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 large number of analyzers 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 briefly describe the main features Charlie comes with and discuss various features which can be expanded by a plugin. Full details allowing readers to write their own plugins are provided in an appendix, which is available on Charlie's website.
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 (p/t invariants for short, see, e.g., [13]), siphons and traps (see, e.g., [14]) and to construct and evaluate the reachability graph (see, e.g., [20]), just to mention a view basic analyzers. Charlie has been greatly inspired by the tool Integrated Net Analyzer (INA) [25], 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 and Figure 2. A complete overview and explanation of the abbreviations can be found in [26] or [13], and more technical details in [24].

Charlie comes with an intuitive and easy to use Graphical User Interface (GUI), see Figure 1 for a screenshot and its explanation. Furthermore Charlie can be used via the command line interface, allowing a user to integrate Charlie in script files.

Charlie reads Petri nets written in the Abstract Petri Net Notation (APNN, see, e.g., [7]) and Petri nets that are stored in the INA or Snoopy (see, e.g., [15], [22]) file format. Snoopy is a program for drawing, animating 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., [19], [6]). Additionally, there is an interface for writing new readers to load Petri nets from files stored in a different file format.

CONCURRENCY, SPECIFICATION AND PROGRAMMING
M. Szczuka et al. (eds.): Proceedings of the international workshop CS&P 2011
September 28-30, Pułtusk, Poland, pp. 531-554

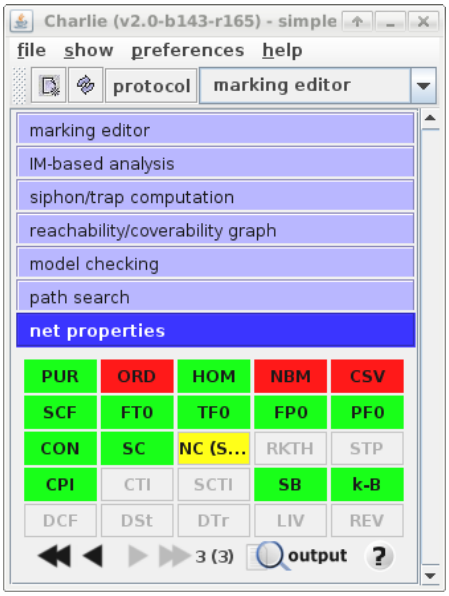

Fig. 1. A screenshot of Charlie's GUI showing the standard features without any extension. Starting from the top one sees the menu bar, followed by some buttons for a quick access to often used functions and the dialog boxes for the different analyzers. At the bottom a special dialog appears, the "net 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.

The development of Charlie began in the year 2006 with the Master Thesis by Martin Schwarick [23]. Since then, Charlie has been improved by some students' projects; e.g., Ansgar Fischer [9] extended Charlie's capabilities by analyzing time-dependent Petri nets (see, e.g., [5], [28]).

In the year 2009 Andreas Franzke [11] introduced a new GUI and redesigned the basic architecture of Charlie's code. 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 analyzing 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.

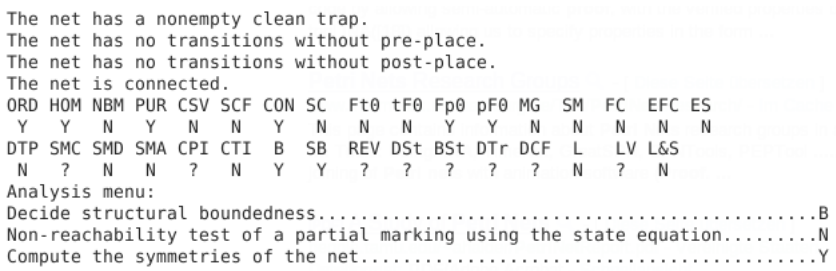

Fig. 2. A screenshot of INA showing a cut-out of an analysis session. One can see that the list of properties shown in INA are similar to the properties used in Charlie.

This also helps to save 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 briefly describe Charlie's features implemented in the standard version of Charlie and give an exhausting overview of Charlie's plugin system.

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 give an overview of the most important features shipped with Charlie in the standard version, i.e., in the version without any plugin extensions. In the following section we state our requirements on a plugin system. Section 5 gives a brief overview on the integration of a plugin system into an existing application. Furthermore some limitations on Charlie's plugin system are stated. After that, i.e., in the sixth section, we give an overview of Charlie's plugin system. Before describing the features that can be extended by plugins we motivate the usage of a plugin system. Finally we summarize our work and give an outlook of our future work.

Full details of the internals of Charlie's plugin system and a detailed description of how to write new plugins can be found in the appendix of this paper, which is available on Charlie's website. The code of the running example from the appendix can be downloaded from that website as well.

2 Related Work

There are several other tools for performing structural and dynamical analysis of Petri nets. Most tools for analyzing Petri nets come with their own editor for drawing Petri nets. Beside of an editor for modifying the set of markings Charlie neither provides any further features for modifying a loaded Petri net nor does it provide a feature for creating Petri nets. The intention is the separation of creating and modifying Petri nets from its analysis.

There are also tools for analyzing Petri nets that 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. By contrast, 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. There are some features PIPE provides that Charlie does not (e.g., analysis and simulation of stochastic Petri nets) and vice versa (e.g., analysis of the rank theorem [21] or the analysis of the siphon trap property [14]). However PIPE does not have a rule system[3] comparable to Charlie's rule system. The rules provided by PIPE cover basic p/t-invariant analysis, e.g., if a Petri net is covered by p-invariants then it is (structurally) bounded (see, e.g., [13]), if a Petri net is covered by t-invariants it is possibly bounded and live (see, e.g., [17]). Furthermore PIPE seems not to consider previously computed results when applying theorems. Charlie however keeps track of previously computed results, either computed by an analyzer or by the application of a rule, and thus is able to compute further properties simply by applying rules to already known properties. The advantage of finding rules is that it is less time-consuming than using analyzers. This approach of using a set of rules is one of the main advantages of Charlie compared to other Petri net analyzing tools.

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. However, TINA has more features for calculating the reachability graph than Charlie. Furthermore TINA provides a tool for simulating (Time) Petri nets.

3 Charlie's Main Features

In this section we give a short overview of the main features of Charlie. We explain the terms analyzer and rule; the centerpieces of Charlie for analyzing Petri nets. Then we give some brief instruction on how to get started using Charlie and give an overview of the most important analyzers implemented in Charlie.

An analyzer is a piece of code that tries to decide properties of a given Petri net based on one or several algorithms. For example if one wants to check if the loaded Petri net is structurally bounded then the well known simplex algorithm is used. Although most analyzers perform a structural analysis, they are not restricted to it. Some of the analyzers can also perform dynamic analysis based on the marking of the loaded Petri net, e.g., by computing the reachability graph the analyzer also checks for the liveness behavior and the reversibility of the Petri net under investigation.

[3] In Charlie a rule is an implementation of a known and proofed theorem. See Section 3 for more information on 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: 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.

After Charlie has been started one can load a Petri net by opening the menu "file" and selecting the menu item "open ...". When one has selected a Petri net with the file chooser a basic structural analysis is performed. This basis analysis only performs computations that can be performed while reading the structure of the Petri net, e.g., the net class is determined, a check whether the net is pure or ordinary. Starting from this basis Charlie directly tries to find rules from its set of known rules that can be applied on the resulting properties. Charlie repetitively tries to apply rules to the set of results until Charlie cannot find a rule that can be applied to the set of results. As already mentioned, Charlie is a 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 3. 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. After the basic structural

Fig. 3. 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".

analysis has finished one can select an analyzer and start its analysis. The analyzers, directly shipped with Charlie, are classified as follows (see Figure 1):

- Analysis based on the incidence matrix of a Petri net (e.g., check for structural boundedness, compute p/t-invariants);
- Analysis of siphons and traps of a Petri net (e.g., minimal siphons, minimal traps, decide siphon/trap property);

- Computation and analysis of the reachability graph (e.g., computing and showing the reachability graph, checking the boundedness and liveness of a Petri net with a specific marking);
- Model checking based on the computed reachability graph (LTL and CTL model checking);
- Computation of paths in the reachability graph (e.g., shortest or longest path from the initial marking to a target marking).

After an analyzer has finished its analysis, it sets a property to either true or false or, if more information is accessible, to a specific number (e.g., k-boundedness gives the actual value if known) or string (e.g., the net class is given as a string). The results can then be seen in the dialog "net properties" (see Figure 1). Furthermore Charlie automatically tries to find rules that can be applied to the set of results as described above.

Beside the already mentioned properties there are still some more properties known to Charlie. Some of these properties are:

- ORD: ordinary (each arc has a multiplicity of 1);
- HOM: homogeneous (for a given place the output arcs have all the same multiplicity);
- CSV: conservative (firing a transition preserves the amount of tokens);
- CON: connected (for every two nodes in the Petri net there exists an undirected path from one to the other);
- SC: stronly connected (for every two nodes in the Petri net there exists a directed path from one to the other);
- CPI/CTI: covered by p/t-invariants (every place/transition belongs to a p/t-invariant);
- LIV: liveness (for every reachable state and for every transition there exists a successor state so that the transition is ready to fire).

A complete list and description of Charlie's properties, analyzers and rules can be found in Charlie's help.

4 Requirements on Charlie's Plugin System

A plugin is a piece of code that adds functionality to a program. We say that a plugin system is a piece of code that is responsible for finding and loading plugins and that is responsible for integrating loaded plugins into the application.

From the definition of a plugin system we can derive some requirements on a plugin system. A plugin system must have at least an interface for extending the main part of the application. Furthermore a plugin system is responsible for loading external code at runtime, i.e., code that is not on Java's class path while starting the application. The loaded code must be integrated into the running or starting software.

Every application has its own set of additional requirements on a plugin system. In the case of Charlie we added the following properties as must haves:

(i). A possibility for accessing basic parts of the main software, e.g., starting an analyzer, evaluating the results of a finished analyzer, adding information to the log file, etc.
(ii). A basic framework to work with, e.g., data representation of a Petri net and its basic components like places and transitions, basic algorithms like the Gauss algorithm or the simplex algorithm, etc.
(iii). It must be possible to put external libraries into the plugin file. The libraries should be placed into the plugin file as they are, i.e., in general as a jar file;
(iv). Plugins must be able to share code, but each plugin must be able to have its own version of library.

While the requirement (i) usually requires some additional work on the structure of an existing software, the second additional requirement is usually given per se. In the case of Charlie there were only minor adjustments necessary to meet requirement (i), as requirement (ii) was automatically given. This is due to the architecture of Charlie; the separation of its main layers: the data representation, the internal logic and the visual representation of the data. Furthermore it helped that similar tasks extended one basic class, e.g., all analyzers extend an abstract class `charlie.analyzer.Analyzer`.

For fulfilling requirement (iii) one already needs a custom class loader, since the standard Java class loaders are not able to load classes from jar files placed in a jar file. The fourth additional requirement is not a trivial task in Java and our solution of this problem is described in detail in Section A of the appendix.

Before we started to write our own implementation we have looked at some existing plugin frameworks. The frameworks Java Plugin Framework (short: JPF) and Java Simple Plugin Framework (short: JSPF) looked quite promising. Unfortunately, the framework JPF seems to be no longer supported since their latest deploy was four years ago. This was the main reason why we do not use this framework. The second framework, JSPF, is an easy to use and easy to setup framework which is still under development. We decided us against this framework since it neither fulfills requirement (iii) nor requirement (iv).

5 Integration of a Plugin System

The effort needed to integrate a plugin system into existing software mainly depends on the given design of the software. Furthermore it depends on whether one requires that plugins can be activated and deactivated during runtime or one requires that plugins are loaded during start-up only. In the case that one wants plugins to be activated and/or deactivated during runtime not only the plugin system must support this but also the application requires more adjustments than it is the case when plugins are activated only when the application is started. Especially if plugins shall be able to be deactivated during runtime there are some non-trivial problems one has to consider. Some of these problems are:

- Are there plugins depending on the plugin that is deactivated?
- Are there pending or running computations one has to stop?

- Did the plugin start any threads that need to be stopped?
- Has the plugin opened any window that needs to be closed?

While the first and the second point can be handled by the plugin system and application, respectively, the third and fourth point cannot be handled by neither the plugin system nor the application. Therefore this short list of questions already shows that the application and the integrated plugin system cannot handle all situations by themselves and thus it requires the developer of a plugin to put more effort into writing a plugin.

Currently, activating plugins is only possible when Charlie is starting. For the future it is planned to implement a system that gives the possibility to activate and deactivate plugins during runtime.

During the initialization phase, plugins are loaded and integrated into the system. There are several phases and in each phase different parts of a plugin are loaded: analyzers[4] are registered to the system, the set of rules[5] is extended, dialogs[6] are integrated into the GUI, and additional readers[7] are registered to the system.

A plugin does not need to provide all of these features. In fact a plugin does not need to provide any of these features, it can also provide algorithm or data structures that then can be used by other plugins[8]. At the moment there is no dependency check implemented and thus dependencies should be well documented. A plugin dependency check is planed for the future.

6 Charlie's Plugin System

Although Charlie is shipped with many analyzers, it cannot cover every single aspect of analyzing Petri nets. Furthermore Charlie's core analyzes classical Petri nets only, i.e., colored Petri nets (see, e.g., [16]), time-dependent Petri nets, continuous Petri nets (see, e.g., [8]), etc. are not supported. When a Petri net file is loaded which contains an extended Petri net then the additional information is discarded and the skeleton (the classical Petri net without any extensions) of the extended Petri net is considered only.

Despite the fact that Charlie is free of charge, currently it is closed source software. Thus it follows that a user, who needs an extension to Charlie, first has to contact the authors of Charlie and then has to wait for an updated version, containing the requested feature, to be deployed. Sometimes this process might be too slow or the authors of the software decide not to implement the requested feature (e.g., it is too time consuming, feature is of interest for the authors or whatever other reason[9]). However, also open source software is usually not easy to be changed or easy to be extended so that it fits the needs of a user.

[4] See Subsection B.1 of the appendix for more information on writing analyzers.
[5] See Subsection B.3 of the appendix for more information on expanding the set of rules.
[6] See Subsection B.2 of the appendix for more information on writing dialogs.
[7] See Subsection B.4 of the appendix for more information on providing a custom reader.
[8] See Section A of the appendix for more information on sharing code between plugins.
[9] This does not mean that bugs or feature request should not be reported. The authors of Charlie welcome every bug report and suggestions for improvements.

Adding every analyzing method and extension of Petri nets to Charlie leads to another problem: the usability. Having too many options and dialogs packed into one application is usually too much for an average user, especially if many features are designed for specialist of a very specific field.

Due to the problems mentioned above we decided to implement a plugin system for Charlie. This keeps the basic core version of Charlie small and clear but also gives users the possibility of extending Charlie, either by their own code or by plugins provided by other users. Extensions for analyzing a special kind of transition conflict graphs (see, e.g., [27]) of Petri nets with capacities, or analyzing Time Petri nets have already successfully been implemented using Charlie's plugin system. Figure 4 gives an example of a plugin used within Charlie.

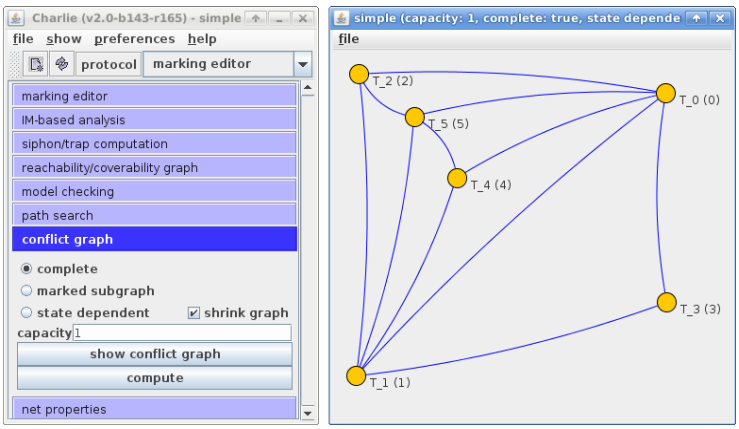

Fig. 4. A screenshot showing Charlie's GUI with an integrated plugin ("conflict graph"). The plugin opened a window showing a special kind of transition conflict graph of a Petri net with capacities. In this transition conflict graph every node corresponds to a transition; two nodes are linked by an arc if there exists a (not necessarily reachable) marking so that the transitions, which correspond to the nodes, are both enabled in that marking.

Charlie's plugin system is easy to use. In order to install a plugin, a user simply needs to copy the plugin file into a specific plugin folder of the installation path of Charlie, called `plugin`. The plugin file itself is a zip file containing the code and data of the plugin. If a plugin is no longer needed or wanted a user simply removes the plugin file from the plugin folder. Currently there is no mechanism for activating or deactivating plugins during runtime therefore one has to restart Charlie in order to activate (or deactivate) a plugin[10].

When Charlie is started it automatically scans the plugin folder for plugins. More precisely, during the initialization phase Charlie searches for files with the file endings ".zip", starting in its plugin folder. The search for plugin files is recursive and thus the plugin files can also be stored in sub-folders, e.g., for sorting plugins by the developer

[10] See Section 5 for more information on this topic.

of plugins or by topic. If the folder contains a file with a ".zip" ending then Charlie assumes that this file is a plugin file and Charlie tries to load the content of this file. In every plugin file there is a configuration file. By searching for this file Charlie can make a quick check whether this file is indeed a plugin file. If the file is a plugin file Charlie tries to load the content of the plugin and if it was loaded successfully[11], it is integrated into the system and the user can access the provided functionality of the plugin. Otherwise, i.e., if the content of a zip file does not have a configuration file, Charlie does not load its content immediately. However there is still a class loader ready to load content from the file since it is possibly an archive file sharing code with other plugins[12].

At the moment there is no dependency check for plugins. Therefore, if a plugin depends on another plugin, and this plugin is missing, Charlie does neither warn the user nor does Charlie unplug the broken plugin. This can lead to a misbehavior of the plugin and even to a crash of Charlie. Implementing a dependency system is planned for the future.

There are several features that can be extended by a plugin. A plugin can extend either all of the following features, only some, or even none at all.

The features most often extended are Charlie's analyzers[13] combined with a GUI component[14] to start the implemented analyzer.

Another feature that can be extended by a plugin is the set of rules[15], i.e., if a researcher finds an interesting theorem that has not yet been implemented by Charlie, the researcher can write a small plugin that extends the set of rules by his or her theorem. This can become quite powerful since applying a rule is less time-consuming than running an analyzer. The functionality of Charlie's rule system is briefly described in Section 3, for more detailed information on Charlie's rule system see [11].

The fourth feature that can be extended is the set of file format readers[16]. A file format reader is a piece of code that is able to read Petri nets stored in a specified file format. With this extension it is possible to read Petri nets from files that are used by Petri net editors using a file format other than Snoopy or INA. This opens the possibility to users to use their favorite program for editing Petri nets while using Charlie for analyzing the nets. When a reader is provided by a plugin, a Petri net can be imported by the menu item "file → import...". If there is no plugin loaded which provides a reader then this menu item is hidden.

7 Conclusion

The functionality of Charlie, a tool for analyzing Petri nets, can be extended by plugins. In this paper we gave a brief overview of the main features of Charlie. This

[11] It is still possible that an exception is thrown and thus the loading process fails, e.g., if a needed class has not been put into the zip file.

[12] See Section A of the appendix for more information on Charlie's class loaders and code sharing mechanisms.

[13] See Section B.1 of the appendix for information on writing an analyzer.

[14] See Section B.2 of the appendix for information on writing a computational dialog.

[15] See Section B.3 of the appendix for information on writing a rule.

[16] See Section B.4 of the appendix for information on writing a reader.

overview is based on Charlie's core without any extensions. Then we stated our requirements to a plugin system and briefly described the integration of the plugin system into the application. Afterwards we motivated the usage of a plugin system and described Charlie's plugin system. We have given a brief description of some important internal mechanisms Charlie uses for loading new plugins. Furthermore we have seen that it is very easy to install and uninstall plugins. Full details of how to write plugins for Charlie are given in an Appendix, available at Charlie's website.

For the future we plan 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. Furthermore we intend to implement a dependency system for Charlie so that Charlie has the opportunity to unplug a plugin that depends on a missing plugin.

Acknowledgement

The authors want to thank the anonymous reviewer for comments and constructive critics that helped improving this paper.

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. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.
13. Monika Heiner, David Gilbert, and Robin Donaldson. *Petri Nets for Systems and Synthetic Biology*, volume 5016 of *LNCS*, pages 215–264. Springer, 2008.
14. Monika Heiner, Cristian Mahulea, and Manuel Silva. On the Importance of the Deadlock Trap Property for Monotonic Liveness. In *Int. Workshop on Biological Processes & Petri Nets (BioPPN), satellite event of Petri Nets 2010*, June 2010.
15. 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.

16. Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 1.* Springer-Verlag, Berlin - Heidelberg - New York, 1997.
17. Kurt Lautenbach and Hanno Ridder. Liveness in Bounded Petri Nets Which Are Covered by T-Invariants. In *Application and Theory of Petri Nets*, pages 358–375, 1994.
18. Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.
19. Marco Ajmone Marsan. Stochastic Petri nets: an elementary introduction. In *Advances in Petri Nets 1989*, pages 1–29, Berlin - Heidelberg - New York, June 1989. Springer.
20. Ernst W. Mayr. An Algorithm for the General Petri Net Reachability Problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 238–246, New York, NY, USA, 1981. ACM.
21. Laura Recalde, Enrique Teruel, and Manuel Silva. On Linear Algebraic Techniques for Liveness Analysis of P/T Systems. *Journal of Circuits, Systems, and Computers*, 8(1):223–265, 1998.
22. 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).
23. Martin Schwarick. Ein Werkzeug zur Analyse von Petrinetzmodellen (in German). Master thesis, BTU Cottbus, Dep. of CS, September 2006.
24. Peter H. Starke. *Analyse von Petri-Netz-Modellen (in German).* Teubner Verlag, 1. edition, 1990.
25. Peter H. Starke. Website INA. `http://www2.informatik.hu-berlin.de/~starke/ina.html`, 2003.
26. Peter H. Starke and Stephan Roch. INA, Integrated Net Analyzer, Version 2.2, Manual. 2003.
27. Luis Torres and Annegret Wagler. Encoding the dynamics of deterministic systems. *Mathematical Methods of Operations Research*, 73:281–300, 2011.
28. 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.

Appendix

In the appendix we give detailed information on the internals of Charlie's plugin system and detailed information about writing a plugin for Charlie.

The next section briefly describes Java's class loading mechanism, followed by a detailed description of Charlie's class loader. Furthermore we describe the features of Charlie's plugin system and discuss its limitations.

In the second section of the appendix, detailed information about writing plugins for Charlie is given. The complete procedure is illustrated by a running example: a very simple analyzer for checking whether a Petri net is structurally bounded. The first four subsections give detailed information about the implementation of the four expandable features: analyzers, computational dialogs, rules and readers. After describing the implementation of the different parts of a plugin we explain how a plugin file is created and configured. In the final subsection we give an overview of the class hierarchy of the classes that can be extended.

A Details on Charlie's Plugin Class Loaders

The way a plugin is loaded and integrated into the software can be different, resulting in different behavior about the ability of sharing code between plugins. It is possible to absolutely separate the plugins or to integrate all loaded plugins into a common namespace. Using the first approach it is, in general, not possible for plugins to share common code, e.g., it is not possible in plugin P_1 to use an already implemented algorithm from plugin P_2. By contrast, it is possible that each plugin has its own set of additional libraries. In the second approach this is, in general, not possible, i.e., if a plugin P_1 uses a library *lib* in a specific version and a plugin P_2 uses the library *lib* in a different version then it is very likely that there are conflicts between the different versions of the library. As a result some of these plugins might not work at all or even worse, they give wrong results. For Charlie we decided to implement a loading mechanism that is in between of the methods described above. More precisely, we want plugins to be able to share common code but we also want each plugin be able to have their own version of a library without getting in conflict with other plugins.

In order to understand the problem of sharing the common code of a plugin but separating the code of its external libraries, it is useful to understand Java's class loading mechanism. Therefore we give a brief description about the standard functionality of Java's class loading system. For a detailed and exhausting description of Java's class loading mechanisms see [12] and [18].

In Java a class loader is a class that is responsible for loading classes and interfaces, i.e., a class loader tries to find binary representations of classes, e.g., by creating binary representations "on the fly" or by loading binary representations from previously created class files from the file system, and constructs object representations of the classes or interfaces from these binary representations. There are two kinds of class loaders[17]: the bootstrap class loader and the user defined class loader. Unlike the user

[17] One can find documents on the Internet where three class loaders are described: the bootstrap class loader, user defined class loaders and the system class loader. The system

defined class loaders, in a Java virtual machine (short: JVM) there exists only one bootstrap class loader.

Java's default class loaders can be described as follows: if the class loader has a parent class loader (every class loader has a parent class loader except the bootstrap class loader) then the parent class loader is asked to load the class. If the parent class loader cannot find the class then the class loader itself tries to find a class with the given name. In the case that this fail as well, an exception is thrown. This behavior tries to ensure that, given a class name, always the same class loader loads the class. Due to requirement (iv) this attempt is not suitable for us. This becomes clear by examining the next problems.

Another problem that occurs is how classes, referenced by a class, are loaded (see, e.g., [18]). Let us consider a class C and a class D, where C is referenced by D (e.g., C is a type of a field of D or a return type of a method of D). Furthermore let D be loaded by a class loader L. Then L initiates the loading of the class C. This concept is visualized in Figure 5.

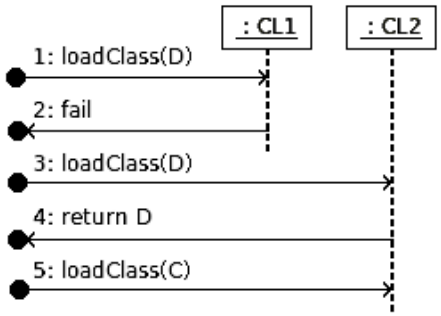

Fig. 5. A diagram showing which class loader loads classes referenced by another class. In this diagram C is a class that is referenced by a class D. The class loader $CL2$ loaded D and thus $CL2$ is called for loading C as well. $CL1$ never tries to load C.

Furthermore the default behavior of a class loader is to ask its parent for loading a class but its children are never asked. Let us consider the following example: let L_1 be a class loader and let L_2 be its parent class loader. Furthermore let C be a class that can be loaded by L_1 but cannot be loaded by L_2. If the method `loadClass(String)` from L_2 is called first then the class is not found, resulting in a `ClassNotFoundException`. Conversely, if class loaders would ask their children for loading a class then the returned class would be chosen non-deterministically (assuming a class loader has several children and some of them are able to load classes with the same name). From this behavior also follows that two class loaders L_1 and L_2,

class loader is usually responsible for loading classes on the class path and usually it is the parent class loader of all other user defined class loaders. However, from a JVM point of view, the system class loader is a user defined class loader as well.

both having L as their parent class loader, are able to share code of a class C if only if L (or its parent class loader) has loaded C.

Finally it is important to know, if code shall be shared between plugins, how classes are identified. In Java a class is determined by a pair of its fully qualified name, i.e., the name of the package and the name of the class, and the class loader that defined the class, i.e., the class loader that on that `loadClass(String)` was called and that called the method `defineClass(..)`. Thus it follows that classes that were defined by two different class loaders are always different. This is also true if the classes have the same fully qualified name and even if the byte code of these classes are equal.

To clarify that sharing code between plugins and having each plugin be able to have its own version of a library is not an easy task, let us consider the following example.

Example 1. Let P_1 and P_2 be two different plugins and let C be a class contained in P_1 and let D be a class contained in P_2 referencing C. Furthermore let lib_1 be a library contained in P_1 and let lib_2 be a library contained in P_2 both containing a class K but in different versions (see Figure 6). In this example let C reference to K in lib_1 and let D reference to K in lib_2.

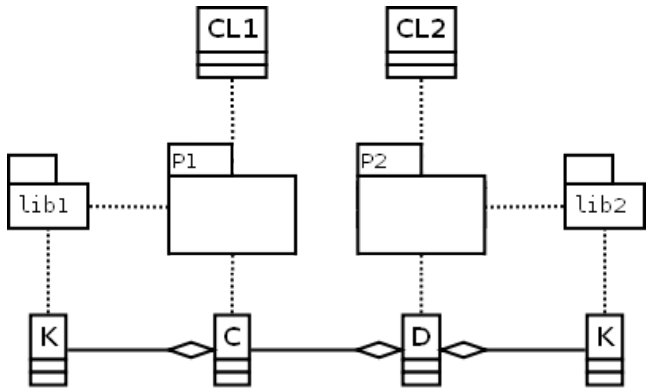

Fig. 6. A diagram showing a problematic setup when using Java's standard class loaders only. In this setup either the class loader of class D cannot find class C or only one version of the class K is loaded, i.e., either the class K contained in $lib1$ or the class K contained in $lib2$ is loaded.

If for C and D the same class loader is used then it is impossible to distinguish between the two versions of K, i.e., either K from lib_1 is loaded or K from lib_2 is loaded.

Conversely, if there is a class loader L_1 for P_1 and a class loader L_2 for P_2 then K contained in lib_1 is loaded by L_1 and K contained in lib_2 is loaded by L_2, but the class D cannot "see" the class C (using the default class loaders from Java only).

For this reason we decided to implement a manager that handles the communication between the class loaders of each plugins, i.e., each plugin loaded has its own class loader and all class loaders have a reference to the manager. If a plugin class loader cannot find a class then the manager is "informed". The manager then "asks" all other plugin class loaders if they can load the class. If none is able to find the class then the parent class loader of the plugin class loader is asked to load the class.

This practice ensures two things:

(i). Every class D contained in a plugin P referencing to a class C contained in P finds the class C;
(ii). Every class D contained in a plugin P_1 referencing to a class C contained in a plugin P_2 finds the class C.

Using the technique described above, every library a plugin uses is shared as well. But this will have the same side-effect that we initiately tried to avoid, namely that it is uncertain which version of a library is actually loaded. Due to this reason we modified the plugin loader so that it returns a class to the manager if and only if the class loader finds the class to be loaded in the plugin archive and it is not in a library in the plugin. Any "internal" calls for loading a class, i.e., if a class in a plugin references to a class in a library in the same plugin, still returns the class; assuming it is contained in the plugin.

As already said above the defining class loader of a class C is associated with C. Therefore it is important that the manager does not define a class by itself.

This procedure does not work when there exist classes in several plugins that have exactly the same names, i.e., their names are equal and they are in the same packages. In this case it is non-deterministic which class is loaded. However using Java's naming convention for packages this problem should not occur.

B Writing a Plugin for Charlie

B.1 Writing an Analyzer

In this subsection 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. 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() {
  if (bounded) {
    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 AnalyzerManagerFactory.getAnalyzerManager().register(
    new SimpleStructurallyBoundedAnalyzer(),
    new PlaceTransitionNet(),
    new SimpleStructurallyBoundedResultSet());
}
```

As a final note we want to mention that currently it is not possible to exchange analyzers, i.e., if an analyzer is integrated into the Charlie then it is not possible to remove it and replace it by another analyzer.

B.2 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.

Writing a computational dialog is always optional, e.g., plugins that extend the set of rules only (see Subsection B.3), do not need to provide a dialog. However, if a plugin provides an analyzer but it does not provide a computational dialog then it becomes hard for a user to invoke this analyzer.

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,TableLayoutConstants.FILL,5},
    {5,TableLayoutConstants.PREFERRED,5}};
  setLayout(new TableLayout(size));
  JButton computeButton = new JButton("compute");
  final Initiator thisInitiator = this;
  computeButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      AnalyzerManagerFactory.getAnalyzerManager().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 `AnalyzerManagerFactory.getAnalyzerManager().compute(Object, Object, OptionSet, Initiator)` or the method `AnalyzerManagerFactory .getAnalyzerManager().compute(OptionSet)` to provide the analyzer with the settings.

B.3 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 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.

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, continuous Petri nets, 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 and about the implementation of new rules can be found in [11].

B.4 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;
}
```

and

```
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.

B.5 Deploying a 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.

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 cannot be loaded.

B.6 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 7 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 8 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.

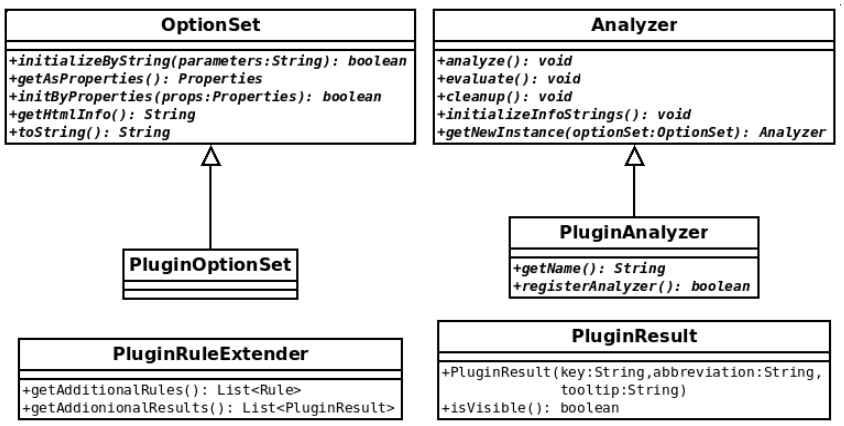

Fig. 7. The class hierarchy and the abstract methods of its classes in the package `charlie.plugin.analyzer`.

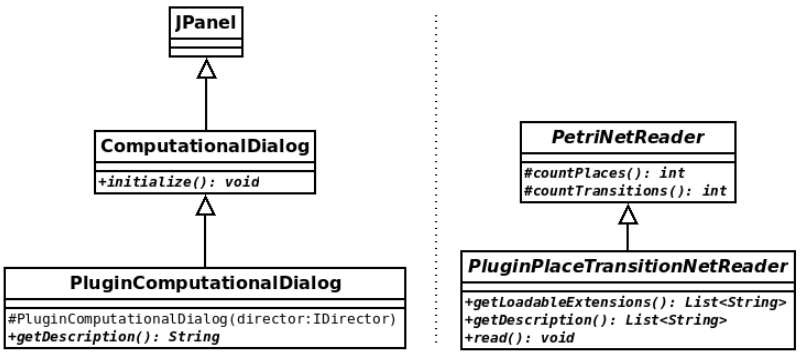

Fig. 8. The class hierarchy and the abstract methods of its classes in the packages `charlie.plugin.gui` (at the left) and `charlie.plugin.io` (at the right).

The 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.