# DIPLOMA THESIS

# Charlie 2.0 - a multithreaded Petri net analyzer

by Andreas Franzke
Andreas.Franzke@tu-cottbus.de

for the chair
Data Structures and Software Dependability
at the
Brandenburg University of Technology at Cottbus

Cottbus, September 30, 2009

I hereby declare that I have written this document by myself without the help of others and have only used the literature and software named in the references.

Andreas Franzke, Cottbus 30.09.2009

**Abstract**

This thesis describes the actions performed to transform the existing version of Charlie, a software tool to analyze Petri nets, into a better usable and extendable software. It is the aim to clean up the structure of classes and packets and create a basis for new extensions. Here, the algorithms of the existing analyzers are not touched or optimized, only the organization of the analyzers is overworked. Also the data structures are not examined.

As a raw model for the new version, the prototype created in my study assignment "Concept for redesigning Charlie 1.0" [AF2008] is used. The new functionalities of the prototype are incorporated into the new version, as there are the GUI generator for command line tools and the marking editor.

The product of my work is a new version which guides the implementer of new features and eases the integration. The structure of Charlie is now clearer and better maintainable. The benefit for the user is the improved usability.

The new version offers handling and managing of analysis threads, a way to define deduction rules for Petri net properties and a textual interface based on parameters to start analyses from a command line.

# Contents

# List of Figures

# List of Tables

# Part I
# Analysis of the existing software and plans for the reengineering

## 1 Introduction

**What is Charlie?** Charlie is a software, which performs several analyses on Petri nets. It is able to determine certain properties of those nets. Charlie is the product of the thesis from Martin Schwarick [MS2006]. It was firstly created with few features and later extended by a few more. Since those features and their integration were not initially planned the implementation had a large impact on the program's structure. The GUI elements were created as needed and did not follow a repeating pattern or a global plan. So the use of the program was not very comfortable. The inexperienced user was confronted with unclear and inconsistent program behavior. When looking behind the GUI the structure of the software was inconsistent in some parts and some features were integrated, but not throughout the whole program. The class structure was not prepared for future extensions. Furthermore the program was restricted to only one analysis at a time. Especially with longer lasting analyses, like the construction of a reachability graph, modern computers with more than one processing core, are not used efficiently.

Then the further extension of the program became more and more problematic, also new features were wanted and the use should become more user friendly. That's were my study work started off. The requirements for a new version were:

- A new extendable GUI interface which is able to integrate all existing features in a consistent and convenient way.
- An editor to change the amount of tokens on a place and to use the modified net with all analyses.
- A user interface to use command-line based external tools. Which works without typing in the necessary parameters and their values by hand. The interface should be extendable for further tools or changes in the parameters.

So a prototype that incorporated those features and showed the new user interface was created. The prototype proved it's usability. The implemented features showed the GUI concept. This prototype was chosen as the basis for the final version. During the implementation ideas concerning the organization of analyzers and modules were created.

The requirements for the final version were sometimes changed and extended.

**Which aims should be reached?**

- A concept for managing and controlling analysis threads.
- A rule system to show the effects of determined net properties. The derived properties should be presented to the user. This rule system should be easily editable and extendable.
- Integrate all features from the prototype and the existing Charlie version into the final version.
- Each analysis can be started by using command-line parameters, without displaying GUI elements.

These requirements seem to be quite simple and easy to integrate, but they lead to fundamental changes in the source code and the class structure.

The analysis thread concept for example made large modifications necessary. In the initial version analyses were sometimes threads, sometimes not. The analyzers wrote all their output directly to the protocol, results were stored in a global result set. Each analyzer could only be instantiated once. The final version should be able to instantiate a type of analyzer more than once and manage the parallel execution of analyzers. So results could not be stored in a global object anymore, the output had to be captured and stored, until the analyzer finishes. The access to several objects had to be coordinated. Furthermore, the existing analyzers incorporated features, which do not necessarily belong to an analyzer, like loading nets or returning properties of Petri nets.

In between further plans for extensions were made, certainly the integration of those had to be simple.

So you see that almost every part of the software hat to be checked and lots of them had to be modified. The only exception were the algorithms of the analyzers and the data structures.

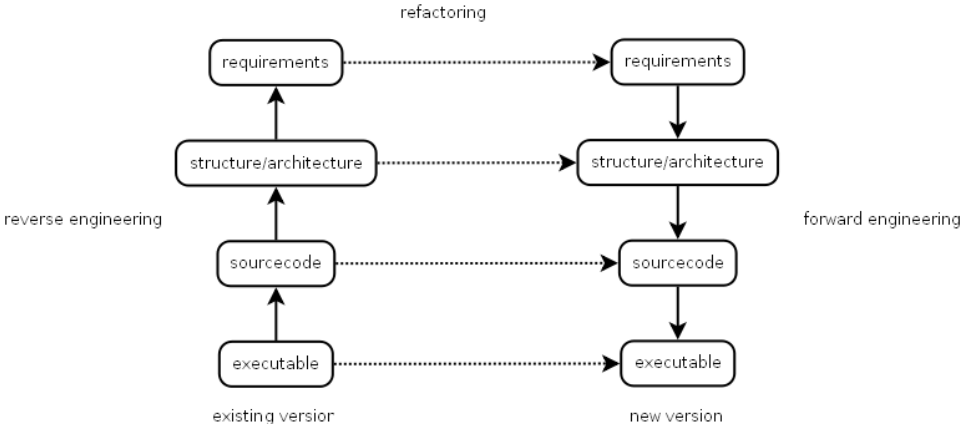


Figure 1: Process of reengineering.

# 2 Reengineering plan

## 2.1 Why reengineering became necessary?

The software was developed by a single developer, next to his normal work. Initially it was not intended to grow so big. So initially no concrete plans existed which incorporated this growth. Today more extensions are planned and the number of developers is planned to grow. A certain number of students will work on modules which extend Charlie by new functionality. The initial version is found to be not ready for these extensions.

Usually reengineering becomes necessary if: "Your software comes to a point where extensions and repairs get more and more difficult and uncontrollable, you will make yourself familiar with questions about software reconstruction..." [BSB2008, p. 233]. Reengineering means three possible actions, which can be used [BSB2008, p. 234]:

- reverse engineering
- refactoring
- forward engineering

In this case, all of them are used at different parts of the software.

**Reverse engineering** is used to get familiar with the programs structure and the interactions and relations between all parts of the software. It means to gain information from lower levels and to create a higher level of information. Since both a running version and source code are available as documents for reengineering, static and dynamic analyses of the program can be performed. Because the program is driven by a graphical user interface the events within the software are not always linear and program behavior must be determined

by using both source code and the real program. By using those techniques a good understanding of the program's structure and behavior is created.

**Refactoring** operations transform information from one level to new information on the same level, e.g. source code is modified and creates new source code. Possible operations are removing variables, changing the code style or moving methods from one class to another.

**Forward engineering** transforms the requirements for the new features into source code and then into executable files.

In this case actions from of all of them need to be performed at different levels of abstraction.

## 2.2 What concrete actions have to be taken during this process?

- Get an understanding of the existing program and its structure.
- Identify possible problematic areas in the structure and the source code.
- Define goals for the new structure of the program.
- Define additional features for the new program version.
- Design a new structure that reflects the defined goals.
- Adapt the existing classes to the new structure.
- Define the requirements for the new features.
- Implement the new features.

**What are the benefits of the refactoring?** In the current state understanding and using Charlie is complicated. The extension is difficult since there is no defined structure. The refactoring should therefore produce a usable extendable new program version, with a documented software architecture.

**How to inspect the software?** The only documents provided were the source code files and binaries. So mainly static analysis and some dynamic analysis is used to understand Charlie.

The book "Software Wartung" names some static methods for understanding a software architecture:

- Analyze package structure, relations between packages [BSB2008, p. 219] and their interconnections
- Analyze dependencies among classes (Which classes are used by a class?) at least for the analyzer classes

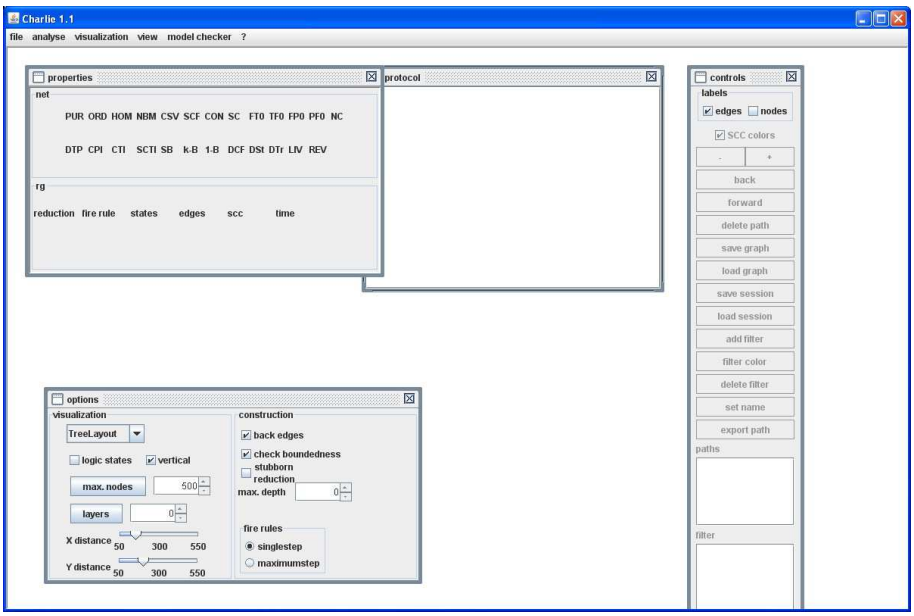


Figure 2: Charlie 1.0 start screen.

- Create a Dependency structure matrix [BSB2008, p.220] and analyze them.

Also the analysis of the source code helps to identify the behavior of the program.

Since the program has about 28000 lines of code [see output listing on page 10] it is almost impossible to recover all of the behavior in an adequate time. So I decided to to start over with the previously created prototype and created a new package for the graphical user interface which got all of the new features. If classes of the Charlie package were useful or necessary they can be transferred to the new packages. I only used and modified classes which were needed to perform the analysis. After finishing the implementation the class structure is examined for unused classes, which are removed then.

## 2.3 The existing GUI

I will explain a few short examples, which give an insight into some problematic aspects of the existing version. To visualize the differences between the versions, take a look at the following pictures (Figure 2). This Figure shows the initial state of Charlie after being started.

As you can see the user interface consists of a single frame with several internal frames. The internal frames: controls and option's relate to two things that are not visible yet. The controls frame contains the buttons and checkboxes, which control the reachability graph visualization frame. In the options window the visualization part on the left side is also responsible for settings that relate to the reachability graph visualization frame. The right part relates to options

which control the construction of a reachability graph. The construction itself is started by using the menu at the top of the frame. Within this simple example several problematic aspects can be named, which collide with the "easy-to-use" of the application. I will only list a few of them:

- Some controls are disabled some are not, although related to the same thing.
- Why are controls visible when they cannot be used?
- The naming of titles is insufficient, e.g. construction or controls. What exactly is constructed? What is controlled?
- The visualization options affect only newly created visualization frames, not existing ones.
- The internal frames seem to be placed quite randomly.

Solutions for those issues could be the following:

- Place the visualization controls inside the controls frame.
- The controls frame only becomes visible, when a reachability graph visualization frame is opened.
- The visualization options get a button labeled with "new visualization".
- The construction options are placed inside a dialog window, which appears if the user clicks construct RG in the analyze menu of the application.
- All titles are modified or replaced by more informative ones.

## 2.4 The new GUI design

During my study assignment [AF2008]ideas about the new look of Charlie were created and discussed. The result of these talks was a GUI design that oriented to the design of GIMP, an open source picture manipulation program [GIMP]. With some independent frames, a main frame with all tools and additional ones that contain more options or visualize something. The user gets a simple and clean interface which doesn't confront him with an overwhelming number of controls.

As we can see (Figure 4), the main window stays at the left side of the screen. The visible GUI elements can be reduced by clicking onto the captions of each sub dialog. The sub dialog for deadlock/trap computation is closed for instance (Figure 3). Each module window can be closed without loosing its contents. If the marking editor is not needed, then the window can be closed and redisplayed if needed. The user gets full control of all windows. Options and controls are now only displayed if the corresponding dialog or window is visible. For example the controls for the reachability graph visualization now only appear in the context of the visualization window.

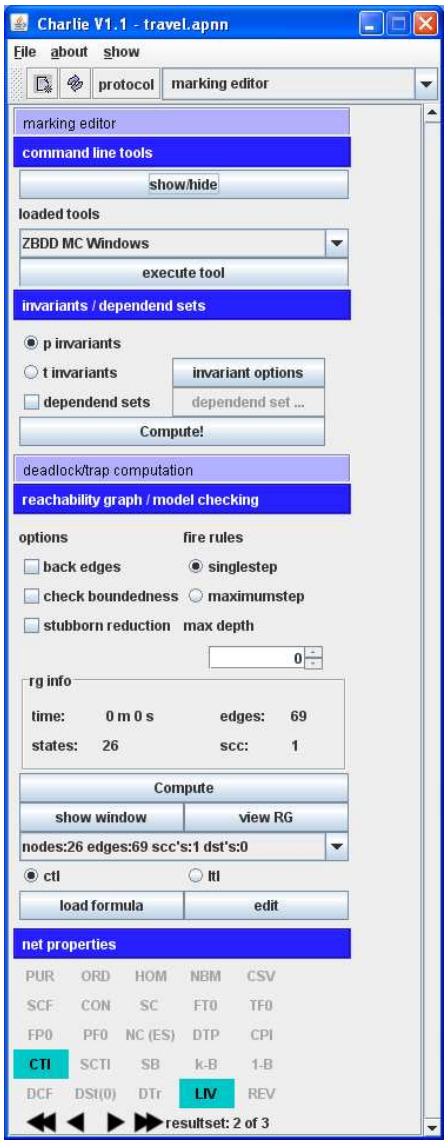


Figure 3: Main window in the new design.

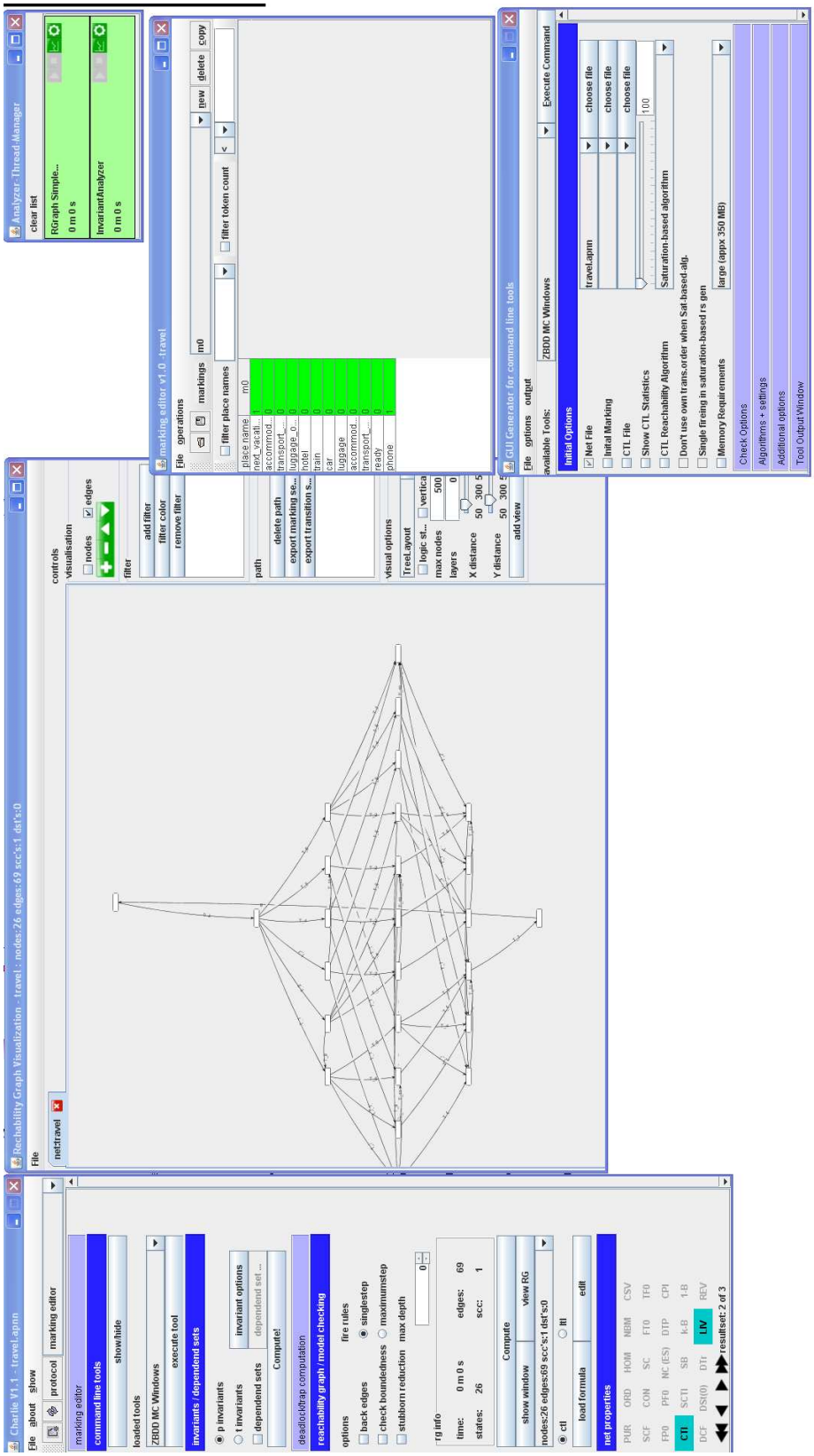


Figure 4: Main window with module windows.

## 2.5 Principles for the GUI design

The book [IW1998] describes a lot of principles that should be followed if a user interface is designed. I tried to consider most of them. As the design shows, all dialogs are made up similar. Each dialog uses an one- or two-column layout and aligns the controls to the left. Buttons are rather big, but so they can be easily hit by the mouse pointer. The big compute button is repeated through the dialogs and the user soon expects a button like this within every dialog in the main window. The old design didn't care for alignment of frames or dialogs.

Wherever possible the following principles were followed:

- Alignment - The layout manager TableLayout [TableLayout] is used instead of the built in ones.
- Hotkeys - Within the menus each menu item should be reachable by pressing keys on the keyboard.
- Tooltips - A description is added for almost every GUI element.
- Tabs are used when possible instead of multiple frames.
- Multiple ways of interacting with GUI elements. When possible mouse or keyboard can be used (e.g. spinner or slider buttons use the mouse wheel and the mouse pointer).
- Colors are only used where necessary in order not to create a colorful distracting interface, but used where reasonable.
- Meaningful hints and question dialogs, if something did not work properly or finished, so the user is informed about the state of the program.
- Hiding of elements is possible if the user does not need them currently.

Realizing these points created a usable interface for the user.

## 2.6 Show nets in snoopy

A small but very useful feature is the menu "show" in the main window, which opens the Petri net editor Snoopy [SNOOPY] with the currently loaded net. Of course Snoopy has to be installed before.

If the executable is not found at the standard place the user can select the new location, this is stored in the properties of the program and remembered the next time Charlie is started.

So the user can open both the editor and the analyzer, apply changes in the editor and press reload in Charlie and the changes can immediately be examined.

# 3 Package and class analyses

## 3.1 Overview

At first an overview about the packages and classes and the size of Charlie is created. The sizes of packages and the number of classes is determined to get an overview over the structure.

Here are some facts, which will later be compared to the final version:

- 13 (sub-)packages
- 237 classes
- the tool SLOCcount [SLOCcount] counted 28324 lines of source code

```
SLOC    Directory    SLOC-by-Language (Sorted)
7048    pn           java=7048
4547    gui          java=4547
3703    ds           java=3703
3530    ltl          java=3530
2778    ctl          java=2778
2402    rg           java=2402
1585    vis          java=1585
1312    filter       java=1312
1299    inv          java=1299
120     dtp          java=120
Totals grouped by language (dominant language first):
java:          28324 (100.00%)
Total Physical Source Lines of Code (SLOC)                =
    28,324
Development Effort Estimate, Person-Years (Person-Months) =
    6.70 (80.35)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 1.10
     (13.24)
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  =
    6.07
Total Estimated Cost to Develop                           = $
    904,490
(average salary = $56,286/year, overhead = 2.40).
generated using David A. Wheeler's 'SLOCCount'
```

As we can see the structure of packages is not very complicated there are only 13 packages (Figure 5) . But if you look at the number of classes the range is quite big. The smallest package has only 1 class and the largest 59 classes.

From this analysis the following questions arise:

- Is the package dtp necessary? Can the class be moved into an other package?
- The packages gui, ltl, pn, rg, vis contain at least 22 classes. Is it useful to split those packages into smaller ones?

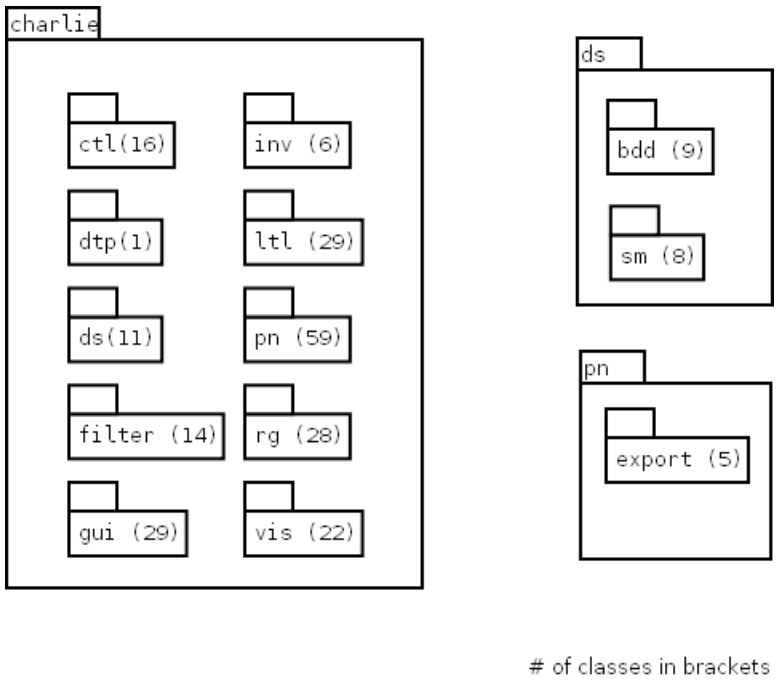


Figure 5: Package structure of Charlie (numbers in brackets = # of classes in the package).

- Is it possible to concentrate only on relevant packages and leave out packages which contain classes that are not needed anymore (Which will probably be the case with the gui package)?

So the static analysis of the existing packages is a good source of information, where problems might occur during the reengineering process.

## 3.2 Relations between packages

Then the relations between packages were examined. This gives an insight in the dependencies between packages. If there are strong dependencies this might indicate a bad-smells. If many classes in one package use many classes in another package, changes in classes might affect a lot of other classes. A pattern like "facade" [BSB2008, p. 45] that manages the interaction between packages and the access to objects would decrease problems if the program needs to be changed or extended (Figure 6). The number next to the arrows are the numbers of classes the package depends on in the other package. High numbers and many arrows hint to packages which are hard to modify since a modification can affect many classes in other packages. Not only modification is more problematic, testing gets also difficult. Because of the many dependencies a module tests is hardly impossible since many classes from outside the module influence the behavior. Also the use may be affected [BSB2008, p. 218-219].

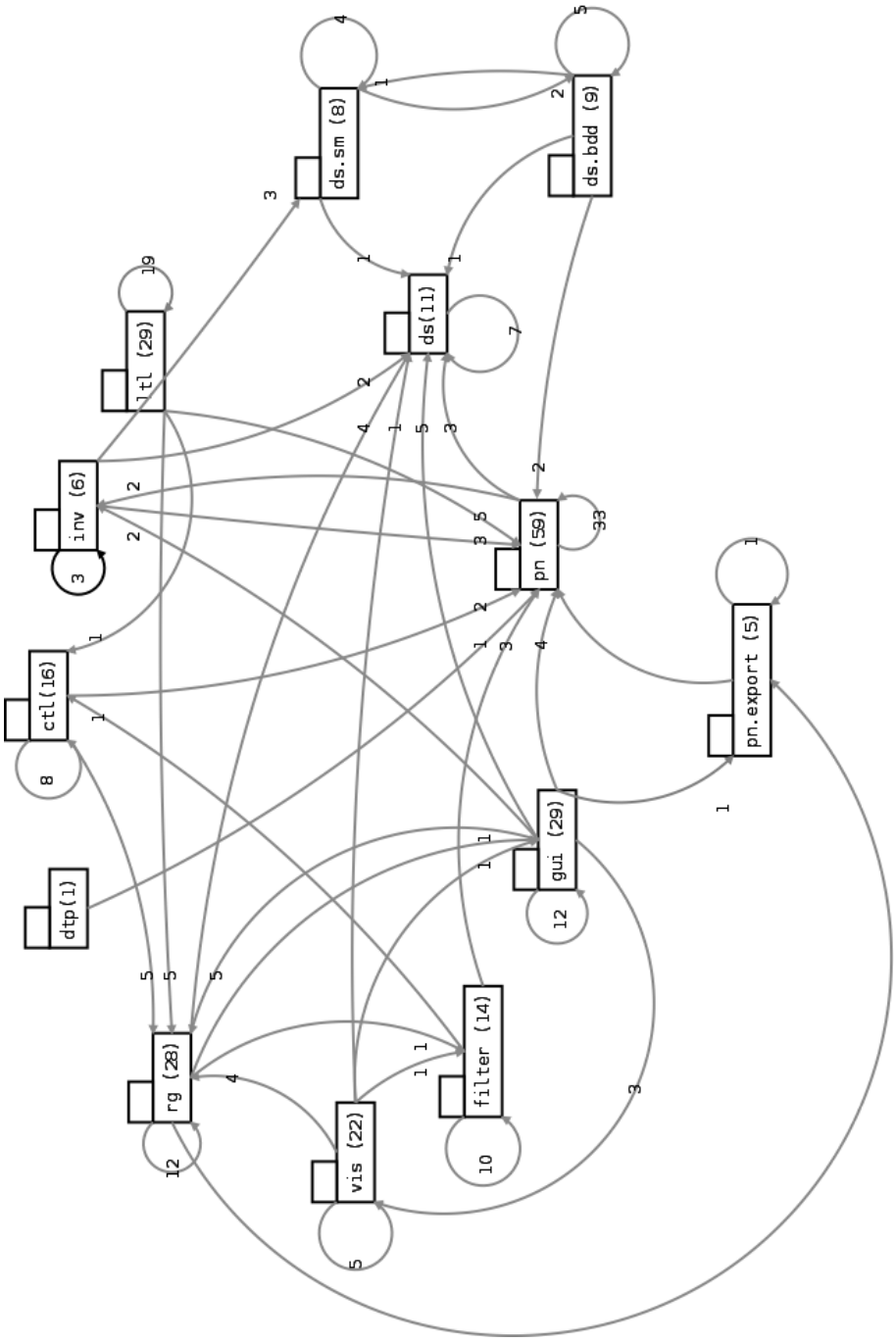


Figure 6: Package dependencies.

As we can see(Figure 6) the three biggest packages also have the most dependencies. The packages pn, gui and rg are strongly connected to other packages. The package gui is not considered anymore since almost all of its classes get obsolete with the new user interface. The package pn should only hold the data classes connected with a Petri net. But this package also holds analyzers, input and output classes. There are also classes which are never used in one of the classes inside the package or which are never used at all.

## 3.3 The new package structure

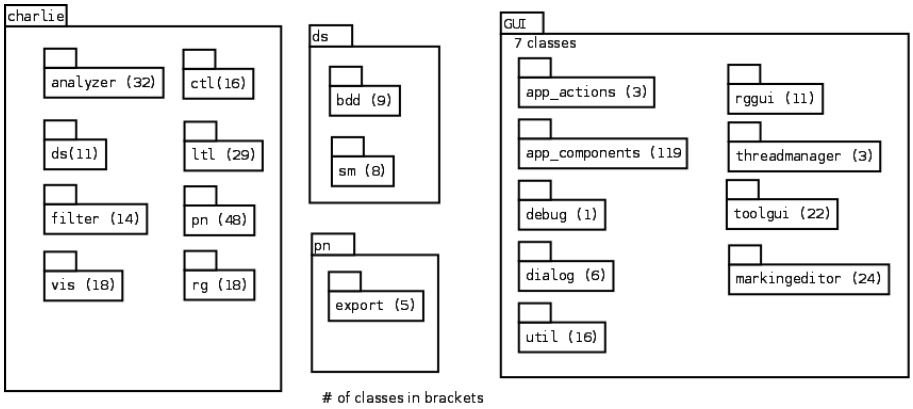


Figure 7: New class structure.

Mostly all GUI related classes are moved to the new big GUI package. The analyzers are placed inside the sub package Charlie.analyzer. This package contains more sub packages, which are thematically sorted.

Now the Charlie package contains:

- 186 classes (before 237, -22,5%)

The new package GUI contains:

- 97 classes

Together there are 283 classes ( + 19,4%).

The lines of code analysis of the final version, with SLOCcount produced the following result (the output was slightly modified and shortened):

| SLOC | Directory | SLOC–by–Language (Sorted) | (Before) |
|---|---|---|---|
| 6674 | pn | java=6674 | (7048) |
| 3681 | ds | java=3681 | (3703) |
| 3517 | ltl | java=3517 | (3530) |
| 3433 | analyzer | java=3433 | (---) |
| 2773 | ctl | java=2773 | (2778) |
| 1508 | vis | java=1597 | (1585) |

```
1432    rg              java=1432               (2402)
1312    filter          java=1312               (1312)
258     top_dir         java=258                 (---)
120     dtp          java=120                 (120)
-------------------------------------------------------
24797   Charlie         java=24797              (28324)


SLOC    Directory    SLOC-by-Language (Sorted)
4640    toolgui         java=4640
2426    rggui           java=2426
1816    app_components  java=1816
1786    markingeditor2  java=1786
748     util            java=963
698     dialog          java=698
685     top_dir         java=685
621     setup           java=621
161     threadmanager   java=367
103     app_actions     java=103
84      debug           java=84
-------------------------------------------------------
14189   GUI             java=14189

Totals grouped by language (dominant language first): java:
          38986 (100.00%)

Total Physical Source Lines of Code (SLOC)
       = 38,986
Development Effort Estimate, Person-Years (Person-Months)
       = 9.36 (112.37)
(Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)
       = 1.25 (15.04)
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)
       = 7.47
Total Estimated Cost to Develop
       = $ 1,265,015
(average salary = $56,286/year, overhead = 2.40).
"generated using David A. Wheeler's 'SLOCCount'."
```

As we can see the lines of code in package Charlie dropped by about 3500 lines. The new GUI package adds more than 14000 lines. The new user interface shows more complex behavior than the old one. Some of the old packages are rather unmodified, perhaps some lines are changed if the use of a class that is replaced has to be modified. The lines of code increased by about 40%, while the class number increased only by 19,4%, so obviously more complex classes are added to the program.

# Part II
# The rule - result system

## 4 Introduction to the rule system

One task of this thesis is to create a system that allows the definition of "rules" and their application to Petri nets. Why that? In the field of Petri nets a lot of properties can be determined, often in different ways. Then the determination of one or a set of properties allows conclusions about the values of other properties. So by a certain set of properties other properties may be deducted by rules. For teaching purposes it is helpful if students get informed about those rules and their application to a Petri net. The learning effect increases if the textual representation of a rule and its application to the properties of the net appears in the analysis protocol. So the following requirements for the rule system were established:

- Rules can be defined inside the source code inside a special class. The definition of a rule should be simple and intuitive. The set of rules should be easily extendable.
- Each rule should have a description, which illustrates what properties are necessary and which will be applied.
- A simple way to apply the defined rules to a set of existing results.
- A basic set of well-known rules, which are maybe already implemented by Charlie.

**What benefits can we expect by using this rule system?**

In the previous version the rules were implicitly applied by Charlie.pn.Analyzer or Charlie.rg.RGAnalyzer. Each of those analyzers is able to determine a rather small set of properties directly and deducts possible further properties directly in its source code. This implies knowledge of almost all properties inside each analyzer class. Each class determined properties which did not directly belong to its direct property set and applied properties which also did not belong to this set. Also if there are different ways to determine a certain property all the deduction rules have to be implemented in different analyzers or each analyzer needs deep knowledge of other analyzers. In the previous version the class RGAnalyzer made intensive use of methods inside class Analyzer, which was designed to mainly determine structural properties. This resulted in a mess of if ... then structures and method calls for properties. Often properties are determined by this method calls, not by the analysis itself. The understanding and extension of these structures is very inefficient and difficult. The likelihood of making mistakes while implementing extensions to this structure is very high. A further problem is that the rules are hidden inside the source code and have

to identified manually. Also rules are scrambled, which means that an if construct checks a certain property and inside the { } brackets other properties are checked by further if constructs. So several rules are checked simultaneously and separation is only possible by analyzing the source code.

The rule system offers the possibility to simplify almost each analyzer. Each one only determines its special set of results and returns its values. The computed set of properties is then merged with the results that have already been evaluated. Together with the new system for analyzers the complexity of analyzers is significantly reduced. Then the rule system looks for rules that can be applied to the complete result set. The analyzers can now work independently from each other, which simplifies or even eliminates the dependencies among different analyzers.

The result is a clear separation of analyzers and rules, a simple system of defining rules and an easy way of applying rules to sets of properties.

## 4.1 A basic set of rules

Before creating an idea of the rule system and its components possible rules have to be found and examined. The examination looks at the values of properties and the types of those values, then which conclusions are drawn. The documents from the lecture about Petri nets at the chair offer several rules.

Rule1 live & bounded => strongly connected

Rule2 conservative => bounded

Rule3 covered by P-invariants => bounded

Rule4 not covered by T-invariants => not live

Rule5 live & bounded => covered by T-invariants

Rule6 not covered by T-invariants & bounded => not live

Rule 1-6 source: [MH2007, page 8]

Rule7 net class == state machine => conservative & bounded

Rule8 state machine & strongly connected & at least one token => live & bounded & reversible

Rule9 state machine & strongly connected & exactly one token => live & safe & reversible

Rule 7-9 source: [MH2007, page 11]

Rule10 net class == marked graph => dynamically conflict free (& covered by P-invariants)

Rule11 marked graph & strongly connected & each elementary circle contains a token => live & bounded & reversible

Rule12 marked graph & strongly connected & each elementary circle contains exactly one token => live & safe & reversible

Rule 10-12 source: [MH2007, page 12]

Rule13 extended free choice & deadlock-trap-property => live

Rule14 extended free choice & ! deadlock-trap-property => not live

Rule 13,14 source:[MH2007, page 14]

Rule15 extended simple & deadlock-trap-property => live

Rule 15 source: [MH2007, page 15]

Rule16 deadlock-trap-property & homogeneous & non-blocking multiplicity => not dead states (not DSt)

Rule 16 source: [MH2007, page 18]

The following additional rules are taken from the source code directly by analyzing.

Rule17 deadlock-trap-property & homogeneous & non-blocking-multiplicity & not extended simple => live
([Charlie] Charlie.pn.Analyzer: lines 443->449)

Rule18 not deadlock-trap-property& (free choice or extendedFreeChoice or state machine or marked graph) => not live
([Charlie] Charlie.pn.Analyzer: lines 450->453)

Rule19 covered by P-invariants => structurally bounded & bounded
([Charlie] Charlie.pn.Analyzer: lines 217 - 224)

Rule20 transitions without preplace (Ft0) => not bounded & not structurally bounded & not safe
([Charlie] Charlie.pn.Analyzer :lines 53 - 57)

Rule21 conservative => bounded & structurally bounded
([Charlie] Charlie.pn.Analyzer: lines 76 - 79)

Rule22 state machine => bounded & structurally bounded
([Charlie] Charlie.pn.Analyzer: lines 115 - 118)

(see Appendix B, page 81)

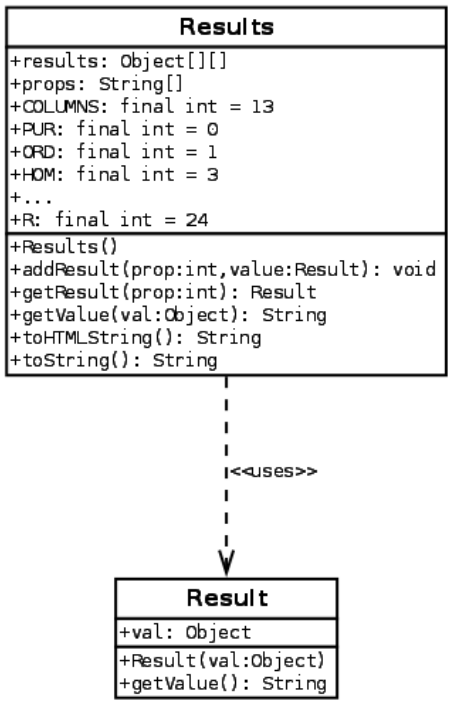


Figure 8: "Old" result system.

# 5 The software architecture for the result system

## 5.1 The original result system

Before introducing the new rule-result-system I want to describe the original architecture of storing results. Martin Schwarick [MS2006] initially introduced a simple system. It consisted of two classes (Figure 8):

Results — The storage of result objects, which were placed inside a two-dimensional array. For identification of single results through static int values were used as indices. Inside the array mainly result objects are stored, but storage of other objects is also possible.

Result — An object that encapsulates a value object and returns a string depending on the value and type of the stored object.

In the previous version of Charlie results are only stored within Charlie.pn.-Analyzer. Only two classes were responsible for setting property values: Charlie.pn.Analyzer and Charlie.rg.RGAnalyzer.
The results are retrieved by calling Analyzer.getResultHTMLString(), which

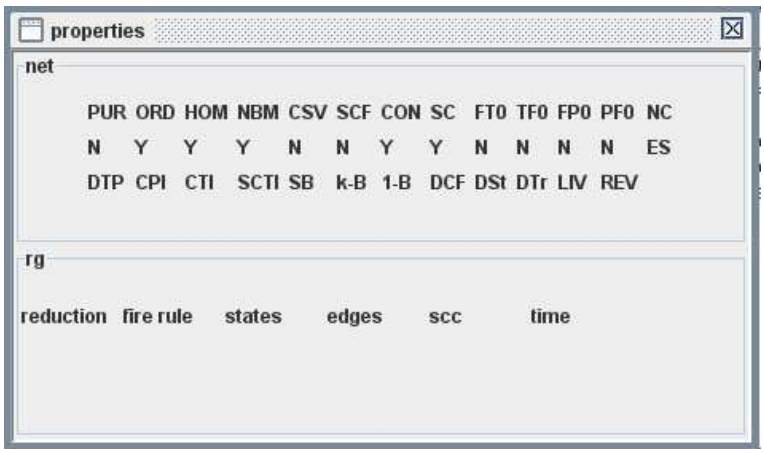


Figure 9: Results panel - net properties panel.

then called Results.toHTMLString(). The single results were presented in a HTML-table within an internal frame of the user interface (Figure 9).

Disadvantages of the existing retrieval and presentation of results:

- The simple textual presentation is not very comfortable to read.
- Only the whole set is displayed and updated. If a single result is set or changed by an analysis, the user cannot recognize which one was set (except for looking inside the output of the analyzer in the protocol window).
- The results are retrieved by using different analyzers, like InvAnalyzer or DeadlockAnalyzer then the result of these analyzers is evaluated in charlie.pn.Analyzer and charlie.rg.RGAnalyzer. So all analyzers depend on each other somehow, which makes changes difficult and offers no consistent way of evaluating results. Moreover knowledge of different topics is necessary in one analyzer. Thematic separation is not achieved.
- Because of the missing rule system, the rules had to be build manually by using complex if-structures. The appliance of rules is not presented to the user so the learning effect is reduced.
- There was only one central Results-object located in Charlie.pn.Analyzer which was updated every time an analysis finished. Safe concurrent access to this object was not possible.

## 5.2 The result system in the prototype of Charlie

My former assignment offered a new way to present the properties of the Petri net (Figure 10). The new window uses colors to represent the current state of a result. In the basic setting light green indicates that a result is set to true,

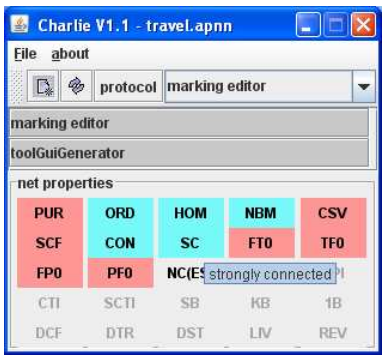


Figure 10: Net properties panel in Charlie prototype.

which formerly was the character 'Y'. Red indicates a property that was set to the value false or 'N'. The colors may be adapted to the users needs, because red might be misunderstood as a bad value, but is not necessarily bad. For example the absence of dead states, which would be indicated by a red colored box, but in fact the absence is seen as a good fact. Furthermore tooltips are used to display more information on the property. This can be easily extended to longer descriptions or definitions for each property. So students don't have to switch between the program and a book to look up definitions each time they need them. Next to colors, strings or integer values can be used to display the value of a property. The value is displayed next to the abbreviation of the property. Unset properties are grayed out and can be easily identified as not determined.

The prototype concentrated mainly on the design of a new graphical user interface, so the way new results are determined and collected was not changed very much. Like the original version the prototype collects all properties in Charlie.pn.Analyzer, then the Results object is passed to GUI.App and the NetPropertiesDialog interprets and presents the values.

## 5.3 The extended rule-result system

The new system introduces some fundamental changes to the way results are collected and displayed. The new system should meet the following requirements.

- Each analyzer should only set properties, which are determined directly by it. No deductions are done inside the analyzer.
- Each analyzer has to set results independently from other analyzers..
- All result sets are passed to a central manager which is responsible for presentation and storage.

- The user can access each single result set, to make a connection to the analyzer which produced the results.
- All deductions (rules) are written in a simple intuitive way and stored in a central managing object.
- The deductions can be easily applied to a set of properties.
- If a deduction is applied, the user is informed about this via a textual output to the protocol window.
- The rules for deduction must allow the use of ordering relations like bigger than ('>'), lower than ('<'), equal ('=='), not equal ('!='), greater or equal ('>=') and lower or equal ('<=').
- There must be a possibility to extend the rule system by automatically called checks, which need further implementation.

In the previous version there was only one current set of properties. Now there is a list of property sets located in the NetPropertiesDialog object, which stores the results of each analysis. So the result sets of each analyzer are distinguishable. This enables the user to see what properties have been set by a special analyzer. The following Figure (Figure 11) shows the classes which store results and rules.

## 5.4 Description of class Result

**Member-variables** The original variable val of the type Object is enhanced by static integer values which represent the possible types of the value object. The method getType() returns an integer value depending on the class type stored in val. The presentation dialog uses the type to decide which way to choose in order to display the result properly.

**Methods** The old version only had a constructor and a method to return the value of the Result object as String representation. The new version needed extensions because a Result object needs more functionality than only returning a string representation. So the following methods have been added.

copy() Tries to create a copy of the Result object. If the object is a Boolean, Integer or String type the created copy is a deep copy with no copied references to the old object. Since the Result object can store any other object too, the copy() method of the object is used in this case and a deep copy cannot be guaranteed.

equals(Result) Compares the Result object to the provided parameter and returns true if the values are equal. Simulates the '==' relation. If the type is equal then the containing value is compared.

getType() Returns the type of the val object. In case of a Boolean, Integer or String object, the constants BOOLEAN, INTEGER, STRING are used. All other types are represented by the NO_TYPE value.

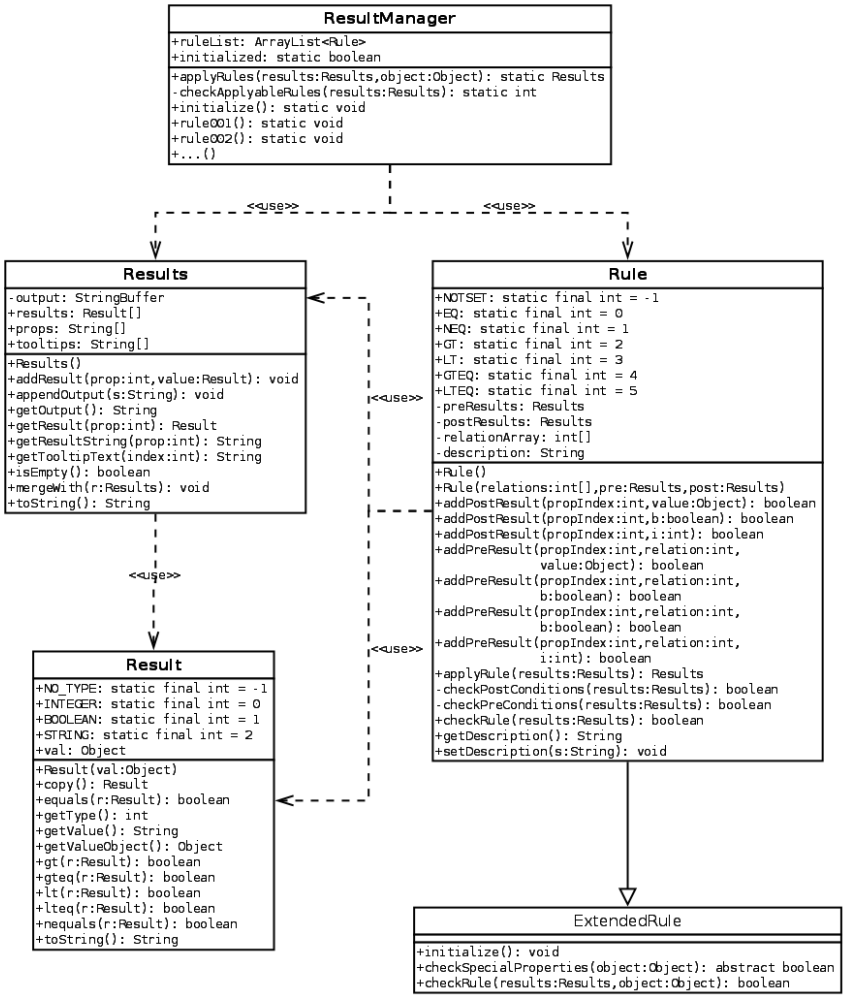


Figure 11: Results - rule system class diagram.

getValue() Returns a String representation of the contained val object.

getValueObject() Returns the val object itself for further evaluation.

gt(Result) Simulates the greater than relation and returns true if the value of the class is greater than the provided parameter.

lt(Result) Simulates the lower than relation and returns true if the value of the class is lower than the provided parameter.

gteq(Result) Simulates the greater-than or equal relation and returns true if the value of the class is equal or greater than the provided parameter.

lteq(Result) Simulates the lower-than or equal relation and returns true if the value of the class is equal or lower than the provided parameter.

notequals(Result) Simulates the not equal relation and returns true if the values of the class and the parameter are different.

The four methods are used by the rules to test if two results are in the defined relation. The methods gt(), lt(), gteq() and lteq() can only be used on Result objects with Integer values, other types will always return false. Comparing String or Boolean types wouldn't make much sense. If larger types than Integer are needed, this methods can be easily extended to larger or smaller types like Long or Short.

toString() This method calls toString on the val object inside the class, this is different from the getResult() method, because getResult() interprets the value of the val object and returns a string. In case of a Boolean value set to true the string " Y " is returned. The method toString() would return "true".

## 5.5 Description of class Results

This class also needed extensions, because of the new Analyzer system and the rule system. For example a rule needs a way to deliver its output to the protocol window, without exactly knowing the object (because of the separation of software layers). I treated the output of an analyzer and the output of a rule as part of the result set produced by the analyzer or the rule. So each Results object owns a StringBuffer object that collects all output. The initially implemented arrays for the Result objects are not touched and so the storage remains unchanged.

### Member variables

output A StringBuffer that collects all output attached by an analyzer or a rule. If Analyzer.setOutput(String s) is called, then the provided string is appended to the output StringBuffer.

results[] An array of Result objects, which stores the values of the properties.

props[] An array of String objects, that is initialized with the abbreviations of the net properties. The properties array is a static variable, so it is accessible without the need to instantiate a Results object. The method getResultString(index) provides access to the array from outside the object.

tooltips[] An array of String objects, that is initialized with short descriptions of the net properties. Like the props array the tooltips array is also static and therefore initialized for all Results objects. It is accessible from outside the object via getTooltipText(index).

**Methods**

Results() The constructor initializes the results array with null values.

addResult() Using the provided indexs the Result object is stored in the results array at the index position.

appendOutput() A String is appended to the output StringBuffer object including a newline character at the end.

getOutput() Returns the collected output as a String object.

getResult() Returns the Result object at the specified index position.

getResultString() Uses the props array to get the abbreviation of the net property. The method is static so the abbreviations are accessible without instantiating a Results object. Furthermore the array uses memory only once for all Results objects.

getTooltipText() The tooltips array is used to return the short description for a property and is also static for easy access.

isEmpty() This method returns true if there is no property set in the results array, if a result is different from "null" then false is returned. This method is used to prevent unnecessary checks.

mergeWith() The rules itself contain two Results objects which contain the preconditions and postconditions of the rule. If a rule is applied to a result set, the set has to be merged with the postconditions of the rule. This method copies the result values which are set in the postconditions object to the provided result set.

toString() The result abbreviations and the values of the results are returned as String object. The previous version of this method always returned all results and their values even if they are not set, this version prints only results which are set.

## 5.6 Description of class ResultManager

The ResultManager is responsible for initializing and administrating all rules. The class has several methods which are named rule001, rule002 etc. Each method initializes a single rule and adds this rule to the list.

### Member variables

ruleList Here all rules are stored. The list is declared as static, which prevents multiple initializations.

initialized The rules can be initialized by calling the constructor or calling ResultManager.initialize() directly. If the initialization is forgotten, the first access to ResultManager.applyRules(Result r) checks if the rules have already been initialized, if not, initialized is called. After initialization the value of initialized is set to "true".

### Methods

applyRules This method applies all rules if their corresponding preconditions are fulfilled by the provided result set. A detailed description about the way rules are applied follows this part.

checkApplyableRules Before rules are applied, a check for rules that can be applied is performed. The method returns the number of rules which can be applied.

initialize If the ResultManager is not initialized, this method calls all the rule001(), rule002() etc. methods to fill the rule list with initialized rules.

## 5.7 Description of class Rule

An instance of the class Rule holds all necessary information on the preconditions and postconditions as well as a textual description of the rule.

### Member variables

preResults The result set that contains the values of the results that are necessary to apply the rule.

postResults This result set contains the values of the results that will be set if the rule is applied.

relations[] An array of int values indicating the relation in which the preconditions have to be if the rule can be applied.

description The textual representation of the rule. Here more information on the rule can be stored.

**Methods**

Rule The constructor without parameters sets up an empty rule, the other one initializes the rule by predefined relations, conditions and post-conditions.

addPostResult The addPostResult methods use the index to place the Result object in the postResults object. The methods with the Boolean and int parameter provide a more comfortable definition, since the creation of Boolean and Integer objects is done by the method and therefore the user is freed from this task.

addPreResult Together with the Result object an int value representing the relationship in which the precondition is to the corresponding result from the analyzer.

applyRule The provided Results parameter is merged with the postResults result set. The description of the rule is appended to the output of the Results parameter.

checkPostConditions This private method returns true if all the postconditions are set in the provided Results object. The value of each single result is not checked, what matters is that the result is set, not its value. If one postcondition is not set then false is returned.

checkPreConditions Unlike checkPostConditions this method returns true only if all the results are set and if their value relates to the value of the parameter result in the way it is defined in the relations array. A detailed example follows.

checkRule Before a rule is applied, checkRule is called. This method returns true if all preconditions are fulfilled and not all postconditions are set.

getDescription Returns the description stored in the Rule object.

setDescription Sets the provided String object as description for the rule.

## 5.8 Description of class ExtendedRule

If additional information on a net are necessary to apply a rule, this class can be used to derive a special class. The derived class must implement the method checkSpecialProperties which takes additional analyses to decide if the rule can be applied. The ResultManager detects an ExtendedRule and calls checkSpecialProperties on the provided object parameter from ResultManager.applyRules(Results results, Object object).

A possible use for a class of the type ExtendedRule would be the Rule 11: MG & SC & each elementary circle contains a token => live & bounded & reversible. Where the part “each elementary circle contains a token” cannot be

covered by a net property and needs further checks. That's why the object is needed.

**Methods**

initialize This method is declared as abstract and the implementing class can place operations there which setup the class instance. The constructor of ExtendedRule makes a call to initialize.

checkSpecialProperties As mentioned above this method performs special checks on the provided object. Calls to external objects or analysis tools may be taken to acquire the result for the special property.

checkRule An overridden version of Rule.checkRule() , which uses an extra parameter that is passed to checkSpecialProperties. Here the preconditions and postconditions are checked as well as the call to checkSpecialProperties is made to ensure this rule can be applied.

# 6 How the rule system operates

This part describes the way the implemented rule system is accessed and used, how rules are defined and its limitations.

## 6.1 Access to the rule system

The access is as easy as possible. The only step that is to be taken is a call to Charlie.pn.ResultManager.applyRules(Results results). Since the Result Manager's methods are declared as static, no instance of ResultManager has to be created or held somewhere in the calling class. The ResultManager will then return the result set with the applied rules. The returned result set will contain the values for the newly set properties as well as the output generated by each rule. The output may be displayed in a dialog or printed to the console. No more steps need to be taken.

### Possible Problems

But there are two possible problems when applying rules

1. The sequence of the rules in the list may have an influence on the ability of rules to be applied.
2. There may be circles in the defined set of rules.

The sequence in which rules are inserted in the rule list is quite random and depends on the implementer. Even changes in the sequence are possible, if new rules are added in later extensions. So the problem is that an imaginary Rule A can only be applied if Rule B sets its postconditions so that all preconditions of Rule A are fulfilled. In this case a second run through the rules with the necessary checks has to be performed. Then Rule A can be applied and will be applied. The Question is when to stop checking for applicable rules. Lets adhere that the ResultManager has to check several times if a rule of the rule list can be applied, since the result set may change every run.

The first possible solution would be to track the changes in the result set and if no change has occurred, the loop can be left. Another solution also addresses the second problem and is therefore favored. The question for the ResultManager is: "Can at least one rule be applied?". If so, apply the rule(s), if not, leave and return the result set. To prevent endless loops, the check for ability to be applied has to consider both preconditions and postconditions. Normally it seems that the rule can be applied, if its preconditions are fulfilled. But once they are set, the rule can always be applied, no matter if it has already been applied or not. So a check for postconditions is performed too, if all postconditions are set, the rule has been applied or the results have been determined through another way. I want to emphasize that all of the postconditions have to be set, if only one is missing the method returns false. In case all postconditions are set, true is returned. This means a rule can only be applied, if checkPreConditions() returns true and checkPostConditions returns false.

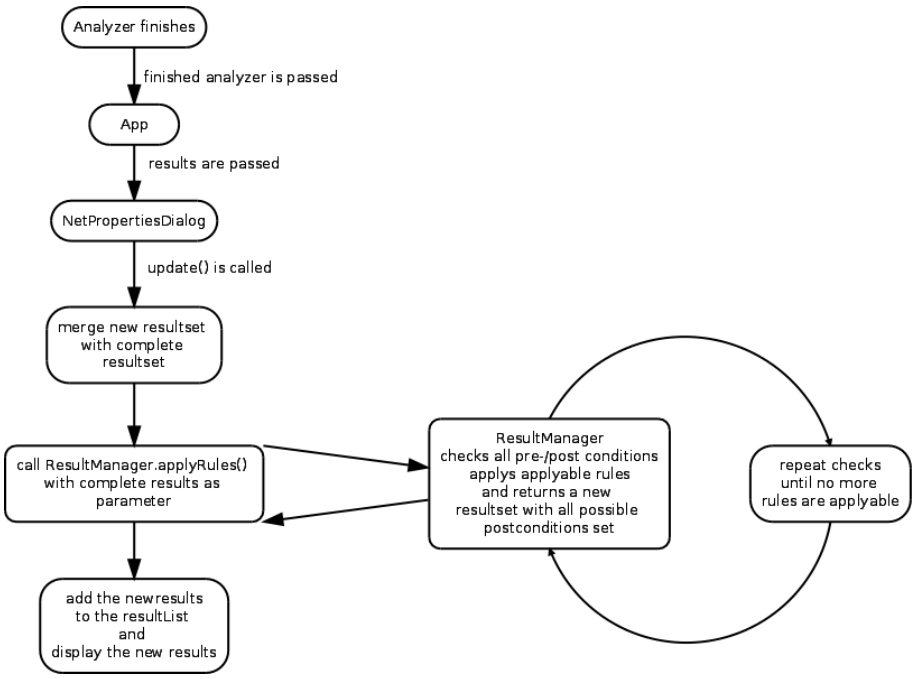


Figure 12: Result - rule -system.

The test in checkPostConditions does not consider the value of the postcondition in the provided Results object, what matters is that the corresponding result is different from null, which means it has been already set. So another possible source of problems is eliminated. The user is able to define the following rules A: b => !c and B: b => c. If c is not set the first run will apply rule A, then the second run rule B can be applied, because the value of c is different from the defined postcondition (c != !c), so rule B can be applied, then rule A is applicable and so on. That's why the check for the value of the postconditions is not performed. If a property is set to a value it should be impossible to determine a different value as valid to, e.g. a net cannot be considered to be bounded and to be unbounded at the same time. If so, then a rule or an analyzer seem to produce wrong results.

The following Figure 12 gives an insight how the results are handled and rules are applied.

After invariant computation finished, ResultManager checked for rules and presents a dialog which asks if the rule should be applied to the current result set.

Looking at the net properties dialog at the bottom of the frame, additional controls allow to switch through the results each analyzer returned. The last result set is always the complete set of results.

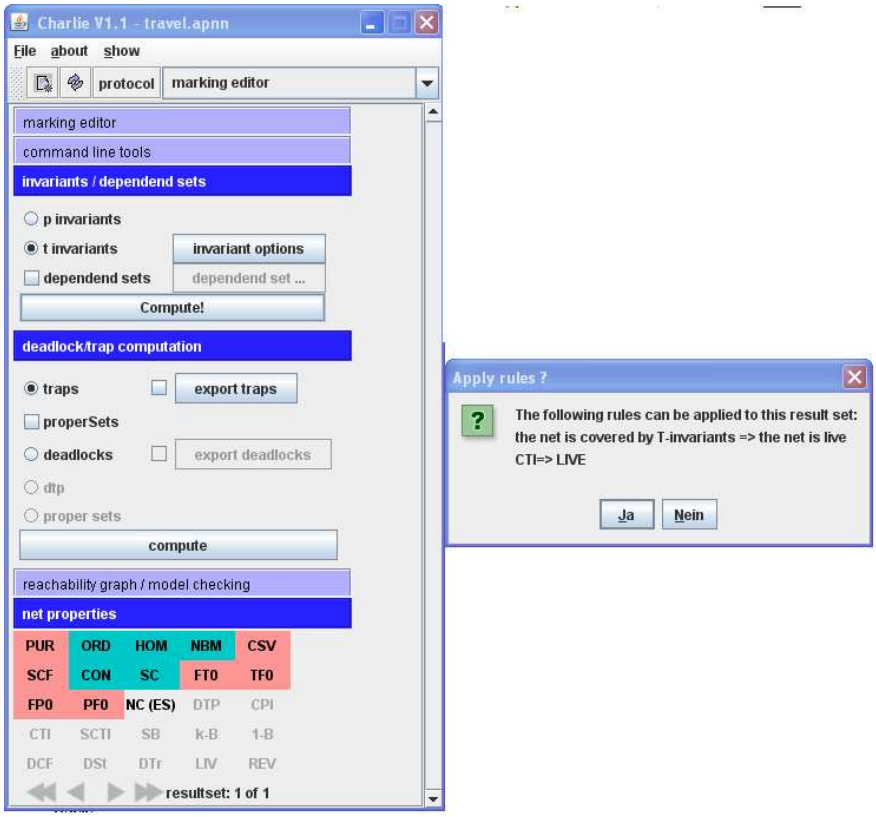


Figure 13: A rule is applied.

## 6.2 How rules are defined

We decided to place the definition of rules directly inside the source code. Rules are rather static and do no change over time, they are complemented from time to time as new properties are introduced or new analyzers are integrated. This means that changes occur very seldom and might therefore be applied direct to the source code. Defining rules using a text or XML file would mean a lot more management and because of the few changes this possibility is not chosen.

There are two ways rules can be defined. The first way is to use addPreCondition(), addPostCondition() and setDescription on a newly created Rule object to define a rule. The second way uses class inheritance to create a new class. The class ExtendedRule is an abstract basis for this new class. The derived class has to implement at least checkSpecialProperties() and optionally initialize() to perform a check. Initialize() should be used to set the preconditions and postconditions, and the description. The method checkSpecialProperties() on the other hand should be used to acquire data on the analyzed object that has not been acquired by an analyzer and is only needed for the evaluation of this rule. So if the special properties can be determined without longer lasting analyses the necessary actions should be placed here. When consequently used, a side effect of this is that the original net class can be kept smaller and simpler. Normally you would introduce a new method in the class e.g. PlaceTransitionNet to determine that 'each elementary circle contains at least one token' (Rule 11). This method is probably only needed together with the rule. So the rule system allows the separation of special methods and the analyzed object.

Here is an example for defining a rule in the first way using method from the class Rule.

```
private static void rule001(){
    // LIV & BND => SC
    Rule r = new Rule();
    r.addPreResult(Results.L, Rule.EQ, true);
    r.addPreResult(Results.B, Rule.EQ, true);
    r.addPostResult(Results.SC, true);
    r.setDescription("the net is live & bounded
        => the net is strongly connected\n LIV & BND => SC\n")
    ;
    ruleList.add(r);
}
```

As you can see the rule live & bounded => strongly connected is defined. After the new result class is created, addPreResult is used to set the preconditions for this rule. So the results with the index Results.L and Results.B are set to true. Because the check for applicability needs a relation, the second parameter is set to Rule.EQ, which means that the tested result set should have the same values set for the properties defined in the preconditions. Alternatively the other relation values (GT, LT, GTEQ,LTEQ,NEQ) can be used to define a relation. The relation always works the following way: precondition property <<relation>> tested property, e.g. the value for live in preconditions equals

the value for live in the tested result set. This is only relevant for integer based values and GT, LT, GTEQ and LTEQ relations, since EQ and NEQ work in both directions.

After adding the preconditions the postcondition is set. In this case the net is considered to be strongly connected, so the result at index Results.SC is set to true. After adding a description to the rule using setDescription(), the rule is added to the ruleList.

If another rule is to be added this way, a new method with the modifier static has to be implemented. The naming is up to the implementer, I chose a consecutive numbering, but this is not a must. The static modifier is necessary to be accessed by initialize or applyRules since they are declared static, too.

The following example illustrates the definition of an extended rule.

```
/** method rule012 from ResultManager.java*/

private static void rule012(){
  // MG & SC & each elem. circle
  // contains at least one token => LIVE & BND & REV
  // see rule011
  ruleList.add(new Rule012());
}
```

The method has to instantiate an object of class Rule012 and add it to the rule list. In this case there is nothing more to do in ResultManager.

The implementation of Rule012 concentrates on two methods: initialize and checkSpecialProperties. In initialize() the common precondition and postconditions are set, as well as the description. Inside checkSpecialProperties the correctness of the provided object is checked and then the analysis is performed. Depending on the analysis result true or false is returned. The check for the correct class type should be mandatory inside checkSpecialProperties, otherwise unexpected exceptions may be thrown by the virtual machine.

```
/** implementation of extended Rule */
package Charlie.pn.rules;
import Charlie.pn.*;
public class Rule012 extends ExtendedRule{
PlaceTransitionNet pn = null;

    public void initialize(){
        addPreResult(Results.NC, Rule.EQ, "MG");
        addPreResult(Results.SC, Rule.EQ, true);
        addPostResult(Results.L, true);        // live
        addPostResult(Results.B, true);        // bounded
        addPostResult(Results.R, true);        // reversible
        setDescription("net-class: marked graph and stronlgy
            connected and each elementary circle contains at
            least one token => live , bounded , reversible");
    }
```

```
    public boolean checkSpecialProperties(Object object){
        if (object instanceof PlaceTransitionNet){
            this.pn = (PlaceTransitionNet)object;
            /* do some operations on the provided object */
            return true;     // or false
        }else{
            return false;
    }
}
```

Both methods are called automatically, so the programmer is freed from additional management.

At this time the only point where ResultManager.applyRules() is called is NetPropertiesDialog.update(). The local PlaceTransitionNet object is passed as parameter. But what if the analysis of another object is needed? If there are ExtendedRules, which need e.g. an object of the type Charlie.rg.RGraph, to perform an analysis, the ResultManager can be called by the class that initiated the creation of the (in this case) reachability graph. When the corresponding analyzer returns from its analysis an the initiator is informed about that, the reachability graph object is returned (inside the option set) to the initiator. Then the initiator can call ResultManager.applyRules( ) with the result object of the analysis. The received results should be merged with the results in the option set.

## 6.3 Limitations

One limitation of the rule system is that all preconditions must fulfill, an "OR"-relation between two properties is not allowed. In case this relation is needed the OR-rule can be substituted by several rules covering all possible cases. The consequence of introducing "OR" relations between preconditions would lead to a complex system of interpreting parameters or parsing orders. This would have impact on the simplicity and usability of the rule system. Since changes to the system are rather seldom, the user would have to learn a complex system of producing rules, which the user probably forgets until the next time this knowledge is needed. So the simpler interface is favored.

# Part III
# Software architecture for analyzers

Charlie now covers a set of analyses, which can be performed on Petri nets. There are the following analyses:

- coverability graph / reachability graph construction
- invariant computation (place- and transition-invariants)
- dependent set(s) computation
- deadlock(s) computation
- trap(s) computation
- shortest path computation
- ctl and ltl model checker

These analyzers are arranged the way they were developed. In the former version, is not possible to access them in a consistent way.

So at first the existing analyzers are examined, then an idea for a new system is developed and last but not least the final system is introduced.

# 7 Structure of the existing analyzer system

At first lets take a look at the class structure of the existing analyzer system, see Figure 14.

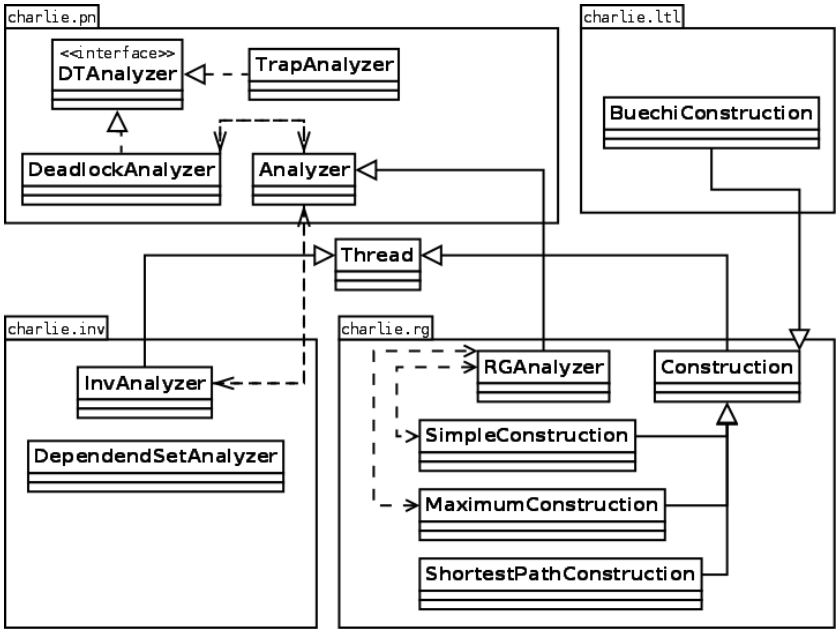


Figure 14: Package and class structure of analyzers.

As the Figure illustrates there is no consistent structure in the existing analyzer system. No clear inheritance structure or clearly defined method for accessing the analysis. Then there are several dependencies among the analyzers, so that they cannot work independently. Maintenance and extension is handicapped by this.

## 7.1 Examination of Charlie's existing analyzers

Before developing a new system for analyzers, the existing analyzers are examined for access, dependencies, task, determined properties and position in the software structure. This gives insight in how things work together and where changes can or should be applied .

| analyzer | Charlie.pn.Analyzer |
|---|---|
| task | initialize the PlaceTransitionNet object from a file<br>determine structural properties<br>evaluate invariants, start invariant computing using Charlie.inv.InvAnalyzer<br>provide methods to acquire the status of Petri net properties |
| construction | Analyzer() which creates an empty uninitialized object.<br>Analyzer(String filename, PlaceTransitionNet pn), the pn-object is initialized by the analyzer using the file name to read the file. Without pn.Analyzer the net is not initialized. |
| class inheritance | no inheritance |
| methods used for analysis | Analysis is done directly inside the second constructor. |
| analysis invoked by | Charlie.gui.GUI.loadNet(String name) constructs a RGAnalyzer, which is derived from Charlie.pn.Analyzer. Then the constructor initializes the PlaceTransitionNet object and performs the analysis. |
| determined properties / analysis results | Structural net properties are determined and deductions are drawn.<br>the following properties are determined: BND, Safe, SB, CSV, NBM, ORD, HOM, CON, PUR, FTB,TFB,PFB,FPB,NC, DTP |
| dependencies | Charlie.inv.InvAnalyzer and Charlie.pn.DeadlockAnalyzer are needed to determine net properties |

Table 1: class analysis Charlie.pn.Analyzer

Charlie.pn.Analyzer (table 1 ) looks a bit overloaded, the main task I see is determining structural properties. The initialization of the PlaceTransitionNet object could be done elsewhere as well as starting the invariant computation and evaluating the invariants. The result set is also stored and managed here, which could be a task for a single object. The dependencies to other analyzers could be removed by passing objects, since often Charlie.pn.Analyzer is used to access the stored pn (PlaceTransitionNet) object.

| analyzer | Charlie.pn.DeadlockAnalyzer |
|---|---|
| task | compute deadlocks for a Petri net |
| construction | DeadlockAnalyzer(PlaceTransitionNet pn) |
| class inheritance | derived from interface Charlie.pn.DTAnalyzer |
| methods used for analysis | Method compute() starts the analysis and returns an object Set which represents the computed Deadlocks. Within compute() the method deadlocks(...) calculates the deadlocks. |
| analysis invoked by | DeadlockTrapPanel sets the options using the state of its GUI-elements and starts the analysis by creating a DeadlockAnalyzer and calling compute(). |
| determined properties / analysis results | No Petri net properties are set inside this analyzer. The property DTP can be determined. This is done by calling Charlie.pn.Analyzer.hasDTP(this) from inside onCompute(), where this is the current instance of DeadlockAnalyzer. The result is evaluated in Charlie.pn.Analyzer. |
| dependencies | Charlie.pn.Analyzer, needed to perform DTP Analysis. |

Table 2: class analysis Charlie.pn.DeadlockAnalyzer

The use of Charlie.pn.Analyzer to compute the DTP property is circumstantial (table 2), and should be moved from Charlie.pn.Analyzer to DeadlockAnalyzer.

| analyzer | Charlie.pn.TrapAnalyzer |
|---|---|
| task | compute traps for a Petri net |
| construction | TrapAnalyzer(PlaceTransitionNet pn) |
| class inheritance | derived from interface Charlie.pn.DTAnalyzer |
| methods used for analysis | Method compute() starts the analysis and returns an object set which represents the computed traps. Within compute() the method traps(...) calculates the traps. |
| analysis invoked by | DeadlockTrapPanel sets the options using the state of its GUI-elements and starts the analysis by creating a TrapAnalyzer and calling compute(). |
| determined properties / analysis results | No Petri net properties are set inside this analyzer. |
| dependencies | no dependencies |

Table 3: class analysis Charlie.pn.TrapAnalyzer

A problematic issue is, that a TrapAnalyzer instance is casted to DeadlockAnalyzer in DeadlockTrapPanel.onCompute(), which works but is not a good style. Within DeadlockAnalyzer checkDTP() the deadlocks are calculated if not done before, so if calculating Traps with the setting calculate DTP to true, deadlocks are also calculated. The user initiates a second analysis without knowing.

| analyzer | Charlie.inv.InvAnalyzer |
|---|---|
| task | compute place and transition invariants for a Petri net |
| construction | *InvAnalyzer(Analyzer analyzer, String filename)*, which initializes the invariant analyzer by using Charlie.pn.Analyzer (to retrieve the PlaceTransitionNet object) and a file name pointing to a file which holds invariants. *InvAnalyzer(Analyzer analyzer,InvOptions io)* additional to the initialization, all options for computation are provided. |
| class inheritance | derived from java.lang.Thread |
| methods used for analysis | Method compute() starts the analysis and returns an object Set which represents the computed Traps. Within compute() the method traps(...) calculates the traps. |
| analysis invoked by | Charlie.gui.InvariantPanel sets the options using the state of its GUI-elements and starts the analysis by calling Charlie.pn.Analyzer.computeInvariants(...), then an instance of InvAnalyzer is returned. |
| determined properties / analysis results | No Petri net properties are set inside this analyzer. Charlie.pn.Analyzer determines CTI, CPI, B , SB, ECTI |
| dependencies | Charlie.pn.Analyzer - to start invariant evaluating |

Table 4: class analysis Charlie.inv.InvAnalyzer

The class InvAnalyzer (table 4) implements about 60 methods, which is quite a high number. Some of the methods are not used, instead variables are accessed directly (e.g. boolean InvAnalyzer.isMCSCenabled() is never used, instead InvAnalyzer.getOptions().mcscenabled is tested). Here some methods maybe removed without replacement. Some methods are for input-output purposes of invariants and may be moved into an invariant class, which in turn can be instantiated by providing a file name. The properties are determined in an external class and cannot be determined without this external class.

The constructor of DependentSetAnalyzer extracts the invariants and the

| analyzer | Charlie.inv.DependentSetAnalyzer |
|---|---|
| task | compute dependent sets on the basis of previously computed invariants |
| construction | *DependentSetAnalyzer(InvAnalyzer invanalyzer, DependentSetOptions mo)* both parameters provide objects for initialization. InvAnalyzer could be replaced by passing the required objects directly. |
| class inheritance | no class inheritance |
| methods used for analysis | Method compute() starts the analysis and calls computeStrongDTS() or computeSupportDTS() according to the state of the options. |
| analysis invoked by | Charlie.gui.DependentSetPanel creates a DependentSetAnalyzer and calls compute(). |
| determined properties / analysis results | No Petri net properties are set inside this analyzer. |
| dependencies | Charlie.pn.Analyzer and Charlie.inv.InvAnalyzer to access invariants and Petri net |

Table 5: class analysis Charlie.inv.DependentSetAnalyzer

analyzer objects from InvAnalyzer. The analyzer object is only needed to access the stored net, so instead of extracting the PlaceTransitionNet object from Charlie.pn.Analyzer, the net object could be passed directly to the analyzer, as well as the invariant analyzer object which is only needed to access the object containing the invariants and to write the computed dependent sets to a file.

| analyzer | Charlie.rg.RGAnalyzer |
|---|---|
| task | compute coverability or reachability graph for a Petri net |
| construction | *RGAnalyzer(String filename, PlaceTransitionNet pn)*<br>Since this class inherits from Charlie.pn.Analyzer, the first constructor uses the constructor from Analyzer and therefore initializes the pn-object.<br>*RGAnalyzer()*<br>An empty RGAnalyzer object is created.<br>*RGAnalyzer(RGraph rg)*<br>This constructor sets the value for the internal RGraph object. |
| class inheritance | derived from Charlie.pn.Analyzer |
| methods used for analysis | The method constructRG() chooses the right Construction class, then this class creates the coverability-/reachability graph using its constructMax(...) or constructSimple(...) methods. |
| analysis invoked by | Class Charlie.gui.Gui invokes RGAnalyzer.constructRG(). The method constructRG() collects the options from global static variables and creates a new ConstructionOptions object, then class Charlie.rg.Construction is used to return an instance of SimpleConstruction or MaximumConstruction, depending on the state of the provided options. If the construction finishes, RGAnalyzer setRG(...) is called and the RGraph object is evaluated. |
| determined properties / analysis results | Inside of this analyzer the following properties are set: LIV, REV, DCF, Safe, DST, BND |
| dependencies | Charlie.pn.Analyzer because of class inheritance |

Table 6: class analysis Charlie.rg.RGAnalyzer

Within RGAnalyzer, empty objects are also passed to the constructor. Those objects are initialized inside the class, this indirect initialization is transparent to the user because the actual task the analyzer is associated with is producing a reachability graph and setting net properties. The initialization of Petri net objects or passing empty objects which are initialized inside does not follow a natural understanding. A better way of producing the result object would be to call the construct(...) method using the correct options and receive the newly created result object as well as the net properties that apply for this analysis.

## 7.2 Problematic issues

The mix of responsibilities makes the analyzer objects quite complex and constricts the understanding. A good example for this is Charlie.pn.Analyzer which covers several topics: input-/output, initialization of Petri nets, evaluation of invariants and initiation of deadlock-/trap evaluation. The following issues will be resolved by the new analyzer structure:

- There is no clear way for GUI components to access analyzers, there are different ways to start an analysis and often knowledge of more than one analyzer is necessary.
- The whole processes looks quite complicated and unordered and is therefore hard to understand. A new system should improve these aspects significantly.
- Then GUI objects have to care for updating status information, and listening for the end of analyses. Inside of the analyzers these listeners need additional management.
- The class Charlie.pn.analyzer calls InvAnalyzer.computeInvariants() and evaluates the invariants too. This functionality will be moved towards the InvariantAnalyzer.
- Also the method hasDTP(...) in Charlie.pn.Analyzer, which has the parameter DeadlockAnalyzer calls DeadlockAnalyzer.checkDTP() to evaluate the result.
- Analyzers are passed as parameters instead of passing result objects.
- The class Analyzer is misused to access the PlaceTransitionNet inside, instead of passing the object.
- Also class RGAnalyzer is passed, instead of passing the RGraph object.
- There are many casts from Analyzer to RGAnalyzer.
- DeadlockTrapAnalyzer needs Charlie.pn.Analyzer needed to be constructed and to to access Petri net.

Objects that are produced during an analysis are hidden inside the analyzers. The passing of analyzers as parameters, masks the real operations which is intended by this, e.g. passing pn.analyzer is only needed to access the Petri net inside analyzer and to call a method from analyzer which could be placed in the calling analyzer.

## 7.3 Idea of the new analyzer system

As we have seen on the previous pages, the existing structure for analyzers shows the way Charlie was developed. Starting with a single analysis more and more features were built around the program. This lead to the existing structure. Now it was time to think about a new way of handling and creating analyzers for Charlie. Together with the new user interface the whole analyzer system was revised.

The task was to create "A concept for managing and controlling analysis threads". What does this mean:

1. There is more than one analysis thread.
2. The threads work independently.
3. Parallelism is realized inside the analyzers, which means that existing algorithms are not paralleled, but left as they are.
4. Controlling means: pausing, resuming, canceling threads.
5. Managing means: get information about the thread and its analyzer, create analyzing threads, collect their results and return the results to the presenting user interface.

So I developed the following requirements which are the direct product of the analysis of the existing structure and the task.

1. Creating (implementing) an algorithm as Charlie analyzer must be very simple.
2. The special analyzer (like InvAnalyzer) itself should be freed from all tasks but realizing the analysis algorithm.
3. The programmer of the analyzer doesn't need to care about creating an analyzer thread and the tasks that are connected with starting a thread.
4. The analyzers must get a generic representation in the user interface, which informs the user about the status of the analyzer and allows him to interact with the analyzer (pause/resume/cancel/status).
5. Since more than one analyzer can be active, the output can't be accessed directly by the analyzer, so appending the output to the protocol has to be managed.
6. Each analyzer is independent from other analyzers. If one analyzer depends on a result created by another analyzer the managing class has to ensure that the analyzer gets his result.
7. Initialization of analyzers is done by a consistent way.

8. Direct access to analyzers by the classes of the user interface or the logic is not possible anymore, the managing class cares for selecting the correct analyzer for the needed analysis.

9. Since a textual interface for Charlie is another task, the analyzer system must be operable independently from any graphical user elements.

These requirements are created because the implementation and integration of an additional analysis should be as easy as possible. The implementer of the analyzer class has to care for so much things, like memory consumption, efficiency, correct algorithms etc. that caring for "lower" tasks distracts attention.. So the new structure should define a clear way to integrate an analyzer into Charlie, to access it and use its results.

My idea of the process is shown in the following diagram.

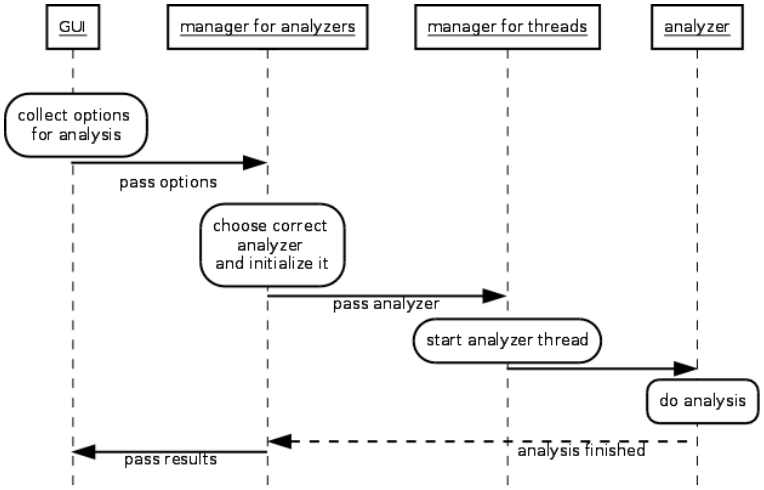


Figure 15: Idea of analyzer use.

The Figure 15 already shows the main actors for this structure. If this structure will be realized at least the following questions have to be answered:

- How stores the analyzer Manager the special analyzers?
- How can the manager choose the correct analyzer?
- Is it possible to initialize all analyzers with the same procedure, even if the options are different?
- How is the analyzer manager informed about the end of an analysis and how is the result passed to the user interface?

## 7.4 Introduction to the new analyzer system

The realized system takes care of those questions. I will split the introduction of the system into three parts:

1. The class that acts as basis for all analyzers.
2. How options are passed and stored.
3. Realized Analyzers.
4. The analyzer manager and how analyses are invoked from the GUI and how the results are returned.
5. How does the thread manager handle the analyzer threads.
6. A summary what has to be considered when implementing an analyzer.

In difference to the existing system all analyzers are located in separate packages below Charlie.analyzer.

The sequence diagram (Figure 16) shows the realized way an analysis is invoked, from pressing the compute button till the return of the results. The following pages will introduce the involved classes in detail.

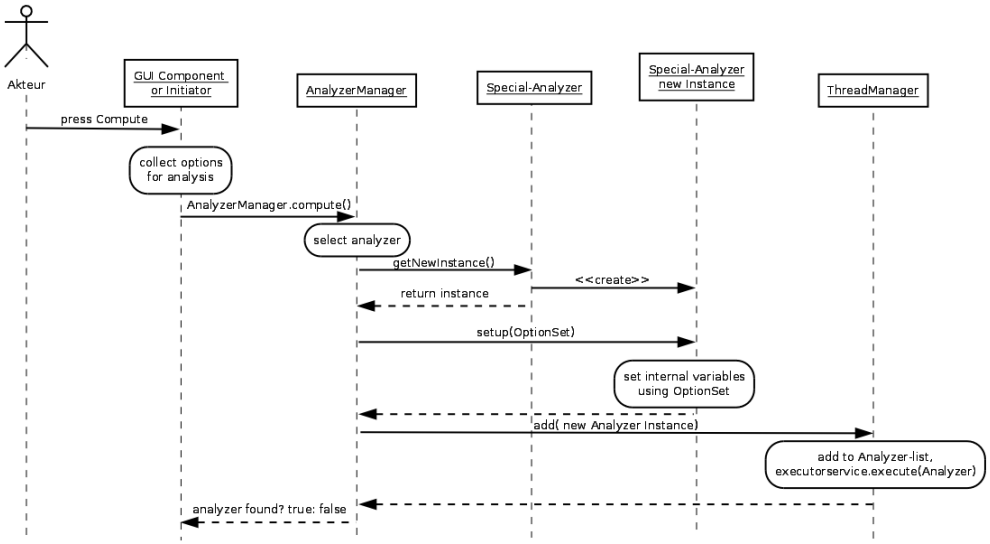


Figure 16: Sequence diagram showing use of analyzers.

## 7.5 Charlie.analyzer.Analyzer - the basis for all analyzers

The class Charlie.analyzer.Analyzer is essential for the simple use of all analyzers, because it handles the most issues for all derived classes. The following tasks are performed by Analyzer:

- the interaction with the ThreadPanel, which shows information on the running analyzer and allows to pause/resume or cancel the analyzer.
- setting and storing the start time and end time and returning it to the ThreadPanel or the output
- provide a check if the analysis should be interrupted because the user pressed cancel on the ThreadPanel
- provide a setup method that initializes all needed variables and prepares all variables for the (short) final setup in the derived analyzer
- provide methods to access results objects or options and set and retrieve results.
- since Analyzer implements the interface Runnable, the run() method is implemented here too and makes the calls to the method that starts the analysis.

Analyzer is declared as abstract class, which means that classes who are derived by Analyzer must implement certain methods which are also declared as abstract. This ensures that all needed methods (like analyze() or evaluate()) are implemented by the programmer. Most methods don't need any parameter, so the implementer is forced to use the derived option set for this analyzer as storage for all settings and objects needed to perform the analysis. This ensures a consistent way of using all analyzers. The following methods are declared as abstract: cleanup(), initializeInfoStrings(), analyze(), evaluate(), getNewInstance(OptionSet) their tasks are explained later.

**Variables of the analyzer base class**

panel — If a graphical user interface is used, the thread manager will create a ThreadPanel for each started analyzer and ThreadPanel places a reference to itself in this variable. The analyzer must know the panel, because the user is informed about the status of the analyzer by using colors in the ThreadPanel. Only the analyzer can determine if the analysis is finished, because the derived analyzer returns from its analyze() method. So the status get automatically updated if the analyzer finishes.

lock — An object that is used to synchronize access to critical parts of the code. Because the status of the analyzer can be changed by user interaction as well as by the analyzer itself, the areas which set the status or get the status are protected from interruption by using this lock object.

options — This is the reference to the OptionSet object which holds all information needed for the analysis. This object is explained later in details. The initialized set of options is passed by the setup(...)

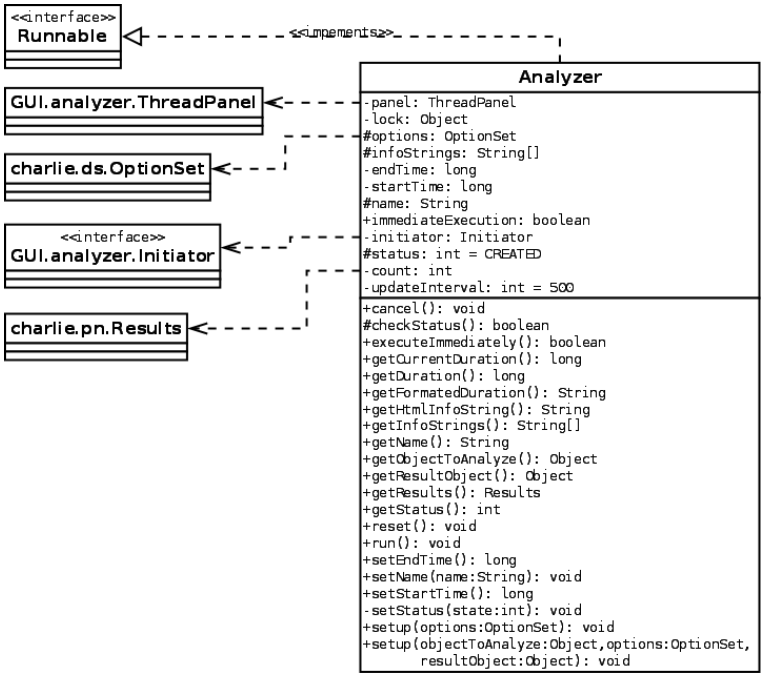


Figure 17: Class structure Charlie.analyzer. Analyzer with classes it depends on.

method. Initialization is done either by using user interface elements or the parameters provided by the textual interface.

infoStrings This array of Strings stores mostly status information on the analyzer and is initialized in the derived analyzers. The implementer is free to choose to display all interesting information his analyzer can provide.

endTime/startTime These are the times the analyzer was started and finished his work. They are initialized and set automatically by the base class, derived classed do not need to care about them.

name A string which represents the name of the analyzer, the implementer should set this variable when the constructor of his analyzer is called. The name for example is used in the ThreadPanel to identify the analyzer.

immediateExecution During the implementation it turned out that certain analyzers may require only a short time to perform their analysis, like the StructuralAnalyzer. If the programmer chooses to set immediateExecution to true, then the analyzer is executed directly without being represented by a ThreadPanel or waiting for other analyzers to finish their work. The default value is false, so if not set the analyzer will use the thread manager.

initiator The only method defined by the initiator interface is analyzerHasFinished(Analyzer finished), which must be implemented by the class who is interested in getting the result after the analysis finishes. In most cases the class which initiated the analysis will implement this interface. The reference to this class is passed with the option set and initialized by the setup(...) method. The initiator is not informed directly by the analyzer, but by the analyzer manager after some checks.

status This integer value represents the current state the analyzer is in. The following final static variables represent the allowed states: CREATED - if the analyzer is reset or instantiated, REGISTERED - if the analyzer is registered within the analyzer manager, SETUP - if the setup was performed, WAITING - if the analyzer waits for execution, PAUSED - if the analyzer is paused, ACTIVE - the analyzer performs its analysis, FINISHED - the analyzer could finish its analysis and returned safely from the analyze() method, CANCELED - if the user interrupted the analysis.

count Since updates to the associated ThreadPanel are done if checkStatus is called, this internal variable counts how often checkStatus is called. In cooperation with updateInterval the update is done every time count equals updateInterval.

updateInterval The programmer may set an appropriate value for his analyzer, because each analyzer might perform more or less loop runs during his analysis. Because the update to the thread panel should not be done every time checkStatus is called (otherwise the computer is more stressed with updating the thread panel than with the real analysis) the updateInterval variable sets the distance between updates.

The base class only holds 11 variables and 6 of them are declared as private and are therefore invisible to derived classes. Some of the needed complexity is hidden inside the OptionSet object, which will be explained later.

**methods of the analyzer base class** I want to start with the abstract methods and illustrate their meaning for the derived analyzers, which have to implement them.

analyze() This method should do all necessary operations to perform the analysis. Since the run method is implemented by the base class, run() makes a call to analyze() to start the analysis after some checks. This method should never be called directly, therefore it is declared as protected. So only Analyzer or the derived class are able to call it.

evaluate() The analysis and the evaluation of results should be separated, because analysis methods tend to become long and this enables the programmer to place the conclusions, the produced result allows, here. The method is called by run() too.

cleanup() If the analyzer is canceled by the user, this method is called by getStatus(), so the analyzer has the possibility to clean up objects that consume memory but are not usable.

getNewInstance() This method is the second stage in the process analyzers are chosen by the analyzer manager. The analyzer manager decides on the basis of the pair of object that is to be analyzed and the object that is the result of the analysis. But if the decision cannot be made because values of variables of the option set decide which special analyzer has to be chosen, then this method must implement this choice. An example for the use of this is RGAnalyzer, which implements the choice between SimpleConstruction and MaximumConstruction by the value of the variable fireRule inside ConstructionOptions.

initializeInfoStrings() The ThreadPanel offers a button, which shows statistics about the current analyzer. These statistics are displayed on the basis of this String array. The array even indices are initialized with the description of a statistic value and the following odd index is filled with the value itself. The implementation should update the values every time this method is called.

Now the non-abstract methods are listed, ordered by their importance.

run() After checking the correct status of the analyzer, the status is set to an active, the start time is set and analyze() is called. Then if the analysis finished the end time is set, the status is updated and the analyzer informs the analyzer manager about the successful end of the analysis.

register() This static method needs to be overridden by the derived analyzer. Within this method the analyzer should perform all actions necessary to register itself in the analyzer manager. Because the method is static no additional instance of the analyzer has to be created.

checkStatus() The return value (true/false) tells the derived analyzer, who calls this method, if he can continue his work (in case true is returned) or if he should stop working (in case false is returned). Next to this, the ThreadPanel is updated. An imported part is the realization of the pause feature of the analyzer. Normally the programmer would check a variable (e.g. via getStatus()) at a certain time and decide if the analyzer has to wait until the status changes so that the work can be continued. This behavior is completely implemented inside this method. So calling this method inside the analyzer automatically pauses the work until the user decides to press the resume button in the ThreadPanel. The programmer doesn't need to care about the pause/resume feature in his analyzer. What is more important is to place the call to checkStatus() at a good position inside the algorithm, but this can be done best by the programmer since he is probably the best source for knowledge about the algorithm.

pause() Sets the status of the analyzer to PAUSED. The next call to checkStatus will put the analyzer to sleep.

resume() Sets the status of the analyzer to ACTIVE, the lock variable is notified so that the analyzer continues his work.

cancel() The status is set to CANCELED and in case the analyzer was paused the lock is notified so that the analyzer can abort his work in a defined way.

setStatus() The method enables setting the status of the analyzer, with informing the ThreadPanel about the changed state (if necessary).

The original analyzers had to care for instantiating and managing timers to retrieve the duration of an analysis, now this is automatically done by the base class. If a derived analyzer needs to know the time it is run, the methods getDuration(), getCurrentDuration() and getFormatedDuration() will tell the value. The methods setStartTime() and setEndTime() are declared as private, so only the base class has access to this methods.So full control about determining the

time lies inside the base class, this prevents possible mistakes by unintended or wrong calls to this methods.

There are several getter and setter methods which need no further explanation, since their name contains enough information about what objects are set or got by this method. These methods can be seen in the Figure 17.

## 7.6 How options are passed - the class OptionSet

The existing analyzer have all been adapted to the new system. The adaption was not always simple, as the analyzer were not prepared for the system. So all calls and operations an global static variables had to be removed, since these operations can only be coordinated safely with big efforts. In difference to the existing system now each analyzer can be instantiated more than once and if a static option is changed the effects on running analyzers can't be predicted.Next to this, the way of appending output to the protocol had to be changed. The immediate output via Charlie.pn.Out is no longer an option, the output of analyzers which are active would be mixed. Therefore all existing implementations had to be searched for those commands.

Also the determination of net properties with the use of the calculated result objects, had to be moved from Charlie.pn.Analyzer to the analyzers which produced the result object. Also almost each analyzer implemented methods for storing the results into files as well as loading results from files, so there are many redundant lines of code. So the import and export of results is placed outside the analyzer, what is left inside is the call to the export functions.

The investigation of analyzers shows the need for two additional classes :

- a class that stores all options, results and (textual-)output and
- a class that represents the result object and can help to identify the analyzer.

Figure 18 shows the needed objects for InvariantAnalyzer.

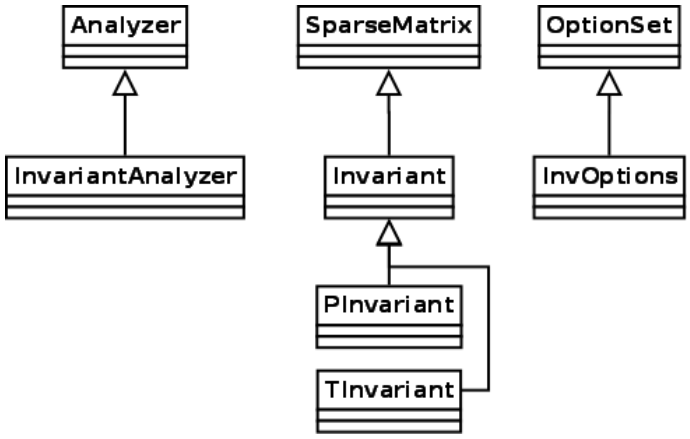


Figure 18: Set of objects needed for an analyzer exemplified with InvariantAnalyzer.

The classes PInvariant and TInvariant are used to identify the analyzer and store the computed invariants. InvOptions store all special options needed to compute the invariants. For example the value of the member variable InvOptions.transitions determines what type of invariant is to be computed. The original analyzer only knew the type SparseMatrix, the type of the invariant was not stored inside the SparseMatrix object.

So each analyzer knows its own result object and the pairs are:

- InvariantAnalyzer - Invariant, PInvariant, TInvariant
- DeadlockAnalyzer - Deadlock
- TrapAnalyzer - Trap
- RGAnalyzer / SimpleConstruction / MaximumConstruction - RGraph
- StructuralAnalyzer - PlaceTransitionNet

## 7.7 Example invariant options

Figure 16 shows the principle of passing options to the analyzer. The invariant dialog in Figure 19 displays some checkboxes and radio buttons. The state of these components is captured when the compute button is pressed and then analysis is started by passing the options to the AnalyzerManager.

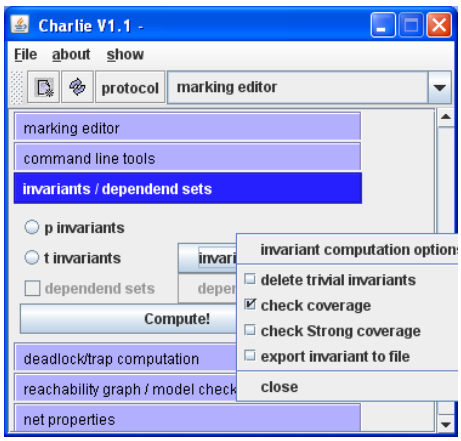


Figure 19: Invariant options dialog.

To stay with the previous example of invariant computation, the following listing of InvariantComputationDialog shows what has to be done to create the OptionSet .

```
1  JButton compute =  new JButton(new AbstractAction ("Compute!")
       {
2      public void actionPerformed(ActionEvent e){
3          if (pn != null){
4              InvOptions io = new InvOptions();
5              Invariant invariant = null;
6              if (jRadioButtonTInvariants.isSelected() && !
                   jRadioButtonPInvariants.isSelected()){
7                  io.transitions = true;
8                  invariant = new TInvariant();
9              }else{
10                 io.transitions = false;
11                 invariant = new PInvariant();
12             }
13             io.coverage = checkCoverage.isSelected();
14             io.extendedCoverage = checkStrongCoverage.
                   isSelected();
15             io.deleteTrivial= deleteTrivialInvariants.
                   isSelected();
16             io.exportFile = invExportFileItem.isSelected() ?
                   invExportFileItem.getAbsoluteFileName():null;
17             ...
18
19             compute(pn, invariant, io);
20         }
21     }
22 });
```

It can be seen that the constructor of InvOptions needs no parameters. All

member variables are public so there is no need to implement simple get/set methods which only pass or return the value. Each value is represented by an element in the user interface. Mostly checkboxes, comboboxes or spinners are used to determine the values needed for the option set. This ensures simple access and storage of the options.

After setting the appropriate options, the dialog's compute(...) method is called , which thereafter calls the AnalyzerManager.compute(...) method.

Each option set should use public member variables for their options, because otherwise a lot of getter and setter methods would have to be implemented.

## 7.8 The OptionSet class in details

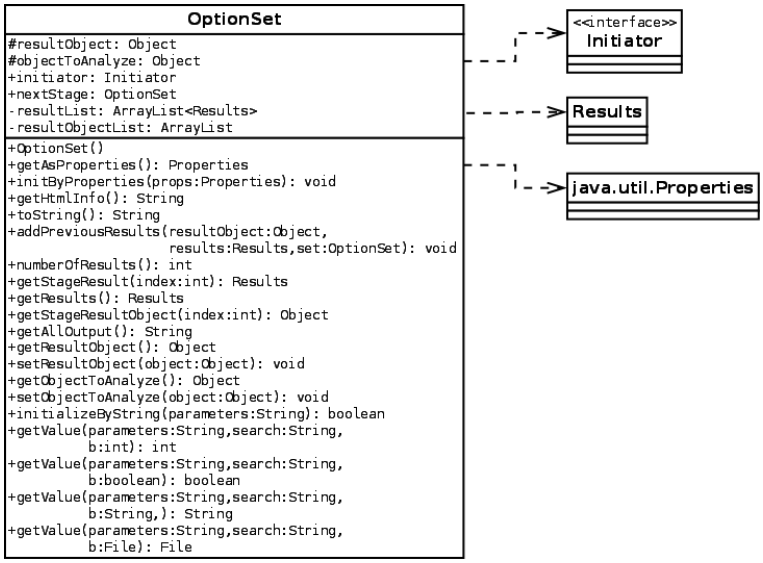


Figure 20: Class diagram Charlie.analyzer.OptionSet.

### Member variables

resultObject Here the analyzer stores the result of its analysis.

objectToAnalyze Here the object that is to be examined by the analyzer is stored. The class that collects all options must also own a reference to this object to pass it to the analyzer.

initiator This variable is needed to return to the class that initiated the analysis. The analyzer manager checks if this variable is set and if so the method analyzerHasFinished() is called on initiator. So the results are passed to the logic or GUI and can be presented to the user.

nextStage This variable is used to realize the concept of multi-stage-analyses. Here the option set for the next analysis is stored. The analyzer manager uses this variable to setup the next analysis.

setList If there is more than one stage,the option sets with the results of each stage are stored here.

**Methods**

getAsProperties() The current state of the option set is returned as java.util.Properties. This method is declared as abstract, so the derived classes have to implement this method. A Properties object is returned, containing the state of the option set.

initByProperties() By using this method an option set can be initialized by passing a Properties object. This method also has to be implemented by the derived option set.

getHtmlInfo() The current state of the option set is returned as HTML table.

toString() The current state is returned as string.

addPreviousResults() If a previous analysis finishes, the analyzer manager puts the results of this analysis into the option set of the next stage using this method.

numberOfResults() If the results of an analysis are evaluated, this method tells the number of previous stages.

initializeByString() The textual interface will need this option, since the parameters for the analyzers are passed as string. The evaluation has to be done in each derived analyzer.

getValue() Needed by initializeByString(), these methods extract the value for an option from the parameter string and initialize the corresponding variable with the return value. If no value can be queried the passed standard value is used. Currently there are 4 getValue() methods, with different types for standard values and return values. The methods are declared as static, so they can be accessed by the textual interface to extract additional parameters not needed by option sets.

The remaining get/set methods are self-explanatory.

## 7.9 Multi-stage analyses

The examination of the invariant analyzer and the dependent-set analyzer showed that the dependent sets cannot be computed without previously computed invariants. So the dependent-set analyzer needs the results of the invariant analyzer. This has to be reflected by the option set. So I introduced an additional

variable in the option set called nextStage. The option set for the second stage is placed in this variable. If the analyzer manager is informed about the end of the invariant analysis, it checks if the nextStage variable points to an option set. If so the manager collects the results from all previous stages, puts them into the option set of the next stage and starts the next analysis. So the results are carried along until the last analysis finishes. If the next stage did not set the objectToAnalyze, the result of the previous analysis is set as objectToAnalyze. So the computed invariants are passed to the dependent-set analyzer.

In terms of the textual interface this is extremely useful. The parameters typed in the console describe the options for an analysis, if more than one analysis is planned and all the analyses should be executed one after another, the nextStage variable can be used to realize this. The next analyze not is invoked till the previous analysis has finished.

## 7.10 Implemented option sets

The Figure 21shows all implemented option sets and their member variables.

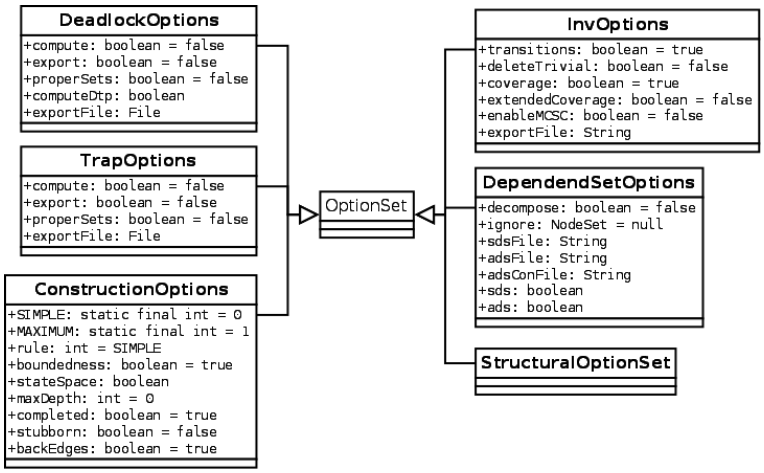


Figure 21: Hierarchy of option sets.

# 8 Realized analyzers

The previous class structure (see Figure 14) is replaced by the new structure in Figure 22.

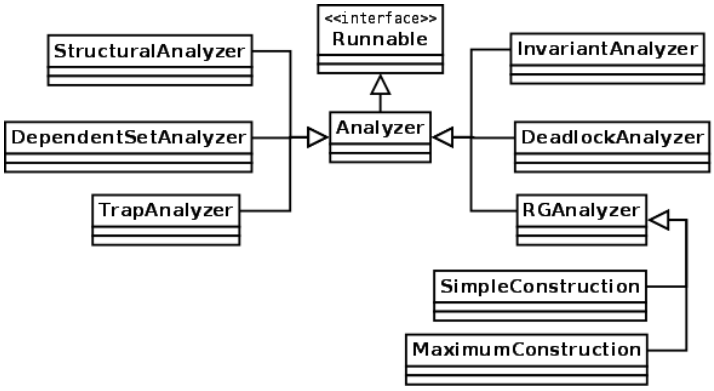


Figure 22: New class structure for analyzers.

The new analyzer structure is also reflected in a new package structure, which places each analyzer in a separate package. If additional objects are necessary to perform an analysis these can be placed there too. So the usual contents of an analyzer package is the analyzer class, the option set, one or more classes that represent the objects that are products of the analyses.

You will notice that analyzers are missing, compared to Figure 14. The Buechi construction is not used in my version of Charlie and shortest path construction is currently revised in an other work [AF2009] , so they are not analyzed and not integrated in the new structure.

The designed structure (Figure 23) reflects the clear and easy to understand system.

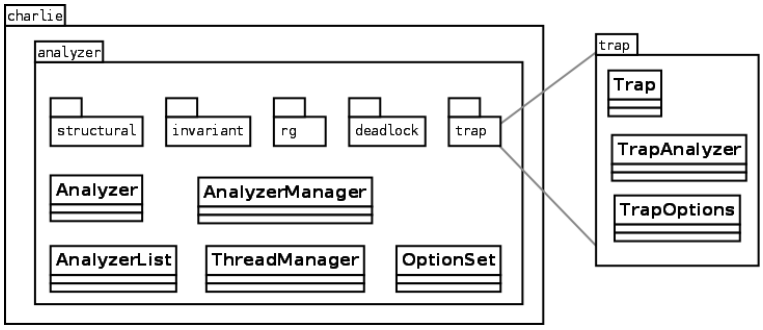


Figure 23: Analyzer package structure with example.

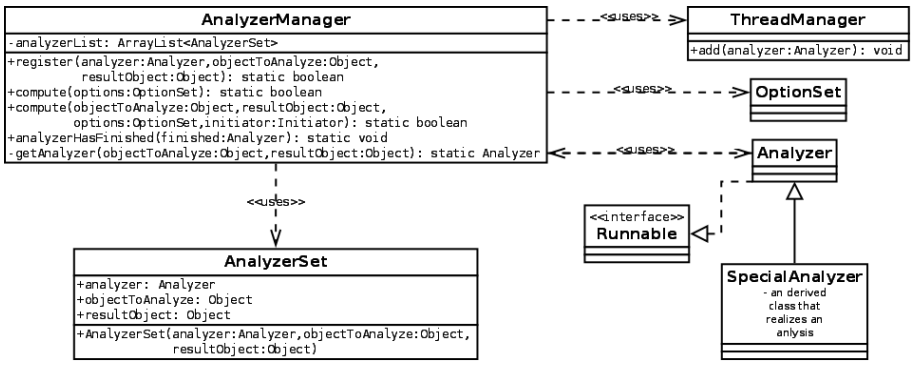


Figure 24: Class diagram analyzer manager.

# 9 AnalyzerManager - how analyses are invoked

Now its time to describe the way the analyzer manager works. Because direct dependencies between analyzers and the user interface are undesired, the analyzer manager takes the part of choosing the correct analyzer and starting the analysis. The analyzer manager holds all possible instances of analyzers and identifies them by the objects that are to be analyzed and the objects that are the results of the analysis (Figure24).

## 9.1 How an analyzer is identified

Before the analyzer manager is able to choose the correct analyzer, the analyzer must be registered in the analyzer manager. This can be done within the logic that collects the options for the analysis or at a central place in the application. I chose to register the analyzers in the corresponding user interface. The system uses the types of the object that is analyzed and the type of the object that is produced by the result to match the analyzer. Therefore the analyzer implements the static method Analyzer.register() in which an analyzer instance along with the analyzed object and the produced object is created and AnalyzerManager.register(...) is called. The following listing of InvariantAnalyzer shows the necessary steps.

```
1 public static boolean register(){
2     InvariantAnalyzer ia = new InvariantAnalyzer();
3     PlaceTransitionNet pn = new PlaceTransitionNet();
4     boolean b1 = AnalyzerManager.register(ia, pn, new
          PInvariant());
5     boolean b2 = AnalyzerManager.register(ia, pn, new
          TInvariant());
6     if (b1 && b2) return true;
7     else return false;
8 }
```

It can be seen that each analyzer is allowed to register itself for more than one result object and even more than one object to analyze. Here the invariant analyzer registers itself for performing analyses on PlaceTransitionNets to compute PInvariants and TInvariants.

If an analyzer with the same combination of objects tries to register, the AnalyzerManager refuses this and returns false. For registration it is not necessary to have initialized objects. The objects should rather be empty and should not consume large amounts of memory immediately after construction. After an analyzer is registered, it is found by AnalyzerManager.getAnalyzer(...). This method looks for an analyzer in the analyzer list, that matches the types of classes. Type in this case means the full qualified class name of the object, which can be easily determined by calling getClass().getName() on the object instance. Then the class names for objectToAnalyze and resultObject in the stored AnalyzerSet are compared with the names of the provided parameters. If both names are equal the adequate analyzer is returned.

## 9.2 How an analysis is invoked

If the option set is created and filled with values the class which created the option set just has to call Analyzer.compute(...) and the analysis will be performed. There is nothing more to do to invoke an analysis. It is possible to put several analyzers in the waiting queue of the thread manager, with each analyzer equipped with an option set with different values.

The compute method only chooses the correct analyzer with the help of getAnalyzer(...) and if an analyzer could be found a new instance is created by calling a.getInstance(...). Since the option set is passed as parameter the method can call setup on the new analyzer instance. When the new analyzer is returned an additional check is performed. If the analyzer is not initialized then the setup method is called on the new analyzer. After the setup the analyzer is passed to the ThreadManager, which cares for the correct execution of the analysis.

```
1  public static synchronized boolean compute (Object
       objectToAnalyze, Object resultObject, OptionSet options,
       Initiator initiator){
2      Analyzer a = getAnalyzer(objectToAnalyze, resultObject);
3      if (a != null){
4          Analyzer newAnalyzer = a.getNewInstance(options);
5              if(newAnalyzer.getStatus()!=Analyzer.SETUP)
                                  newAnalyzer.setup(objectToAnalyze,
                   options, resultObject);
6          if (initiator != null)
7                  options.initiator = initiator;
8          ThreadManager.add(newAnalyzer);
9          return true;
10     }
11     // an error occurred no analyzer could be found for the
           combination of objectToAnalyze and resultObject
```

```
12      return false;
13  }
```

The described system allows to associate only one analyzer with a pair of objects. It's possible that the correct analyzer can only be chosen by knowing the value of an option inside the option set. This needs knowledge of the special option set and the connection of the special analyzer to the option value. The analyzer manager would grow in size and get quite complicated if this logic is implemented here. At this point the method Analyzer.getInstance(OptionSet set) can help. Since the initialized option set is passed to getInstance the analyzer can choose a special analyzer by evaluating certain options. The RGAnalyzer with the derived analyzers SimpleConstruction and MaximumConstruction is an example for this. The value for the fire rule inside ConstructionOptions decides which analyzer is returned by getInstance. So both the combination of objects and values inside the option set can decide about the correct analyzer.

# 10 Thread Manager - how analysis threads are handled

During the development I tried to create my own thread manager which would implement the handling of threads. Some research showed that Java offered an excellent class that exactly realizes the needed functionality. ThreadPoolExecutor is this class. The java documentation names the advantages of a thread pool:

> "Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks." (Java 2SE Documentation)

Around the executor service the thread manager creates a frame which allows the user to see which analyzer is active or waiting or finished (Figure 25). The interaction between the ThreadPanel and the Analyzer is managed by themselves. The thread manager only executes the analysis and creates the GUI representation.

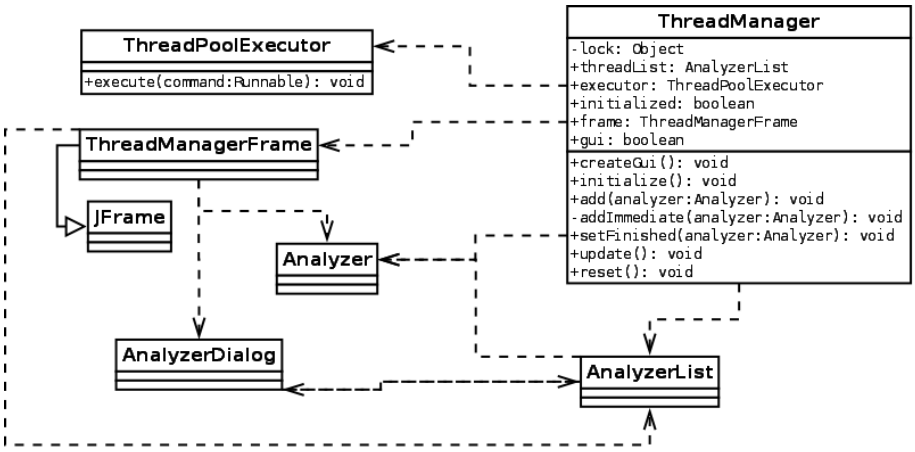


Figure 25: Thread manager class structure.

The classes AnalyzerDialog and AnalyzerList are wrappers for a JPanel and an ArrayList. The AnalyzerList acts as a model for the AnalyzerDialog and the dialog is the viewer of the model. So if the list is changed the dialog is updated too. The thread manager only needs to add analyzers to the list or remove them and the dialog is always updated. If the programmer decides to set the variable

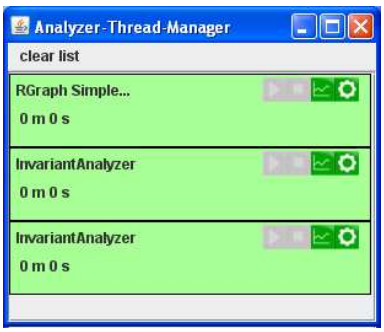


Figure 26: Thread manager frame.

immediateExecution in the analyzer to true, then the ThreadManager creates a thread for this analyzer and starts it without considering other analyzers, which may be currently active. The immediate analyzers do not get a thread panel, the results of their work can only be seen in the protocol or the NetPropertiesDialog.

The thread manager frame (Figure 26) displays a panel for each analyzer which is started. The background color of this panel changes with the status of the analyzer (finished= green, active = blue, paused = orange, canceled = pink). The user is able to pause the work of the analyzer and resume it. This is useful if the processing power of the computer is temporarily needed for other purposes. Then if the analysis is not needed anymore or the options have been set wrong, the user can decide to stop the analyzers work.

As additional information the name of the analyzer and the time it has been active is displayed. Currently the time is calculated by the difference between the point in time where the analyzer started its work and the point where the analyzer finishes, so the time the analyzer is paused is counted too. Furthermore the user can access statistical values by pressing the third button (from the left) in the button row. This calls Analyzer.getHtmlInfoStrings() and should therefore be implemented properly by the derived analyzers. The last button shows a representation of the options with which the analyzer was started. So the user can distinguish between two instances of the same type of analyzer with different option values.

# 11 How to implement a derived analyzer

This section wants to give useful hints for the implementer who wants to adapt an existing analyzer or create a new analyzer for Charlie.

The body for an empty derived analyzer looks like the following program listing.

```
1  package Charlie.analyzer.newAnalyzer;
2  import Charlie.analyzer.*;
3  public class NewAnalyzer extends Analyzer{
4      // member variables to place hier
5
6      public NewAnalyzer(){
7          executeImmediately = true;
8          setName("NewAnalyzer");
9          setUpdateInterval(1000);
10     }
11     public static boolean register(){
12         return AnalyzerManager.register(new NewAnalyzer(),
13                     new ObjectToAnalyze(),
14                     new NewResultObject()
15             );
16     }
17     public void getNewInstance(OptionSet options){
18         NewAnalyzer na = new NewAnalyzer();
19         na.setup(options);
20         return na;
21     }
22     public void cleanup(){
23         // if the analyzer finishes or is stopped this method
               is called
24     }
25     public void initializeInfoStrings(){
26         // statistics about the analysis process are
               initialized here
27     }
28
29     public void analyze(){
30         // here the analysis is performed
31     }
32
33     public void evaluate(){
34         // here the result is evaluated and results are set
35     }
36
37 }
```

At first the fictive analyzer "NewAnalyzer" is placed inside a separate subpackage named Charlie.analyzer.newanalyzer. Certainly the class has to be derived from Charlie.analyzer.Analyzer. Then the implementer places all nec-

essary member variables to perform the analysis.

The constructor should do at least two things:

- call setName(...) to give the analyzer a descriptive name
- decide to set immediateExecution to true or false
- and optionally, if the update of the thread panel is to unresponsive or to fast, adjust the update interval using setUpdateInterval(...). The standard value for update interval is 500, so every 500th call to checkStatus(...) the thread panel is updated.

The static method register() is crucial for the use of the analyzer, if the implementation of this method is forgotten, the method of the base class is called. Because the modifier static cannot be applied together with the modifier abstract, the base class implements an empty method which only places a waring to the console output. But the registration is not done in this case. So the compiler won't show any error if the method is missing in NewAnalyzer. Within register a call to AnalyzerManager.register(...) is done. To call the method, a reference to an instance of NewAnalyzer and the identifying objects has to be created and passed as parameters. If more than one type of result objects can be produced by this analyzer, more calls to AnalyzerManager.register() have to be performed. The analyzer instance itself can be passed more than once to the AnalyzerManager and needs not to be created again each time register() is called. Also if the analyzer is able to process different objects and analyze them, the combination of the object to be analyzed and the produced output object must be told to the analyzer manager.

The method getNewInstance can be as simple to implement as the listing shows. Just create a new instance of NewAnalyzer, call setup using the option set and return the newly created analyzer. But if the choice of the analyzer depends on the value of an option, this option has to be analyzed and then depending on the value, the correct analyzer is created and returned. In this case it is best to derive those special analyzers from the NewAnalyzer class, but this is not a must.

If cleanup() is called the analyzer either finished or was stopped, so if necessary now unused objects or unneeded references may be removed. In short if some cleanup operations can be done after an analysis these operations should be done here. If there are no such useful operations or the operations are not known during the development, this method can be left empty. But the method body has to be in the class since the method is declared abstract in the base class.

If the user presses the statistics button in the thread panel, the panel makes a call to getHtmlInfoStrings() and this method calls initializeInfoStrings() to update the strings. So here useful information about the current state of the analysis can be initialized here. Remember, the variable infoStrings is an array of String objects, with the even indices being the description and the odd numbers being the presented value. The RGAnalyzer e.g. displays the current

number of edges, scc's and states. If this method is left empty nothing will be displayed, when the statistics button is pressed.

The last methods analyze() and evaluate() are called by Analyzer.run(). The analyze() method should implement the algorithm for the analysis and evaluate should gather the results e.g. net properties out of the created object(s). The separation of both aspects allows the replacement of the algorithm while the evaluation can stay the same. RGAnalyzer is a good example since the algorithm is realized in SimpleConstruction and MaximumConstruction, while evaluate() is implemented in RGAnalyzer.

Within analyze() some tasks have to be performed before the real algorithm can start. Using the example from SimpleConstruction these tasks will be explained.

```
1  public void analyze(){
2      this.pn = (PlaceTransitionNet) (options.getObjectToAnalyze
           ());
3      this.co = (ConstructionOptions) options;
4      rg = new RGraph(pn);
5      rg.setBackEdgeOption(co.backEdges);
6      setOutput("reachability graph analyzer:\n");
7      setOutput("computing rechability graph using simple  
           firing rule");
8      try {
9          rg = construct();
10         options.setResultObject(rg);
11     }catch (Exception e){
12         setOutput("simple construction failed due to safety 
               exception.");
13         e.printStackTrace();
14     }
15 }
```

At first the member variables of the analyzer have to be initialized using the option set. In this case, the object which is analyzed is a PlaceTransitionNet object that is stored in the member variable pn. Also a member variable with the type of the used option set should be created, so that the cast has to be done only once. Then other options are evaluated (co.backEdges). If the analyzer generates output that will be seen later in the protocol window, the output must be stored using setOutput(...), otherwise the text will not be displayed automatically.

SimpleConstruction owns a method named construct() which generates the reachability graph using the initialized member variables of the class. Depending on the complexity of the algorithm, the algorithm may be implemented within analyze() or use several methods or even external objects. Before leaving the analyze method the result object has to be set in the option set.

The method evaluate() can use the resultObject and draw conclusions to apply net properties or generate additional output, which explains the conclusions. If this method is left empty and the conclusions are drawn inside analyze()

nothing will happen. Analyzer.run () automatically calls this method, so the separation of tasks should be preferred by the implementer.

These simple steps should be enough to get a working analyzer

# Part IV
# A textual interface for Charlie

## 12 The textual interface

Since some of the analyses use large amounts of memory, the software should be used without graphical user interface on large machines with more memory. Or the analyses should be performed quietly in the background while doing other work. So all the analyzes must be able to be invoked by using command line parameters. The new structure of analyzers and option sets makes realization of this feature possible.

The necessary steps are the following:

- Define a syntax for the command line arguments.
- Create a class that registers all known analyzers, then divides the argument list into pieces for each analyzer.
- Extend all option sets with a method that initializes the option set with the parameter string.
- It must be possible to perform all possible analyses with one call, so the parameters must be distinguished somehow.

### 12.1 Syntax of analyzer parameters and fix parameters

The existing option sets define options of the following types: boolean, int, File and String. Those types must be initialized by the parameters provided by the console. Since parsing parameters can be very complicated, there are some restrictions and rules. So the syntax is defined as follows:

1. Each parameter has to end with a semicolon.
2. The parameter description and the value must be separated by an equal sign.
3. An analyzer is selected by the parameter –analyze=...; The name of the analyzer is not a valid value for the parameter, but the result of the analysis. This means for example for calculating a place invariants the parameter has to look like this:

   ––analyze=PInvariant;

   By naming the result of the analysis, the textual interface follows the design which was introduced with the graphical user interface. The prefix of this parameter must be *–analyze*. The two leading hyphens are needed to separate the parameters of each analyzer. The pattern for more than one analyzer is the following:

```
... --analyze=RG;   ... --analyze=Trap;   ...--analyze=Props;...
  | all RG parameters| all Trap param.|   ....
```

The parameters which belong to an analysis must follow the –analyze parameter, if they are placed before –analyze or behind the next "–analyze" the parameter is not associated with the analysis and is therefore ignored.

4. Boolean parameters are initialized by using the values 0 for false and 1 for true. A Boolean parameter looks like this:

   ```
   backEdges=1;
   ```

5. Integer parameters are initialized by placing the value behind the equal sign.

   ```
   maxConstDeph=1000;
   ```

6. File parameters are initialized by placing the path to the file behind the equal sign.

   ```
   exportFile=D:\data\examples_old\travel.apnn;
   ```

7. String parameters are initialized the same way like file parameters. But the string parameter must not contain a semicolon, otherwise the next parameter will not be interpreted correctly.

8. The net file which will be analyzed must be passed to the program by using the –netfile parameter. If this parameter is omitted the program cannot start.

9. The parameter –sequential controls the behavior of the program and disables or enables parallel processing of analyses.

10. The output of analyzers can be stored in a text file using the parameter –outputfile. An existing output file is overwritten without further questions. If the parameter is not stated, the output appears in the console window.

This small set of rules keeps the system simple and easy to use. If a parameter is not set, the initial values defined in the option set are used to perform the analysis. So each member variable in an option set should be initialized with a standard value. If not the analyzer should be aware that parameters may be uninitialized.

## 12.2 Sequential vs. parallel execution

The analyzer system offers two ways of performing multiple analyses. The first way is to process the analyses one after another, the second way uses all available CPU cores to perform as many analyses as possible at one time. The parameter which controls this behavior is –sequential, it is a Boolean parameter and

must therefore initialized with 0 or 1. If set to true all analyses are started if the previous analysis has finished. This is done by packing the option sets into each other. The member variable nextStage is filled with the next option set. If the analyzer finishes and informs the analyzer manager by calling AnalyzerManager.analyzerHasFinished(). The manager inspects the nextStage value and if it is not set to null, a new analysis is started. The results and the output are passed as previous result to the next option set. So a recursive behavior is realized and all analyzers are executed by once calling AnalyzerManager.compute(options).

If –sequential is set to false all created option sets are passed to the analyzer manger and the thread manager puts them into his queue and takes care that the defined number of threads is always active.

**Multiple instances of the same analyzer** Option sets offer a certain number of parameters which change the result of an analysis. The graphical user interface already gives the user the possibility to start a special analysis twice or more with changed parameters. This behavior is implemented in the textual interface too. If an analysis should be performed twice the parameters just have to be stated twice .

```
... --analyze=RG; backEdges=1; --analyze=RG; backEdges=0; ...
```

The parameter parser does not care how often a special –analyze parameter is called. The only thing that counts is the correct syntax.

## 12.3 Class structure of the textual interface

The realization of the textual interface only needed one extra class and about 10 additional methods in existing classes. The new analyzer system proved its simple extendability. The new class for using parameters provided by the console is Charlie.Charlie (Figure 27).

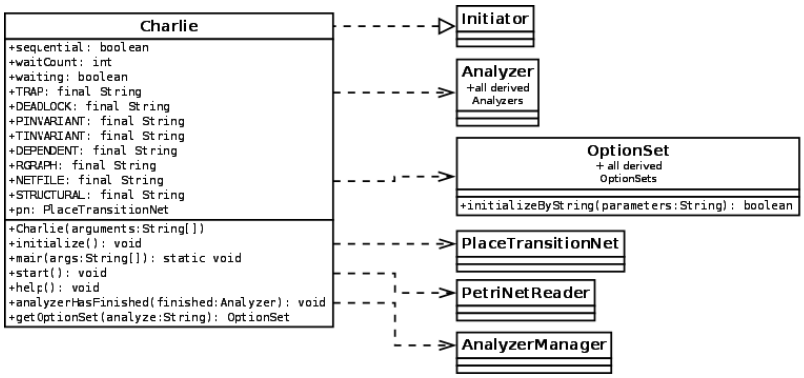


Figure 27: Class structure for class Charlie.Charlie.

Surely the textual interface needs to know all analyzers and their option sets as well as the corresponding result objects.

The initialize method just calls register on all special analyzers, then they wait for work. The start method first extracts the fix parameters for the net file, output file and the processing method (sequential/parallel), then the parameter string is split into smaller strings by using the StringTokenizer class provided by java,util. Using this class the pattern "–" splits the parameter string. So all parameters for an analyzer are now available in this small string.

Then the analyze parameter is evaluated and the correct OptionSet object is returned by getOptionSet(...). After some additional checks and setup actions the option set is either placed in the ArrayList setList or placed in the nextStage member variable of the previous option set object. This decision is made by evaluating the sequential variable. If sequential is true then the nextStage object is used. To start the analyses a call to AnalyzerManager.compute(...) has to be performed, once if sequential is true using the first option set which was created or one call for each element of the setList. Then the start method puts the main thread to sleep and checks temporarily if the last analysis has finished. If the thread would continue the main method of Charlie would be left before the analyzers could complete their work.

Here the variable waitCount helps to determine when to stop sleeping and store the output to a file or display the output. If the sequential mode has been selected, AnalyzerManager only calls the initiator class once, so waitCount is set to 1 in this case. If the parallel execution mode is set, then the AnalyzerManager calls Charlie.analyzerHasFinished(...) every time an analyzer finishes his work. So waitCount is equal to the length of the setList which holds all option sets.

The help method uses the text file help.txt to display hints for the usage of Charlie. By using a file, the text may be easily modified without needing to change code.

## 12.4 Examples

Here some examples are presented to get more familiar with the syntax.

```
// create the reachability graph and calculate use stubborn
    reduction
java -cp Charlie.jar Charlie.Charlie --netfile=travel.apnn; --
    analyze=RG; stubborn=1;
// calculate the reachability graph using the maximum fire
    rule
java -cp Charlie.jar Charlie.Charlie --netfile=travel.apnn; --
    analyze=RG; rule=MAXIMUM;
// calculate traps and proper sets
java -cp Charlie.jar Charlie.Charlie --netfile=travel.apnn; --
    analyze=Trap; properSets=1;
// calculate deadlocks and check the dtp property
java -cp Charlie.jar Charlie.Charlie --netfile=travel.apnn; --
    analyze=Deadlock; computeDTP=1;
```

```
// calculate the reachability graph and store the output to
    log.txt
java -cp Charlie.jar Charlie.Charlie --netfile=travel.apnn; --
    outputfile=log.txt --sequential=0; --analyze=RG;
```

## 12.5 Overview over the option sets and their parameters

In this part the implemented option sets are examined for their variables

DeadlockOptions

| variable with initial value | comment |
|---|---|
| compute = false; | compute is always set to true by initializeByString() ignored if passed as parameter |
| export = false; | the export into a file is enabled, this value is automatically set if the exportFile could be set ignored if passed as parameter |
| properSets = false; | is set to true if the proper sets need to be computed |
| computeDtp = false; | if set to true, the DTP property is checked |
| exportFile = null; | a file handler for the export file is created |

TrapOptions

| variable with initial value | comment |
|---|---|
| compute = false; | compute is always set to true by initializeByString() ignored if passed as parameter |
| export = false; | the export into a file is enabled, this value is automatically set if the exportFile could be set ignored if passed as parameter |
| properSets = false; | is set to true if the proper sets need to be computed |
| exportFile = null; | a file handler for the export file is created |

DependentSetOptions

| variable with initial value | comment |
|---|---|
| decompose = false; | decompose dependent sets |
| sdsFile= ""; | the file name for strong dependent sets |
| adsFile = ""; | the file name for abstract dependent sets |
| adsConFile= ""; | the file name for connected abstract dependent sets |
| sds = true; | if set to true strong dependent sets are computed |
| ads = true; | if set to true abstract dependent sets are computed |

ConstructionOptions

| variable with initial value | comment |
|---|---|
| rule = SIMPLE; | Here the fire rule is determined. |
| boundedness = true; | check the boundedness of a net |
| maxDepth =0; | the maximum depth the reachability graph is constructed |
| completed = true; | not used for parameters |
| stubborn = false; | if set to true stubborn reduction is used |
| backEdges = true; | if set to true the back edges are stored too |

InvOptions

| variable with initial value | comment |
|---|---|
| transitions = true; | Here the fire rule is determined. |
| deleteTrivial = false; | deletion of trivial invariants |
| coverage = true; | check if the net is covered with invariants |
| extendedCoverage= true; | check extended coverage of the net |
| enableMCSC= false; | not used |
| exportFile= ""; | the export file for the invariants |

StructuralOptions – This option set owns no parameters.

Each option name is translated exactly as parameter name, so the variable exportFile is reflected in the parameter exportFile=...; .

## 12.6 Summary

The textual interface is realized by installing only one new class which makes use of the existing structures.

The interface offers a simple but powerful way to access analyses from the command line. The analyzer system is used to create a batch-like way of process-

ing analyses and the user can switch between sequential or parallel processing. The initialization of the option sets is integrated into them, so the tasks are done where the knowledge is. The rather strict rules for the syntax do not handicap a comfortable use, but make extraction of parameter values easier.

# Part V
# Summary

The new Charlie version is now ready for the future. Today not every (small) feature of the original version is integrated into the new version, because some are extended parallel to this work and others will be replaced by more powerful versions. The implementation of the textual interface was the last task that was done, so the new structure had to prove itself in this first test. I would say that the structure passed this test very good. Except from a new class and few methods, which are only needed for the textual interface nothing had to be touched in existing classes. The analyzer manager and thread manager worked independently from any GUI elements and were easy to access.

The work produced a guideline how to implement an additional module or analyzer in Charlie. Together with my study work [AF2008] the system of classes is explained to the programmer.

The user also benefits from the applied changes, a better looking user interface which also is more user friendly and additional features which enable more opportunities.

# Part VI
# Appendices

# 13 Appendix A - Utility classes

During the implementation a small set of utility classes were created, which perform often repeated operations or realize special user interface components. Since Charlie is still enhanced and further analyzers are being developed, these classes might help to create the user interface for the new analyzers.

**package GUI.util**

Align.java This class provides a single method alignToParentWindow(JFrame parent , JFrame child), which aligns the child frame next to the parent frame. If a module uses a frame next to the small dialog in the main window, this frame can be simply aligned next to the main windows by calling this method.

ElapsedTime.java This class offers a delayed update to a JLabel. The time is updated using a timer and SwingUtilities.invokeLater() so that the update is made using the AWT event dispatching thread.

ExportFileAction.java This class implements the class AbstractAction and is used by the class ExportJCheckBoxMenuItem, to realize the reaction on the press of a button. Then a file chooser is displayed and the selected file can be retrieved.

ExportJCheckBoxMenuItem.java This GUI element offers a check box and a label which can be added to menus. If the menu item is clicked a file chooser appears using ExportFileAction. The user can define a suggested file name, which is automatically set to the file chooser so that if the user is satisfied with the name and directory of the file, he only needs to press "o.k.".

FileDisplayDialog.java If an output is created or stored to text file and needs to be displayed in the user interface, this class can be passed a file handler or a string and then a dialog window is displayed at the position of the mouse pointer. The class handles the reading of the file.

HTML.java If the application needs to read HTML files or store output which is presented as HTML page (e.g. the GUI generator output) then this class offers static methods which store an object of the type java.swing.text.html.HTMLDocument as HTML or text file and also converts a string that contains HTML tags or a HTMLDocument to a simple text without the HTML tags.

Html2Text.java Here HTMLEditorKit.ParserCallback is implemented, so handler methods for the occurrence of starting or ending tags are implemented, the tags are then replaced by white space or line breaks.

JComboBoxDelete.java If the user wants to remove an item from a combo box he normally has no possibility to do so. This class enables the combo box to remove the selected item by pressing the 'del' or 'entf' key on the keyboard or pressing the right mouse button and select remove item from the appearing pop up menu. The order of removing the item is then passed to the associated combo box model.

MultiLineToolTip.java Normally the system decides when to end a line and start a new one. If the user wants more control of the tooltips of components, he can choose to use MultiLineToolTip. This class takes the provided string and searches for line breaks, then each line is wrapped into a pair of p-tags and the first line is automatically painted bold.

MyButton.java The assignment of images to a button normally involves multiple steps. This class offers to create a button that uses an image and optionally a roll-over-image only by providing the path to the image files as string. A tooltip text can also be passed. The handling of mouse entered or mouse left events is completely done by the class.

The only method the user must implement is the mouseClicked(...) method, which handles the actions taken when the mouse button is clicked inside the image. The buttons in the thread panel are created this way. The following listing shows how simple it is to instantiate such a button.

```
1 startPauseButton= new MyButton("resources/tp/play.jpg
    ","play/pause"){
2     public void mouseClicked(MouseEvent e){
3         setStatusPaused();
4     }
5 };
```

MyMouseWheelListener.java The use of the mouse wheel offers quick scrolling or changing values without having to press the mouse button very often. So this class can be applied to the following user interface components: javax.swing.JSpinner, javax.swing.JSlider, javax.swing.JComboBox, GUI.dialog.DialogPanel.

The implemented mouseWheelMoved(...) method checks the type of the source by using the *instanceof* keyword and then calls the appropriate method for the component.The constructor of the listener offers to provide a value by which the value of the component model is decremented or incremented. This class can be extended to apply this behavior to other components too. Then a new method for the component should be introduced and the mouseWheelMoved(...) method should get a new check for the desired component type.

PropertyIo.java Properties are used by the application and some modules to store the current state in a file and to remember settings that have

been applied by the user. An example would be the last file that was opened or the maximum number of loaded tools in the GUI generator. These values are assigned to a java.util.Property object and associated with a key string. This class offers to load or store property objects. The methods are declared as static and therefore no instantiation of PropertyIo is needed. The creation of readers and writers is done by the class so the user is freed from this tasks.

StringHashMap.java a hash map originally uses keys to identify an object that is associated with this key. If strings are used as key objects the string a = "A" and the string b="A" are different keys to the hash map because they are different objects. But if the contents of the string is equal the associated object can be identified by using the contents and not the object itself.

XMLReader.java To parse a XML file and get a XML document object, several steps need to be taken XMLReader offers the static method readXmlFile(...), which only needs a file object and then performs the reading process. An object of the type org.w3c.dom.Document is returned and can be traversed using the XML node structure. If an empty XML document object needs to be created the method createNewXmlDocument() can be used.

XMLWriter.java If the gui generator needs to update a XML description for a tool, because the location of tool binary has changed, then the XML document object is updated and stored to a file. The static method writeToXmlFile(...) can be used for this.

FileSaver.java If a file needs to be loaded or stored, the user may be informed if he tries to overwrite a file or if the file he wants to open does not exists.

If the user wants to save a file and the file is already existing, then a message is displayed "Do you want to overwrite...?". If the user accepts the dialog is closed an the selected file is returned. If he does not want to overwrite, the dialog appears again and the user may change the file name or directory or abort the whole process.

If the user wants to open a file, the file should exist, if not the file chooser appears again and the user can change the file name.

This behavior would have to be implemented every time a file chooser is needed to open or save a file. Within Charlie several open/save operations can be taken so this class saves a lot of additional code. The use of the file saver is very easy too, as the following listing shows. Only two lines of code, combined with a check are enough to use this class.

The text parameter in showSaveDialog can be left empty or null, the FileSaver then offers a standard text for the overwrite question.

The extension for the file may be defined too and is automatically appended to the created file handle if the user did not supply a different extension.

```
1  // get a file handle for opening using FileSaver.
       showOpenDialog()
2  FileSaver fs = new FileSaver();
3  File openFile = fs.showOpenDialog(null,
4      new FileNameExtensionFilter("app-session file",".
           app_session"));
5  if (openFile != null){
6      // the user selected a file to open and did not
           abort
7      ...
8  }
9
10 // get file handle for saving using FileSaver.
       showSaveDialog()
11 FileSaver fs = new FileSaver();
12 File saveFile = fs.showSaveDialog(null,
13         "Do you want to overwrite this file",
14         "app_session");
15 if (saveFile != null){
16     // the user selected a file to save and did not
           abort
17     ...
18 }
```

**package GUI.debug**

DebugCounter.java When developing a program, often output to the console is generated to check the value of objects or display warnings and messages. DebugCounter offers a convenient way to capture these messages and store them into a log file. So the developer can provide help if something goes wrong.

The output is listed with numbers and continually stored until the defined maximum log file size is reached, then the contents is cleared. If the user should not see all the output but the creation of the file is needed though, the value of DebugCounter.printImmediately can be set to false. Then the output is captured and stored but not printed to the system console. If larger or smaller sizes of the log file are needed the maximum file size can be set by changing the value of DebugCounter.maxLogFileSize. Both variables are static as well as the methods of DebugCounter.

```
1  // use of DebugCounter.inc
2  DebugCounter.inc("Class.method(): Warning message
       file does not exist" + file.getName());
```

```
3 // write the log to a file
4 DebugCounter.writeToFile(new File("Charlie_log.txt"))
      ;
5 // prevent output to the console
6 DebugCounter.printImmediately= false;
7 // set the log file size to 1 MB
8 DebugCounter.maxLogFileSize = 1000000;
```

# 14 Appendix B

The relevant lines for deduction of rules in the source code of Charlie/pn/Analyzer.java.

```
001 package Charlie.pn;
...
053 if(!pn.ft0.isEmpty()){
054     results.addResult(results.B,new Result(false));
055     Out.println("input transitions:\n"+pn.ft0);
056     results.addResult(results.SB,new Result(false));
057     results.addResult(results.S,new Result(false));
058 }
...
076 if(pn.isConservative()){
077     results.addResult(results.CSV,new Result(true));
078     results.addResult(results.SB,new Result(true));
079     results.addResult(results.B,new Result(true));
080 }else{
081     results.addResult(results.CSV,new Result(false));
082 }
...
114 public void determineNetClass(){
115     if(pn.isSM()){
116         results.addResult(results.NC,new Result("SM"));
117         results.addResult(results.SB,new Result(true));
118         results.addResult(results.B,new Result(true));
119     }else if(pn.isMG()){
120         results.addResult(results.NC,new Result("SG"));
121     }else{
...
209 public synchronized InvAnalyzer evalInvariants(InvAnalyzer
    ia){
...
213         SparseMatrix sm = null;
214         sm = ia.invariants();
215         invariants = sm.rows();
216         if(ia.getCoveragOption()){
217         if(ia.isCovered(sm)){
218
219         if(ia.getOptions().transitions){
220             results.addResult(results.CTI,
                    new Result(new Boolean(true)));
221         }else{
222             results.addResult(results.CPI,
                    new Result(new Boolean(true)));
223             results.addResult(results.SB,
                    new Result(true));
224             results.addResult(results.B,
                    new Result(true));
```

```
225             }
226         }else{
...
439 public boolean hasDTP(DeadlockAnalyzer da){
440     boolean ret = da.checkDTP();
...
443     if(ret && pn.homogenous && pn.hasNBM()){
444         results.addResult(results.DST,new Result(false));
445         if(!isNotExtendedSimple() ){//&& !isMarkedGraph())
    {
...
448             setLive(true);
449         }
450     }else if(!ret){
451         if( isFreeChoice() ||
                isExtendedFreeChoice() ||
                isStateMachine() ||
                isMarkedGraph()){
452             //results.addResult(results.L,
                        new Result(false));
453             setLive(false);
454             }
455         }
456         results.addResult(results.DTP,new Result(ret));
457
458         return ret;
459     }
...
```

# References

[AF2008] Andreas Franzke – "Concept for redesigning Charlie", Study Project, BTU Cottbus, CS Dep., 2008.

[AF2009] Ansgar Fischer – "Analysis of timed Petri nets using reachability graphs", Diploma Thesis, BTU Cottbus, CS Dep., 2009.

[BSB2008] Christoph Bommer, Markus Spindler, Volker Barr – Software-Wartung, dpunkt Verlag, 2008.

[Charlie] Martin Schwarick – Charlie - A software tool to analyse Petri nets, BTU Cottbus, CS Dep., http://www-dssz.informatik.tu-cottbus.de/software/charlie.html.

[GIMP] GIMP, GNU Image Manipulation Program, http://www.gimp.org.

[GS2002] Gernot Starke – Effektive Software-Architekturen, Carl Hanser Verlag, 2002.

[IW1998] Ivo Wessel – GUI-Design Richtlinien zur Gestaltung ergonomischer Windows Applikationen, Carl Hanser Verlag, 1998.

[JDOC] F. Allimant – Java 2SE 6 Documentation (HTML help version) http://www.allimant.org/javadoc/index.php.

[LYX] Lyx – The document processor http://www.lyx.org.

[MH2007] Prof. Dr.-Ing. Monika Heiner – slides from the lecture "dependability engineering & Petri nets", pn06_structuralProperties.sld2.pdf "structural Petri net analysis", January 2007.

[MS2006] Martin Schwarick – "A software to to analyse Petri net models', Master Thesis, BTU Cottbus, CS Dep., 2006.

[Notepad++] Editor Notepad++, http://notepad-plus.sourceforge.net/.

[SLOCcount] David A. Wheeler – Source, SLOCcount - a tool for counting the source lines of code, http://www.dwheeler.com/sloccount/.

[SNOOPY] Snoopy – a software tool to design and animate hierarchical graphs, among others Petri nets, BTU Cottbus, CS Dep., http://www-dssz.informatik.tu-cottbus.de/software/snoopy.html.

[TableLayout] TableLayout – a free layout manager, https://tablelayout.dev.java.net/.