We have seen that it is indeed very easy to implement a new plugin for Charlie. It is even easier for the user to install and use the new plugin. The implementation of a new plugin was demonstrated on a running example, an analyzer for a simple criterion of structural boundedness. We have described the implementation of the analyzer and the implementation of a graphical user interface. Furthermore, we have shown how

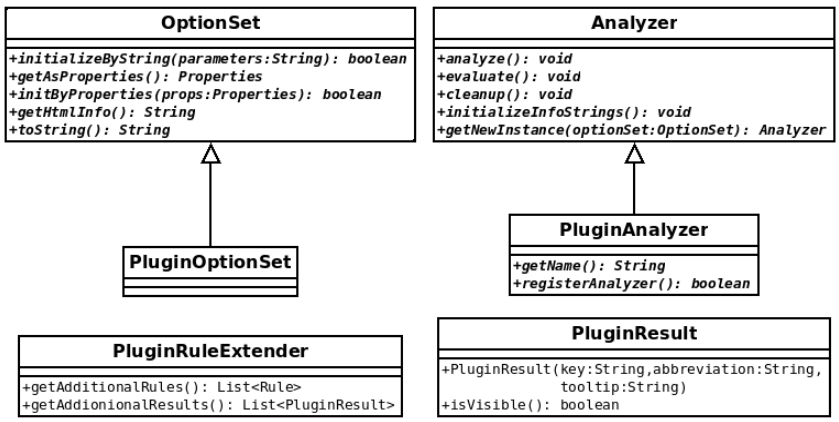

Fig. 3. The class hierarchy and the abstract methods of its classes in the package `charlie.plugin.analyzer`.

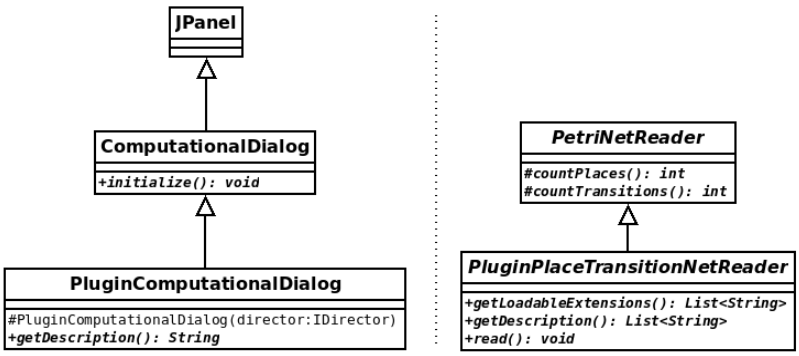

Fig. 4. The class hierarchy and the abstract methods of its classes in the packages `charlie.plugin.gui` (at the left) and `charlie.plugin.io` (at the right).

the set of rules can be extended and how a reader can be implemented. Finally we have outlined how the work is put together to a plugin file and how a user can use the plugin.

In the future we plan to improve the plugin system so that each plugin can have its own version of external libraries. This problem has been described at the end of Section 7. Furthermore we intend to add a mechanism to load and unload plugins at runtime, so that users do not need to restart Charlie when they want to add or remove a plugin.

References

1. Website PIPE. `http://pipe2.sourceforge.net/`.
2. Website Snoopy. `http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Snoopy/`.
3. Website TableLayout. `http://java.net/projects/tablelayout/`.
4. Website TINA. `http://homepages.laas.fr/bernard/tina/`.
5. Parosh Aziz Abdulla, Pritha Mahata, and Richard Mayr. Dense-Timed Petri Nets: Checking Zenoness, Token Liveness and Boundedness. *The Journal of Logical Methods in Computer Science*, 2007.
6. Gianfranco Balbo. *Introducation to stochastic Petri nets*, pages 84–155. Springer-Verlag New York, Inc., New York, NY, USA, 2002.
7. Falko Bause, Peter Kemper, and Pieter Kritzinger. Abstract Petri Net Notation. 1995.
8. René David and Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer, 1. edition, November 2004.
9. Ansgar Fischer. Analyse von zeitbewerteten Petrinetzen mit Erreichbarkeitsgraphen (in German). Diploma thesis, BTU Cottbus, Dep. of CS, October 2009.
10. The Eclipse Foundation. Website Eclipse. `http://www.eclipse.org/`.
11. Andreas Franzke. Charlie 2.0 – a multi-threaded Petri net analyzer. Diploma thesis, BTU Cottbus, Dep. of CS, December 2009.
12. Monika Heiner, David Gilbert, and Robin Donaldson. *Petri Nets for Systems and Synthetic Biology*, volume 5016 of *LNCS*, pages 215–264. Springer, 2008.
13. Monika Heiner, Ronny Richter, Christian Rohr, and Martin Schwarick. Snoopy – A Tool to Design and Animate/Simulate Graph-Based Formalisms. In *Petri Net Newsletter*, April 2008.
14. M. Ajmone Marsan. Stochastic Petri nets: an elementary introduction. In *Advances in Petri Nets 1989*, pages 1–29, Berlin - Heidelberg - New York, June 1989. Springer.
15. Christian Rohr, Wolfgang Marwan, and Monika Heiner. Snoopy - a unifying Petri net framework to investigate biomolecular networks. *Bioinformatics*, 26(7):974–975, 2010. (Advanced Access published February 7, 2010).
16. Martin Schwarick. Ein Werkzeug zur Analyse von Petrinetzmodellen (in German). Diploma thesis, BTU Cottbus, Dep. of CS, September 2006.
17. Peter H. Starke. *Analyse von Petri-Netz-Modellen (in German)*. Teubner Verlag, 1. edition, 1990.
18. Peter H. Starke. Website INA. `http://www2.informatik.hu-berlin.de/~starke/ina.html`, 2003.
19. Peter H. Starke and Stephan Roch. INA, Integrated Net Analyzer, Version 2.2, Manual. 2003.
20. Jan-Thierry Wegener, Ben Collins, and Louchka Popova. Petri Nets with Time Windows: Possibilities and Limitations. *International Workshop on Timing and Stochasticity in Petri nets and other models of concurrency (TiSto)*, pages 63–77, June 2009.