The Petri Net Kernel[1]

Revision 1.1

Documentation of the Application Interface

Ekkart Kindler Michael Weber

Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
D-10099 Berlin

15th September 1998

[1]Supported by the Deutsche Forschungsgemeinschaft (DFG) within the research project "Petri Net Technology"

Preface

This is an English short version of the German documentation of the *Petri Net Kernel*. Originally, we intended to distribute the Petri Net Kernel in Germany, only. Due to many request from other countries we have decided to provide an English short version, which should be sufficient for developing applications for the Petri Net Kernel.

We are grateful for any feedback concerning this documentation and the Petri Net Kernel itself. We apologize for any shortcomings due to the quick process of creation of the English documentation.

Berlin in September 1998
E. Kindler and M. Weber

Contents

1 Introduction **5**

1.1 The concept 5

1.2 Structure of this document 6

2 Using the Petri Net Kernel **7**

2.1 An application 7

2.2 Starting an application 9

2.2.1 Simple applications 9

2.2.2 More complex applications 10

3 Defining Net Types **11**

4 The Editor **18**

5 Details of Petri Net Kernel **19**

5.1 Interfaces 19

5.2 Exception handling 19

5.3 Detailed Description of the Petri Net Kernel 19

5.3.1 Package *Petri net structure* 21

5.3.1.1 Class Net 21

5.3.1.2 Class Node 26

5.3.1.3 Class Place 27

5.3.1.4 Class Transition 28

5.3.1.5 Class Arc 28

5.3.2 Package *Petri net types* 29

5.3.2.1 Class Specification 29

5.3.2.2 Class Marking 30

5.3.2.3 Class Mode 31

5.3.2.4 Class Inscription 31

5.3.3 The package *editor interface*—the class PNK . . . 32

Bibliography **36**

Index **37**

1 Introduction

The *Petri Net Kernel* (*PNK*) provides an infrastructure for translating ideas for analyzing, simulating, or verifying Petri nets into action; viz. for integrating the idea into a tool. The PNK stores and graphically represents the Petri net; the net may be accessed and modified by calling methods of the PNK. This way, the developer of a Petri net application is relieved from building a parser for loading nets from a file or from dealing with graphical user interfaces. Rather, he can concentrate on implementing the algorithmic idea. In this documentation, we briefly describe the functionality of the PNK and how it can be used for building Petri net tools.

The PNK provides several interfaces:

1. The *application interface* comprises the functions and methods of the PNK for programming applications. This interface is relevant for the *application programmer*.
2. The *editor interface* describes the interaction between the PNK and an editor. It comprises functions which are provided by the PNK and functions which must be provided by an editor for a proper interaction with the PNK.

   The PNK is equipped with a simple editor. But, this editor can be replaced by another editor, provided it conforms to the editor interface. Though important for designing nice editors, we do not describe the editor interface in this documentation.
3. The PNK is not restricted to a fixed Petri net type or a fixed set of Petri net types. It is possible to define new Petri net types. The *net type interface* describes how an application programmer may define his own Petri net type with his specific extensions.

This documentation mainly describes the application interface and the net type interface. Moreover, we describe the functionality of the editor which is provided with the PNK. This functionality is relevant for a prospective user of a Petri net application; the user should at least know, how to load, save, and edit nets and how to start application functions.

1.1 The concept

The PNK is object oriented. The PNK consists of classes with access and modification methods. Basically, there are the classes Net, Place, Transition, and Arc which correspond to the respective concepts in Petri nets. There are methods for inserting and deleting places, transition, and arcs. For a place there are methods which return the preset or the postset of this transition. Moreover, there are methods of Net which return the set of all places or the set of all transitions.

Each place is associated with a marking, which may be accessed and modified by corresponding methods. The legal marking and its representation, however, depend of the specific Petri net type. This is not fixed by the PNK itself but by the application when creating a new instance of a

net. The type is given by a parameter which is a class Specification which describes the legal markings of the net type, the legal arc-inscriptions, as well as further extensions of the net type. The PNK is provided with some standard parameters for standard net types (e.g. P/T-systems). In Chapter 3 we will discuss the definition of parameters for other net types.

Beyond providing an infrastructure for easily implementing Petri net tools, there have been some other objectives. For example, the PNK should support the design of distributed Petri net tools. For this reason, the application interface consists of two parts: The first part, allows to access all objects directly and is called *object interface*. The second part, does not access places, transitions, and arcs of a net directly, but refers to these objects by unique *keys*. This interface is called *symbolic interface*. The symbolic interface only allows simple objects such as string and integer as parameters and return values. This way, these parameters can be easily sent as messages. Therefore, applications which only refer to the symbolic interface can be easily executed on a distributed computing network. The object interface, however, is more efficient.

All access and modification operations are available in both, the object interface and the symbolic interface. In principle, applications may use both interfaces; but lose the benefits of both.

1.2 Structure of this document

This is an evolving document; whenever we have some time, we will try to extend and to improve it. Currently, there is an informal introduction to the use of the PNK in Chapter 2 and an example for defining your own Petri net type for the PNK in Chapter 3. In Chapter 5, there is a complete list of all classes, methods, and exceptions defined by the PNK. In particular, Sect. 5.3.1 lists all methods for accessing and modifying nets, Sect. 5.3.2 lists all classes which must be defined for a new Petri net type, and Sect. 5.3.3 lists all methods for displaying information to the user and for organizing user interaction.

Chapter 4 provides a brief introduction to the user interface of the editor which is provided with the PNK.

2 Using the Petri Net Kernel

In this chapter, we show how to use the PNK for developing a Petri net tool. To this end, we discuss a simple simulator for P/T-systems which is implemented by help of the PNK. First, we will discuss the simulation function itself. Then, we will show how this simulation function can be included into an executable tool.

2.1 An application

Here, we describe all functions necessary for simulating a net. The functions are written in the object oriented programming language Python. The meaning of most Python programming constructs is immediate; therefore, we just start to use Python and and teach it by using it. When exploiting Python specific constructs, we will explain these 'online'. For a more comprehensive introduction to Python, we refer to the book of Lutz [Lut96]. For the moment, we just explain one surprising syntactical issue of Python: There is no way to define blocks in Python except for indentation! There are no keywords `begin` or `end` or parentheses for defining the block structure.

Table 2.1 shows all functions which are necessary for the simulator. The function `is_activated` checks whether a transition is activated. The function `get_activated_transitions` returns a list of all activated transitions of a net. The function `fire` simulates the occurrence of a transition. At last, function `animate` implements the simulation loop; including the interaction with the user.

The function `is_activated` checks, if every place in the preset of the transition has at least one token (is marked). To this end, method `get_Preset` returns a list of all places in the preset of the transition. Then, the `for` loop iterates over this list. If there is a place which has no token, `is_activated` returns `FALSE`; otherwise it returns `TRUE`. The method `get_current_Mark` returns the marking of the place; then, method `is_marked` checks whether the returned marking contains at least one token. Note that `TRUE` and `FALSE` are no Python keyword, but have been defined as constants.

The function `get_activated_transitions` initializes an empty list `[]` and then iterates over all transitions of the net, which are returned by method `get_Transitions`. A transition is appended to the list by the Python method `append`, if the transition is activated. In the end, the list of all activated transitions is returned as result.

The function `fire` iterates over the preset and the postset of an activated transition and changes the marking according to the occurrence rule of P/T-systems. It subtracts `ONE` token from every place in the preset and adds `ONE` token to every place in the postset. The current marking of a place is returned by method `get_current_marking`; it is changed by method `change_current_marking`. Adding and subtracting tokens is achieved by the corresponding methods on markings `add` and `subtract`, respectively.

```
from Build_Application import *

TRUE  = 1
FALSE = 0

ONE   = ST_Marking(ST_Specification(),'1')

def is_activated(transition):
    for place in transition.get_Preset():
      if not place.get_current_Mark().is_marked(): return FALSE
    return TRUE

def get_activated_transitions(net):
    activated = []
    for transition in net.get_Transitions():
       if is_activated(transition):
         activated.append(transition)
    return activated

def fire(transition):
    for place in transition.get_Preset():
      place.change_current_Mark(place.get_current_Mark().subtract(ONE))
    for place in transition.get_Postset():
      place.change_current_Mark(place.get_current_Mark().add(ONE))

def animate(net):
    transition = net.select_Transition(get_activated_transitions(net))
    while transition <> None:
      fire(transition)
      transition = net.select_Transition(get_activated_transitions(net))

Build_Application(ST_Specification,animate)
```

Table 2.1: The simulation function

The function `animate` realizes the simulation. The simulation takes several steps. First, the set of all activated transitions is calculated by the above functions. Then, the user may select a transition from this set (or cancel the simulation). At last the selected transition fires. The simulation process starts again, if the user did not cancel the simulation. The PNK offers several methods for displaying information to the user and for user selections. Here, we have used the method `select_Transition`. This function, takes a list of transitions as parameter. Then, the user may select one of these transitions which is returned as result. If the user cancels the selection (by pressing a particular button) the method returns `None`.

This finishes the definition of our simulator example. In Sect. 2.2 we will show how this simulator is brought into action. Note that the functions `is_activated` and `fire` do not consider arc inscriptions in order to simplify the example. Moreover, the simulator cannot deal with nets with multiple arcs between a place and a transition. Of course, there are methods which allow to access the in-going and out-going arcs of a transition and to access the arc inscriptions. With these methods, it is easy to build a more general simulator.

2.2 Starting an application

In the previous section, we have seen how to write an application. Now, we will see how to integrate an application into an executable tool—with an editor, a graphical user interface and visualization of information. There are two ways, to build a tool. The first is easy to use; the second is more flexible.

2.2.1 Simple applications

The simplest way to build a tool is the function[1] Build_Application which is defined in the corresponding PNK module. This function takes two arguments: The first argument specifies the used Petri net type; the second argument specifies the used application function. Here, we use ST_Specification for P/T-systems and the function `animate` from the previous section: `Build_Application(ST_Specification,animate)`. Note that for making the PNK known to the application functions, we need to include the statement

```
from Build_Application import *
```

into the file in which the application is defined (cf. Tab. 2.1).

That's all. If you have a file `simulate.py` with the text of Tab. 2.1 and if the PNK is properly installed, you can start the simulator by `python simulate.py` from the command line. Then, you can graphically edit a net, load and save the net, and start the simulation by pressing a button `animate` in the menu bar. When, the simulation is started, you may choose from the highlighted transitions and the chosen transition is fired. The simulation is stopped, when you press the button `cancel` or no transition is enabled any more.

[1] Actually, Build_Application is not a function but a class. Here, we use the constructor of this class as a function.

2.2.2 More complex applications

The above method for starting an application is quite simple. But it not very flexible because the application is started under the control of the editor. For example, it is not possible to have an application with several nets. Therefore, there is a more flexible method to build applications; this method, however, requires some more knowledge about Python and its `Tkinter` module because it is necessary to write a more or less complex control program.

A simple control program for the above simulator is shown in Tab. 2.2. The control program defines a `control_window` with two buttons `Start`

```
# Global Variables:
#   ed: Editor
#   control_window: Tk

def start():
    global ed
    animate(ed.get_net())

def quit():
    global control_window
    control_window.destroy()
    del control_window

control_window = Tk()
ed =  Editor(control_window,ST_Specification)

start_button = Button(control_window, text = 'Start', command = start)
start_button.pack(side = LEFT, fill= BOTH, expand = YES)

quit_button =  Button(control_window, text = 'Quit', command = quit)
quit_button.pack(side = LEFT, fill= BOTH, expand = YES)

control_window.mainloop()
```

Table 2.2: The control program

and `Quit`, which are associated with the corresponding functions `start` and `quit`, respectively. Moreover, the control program starts an editor for net type `ST_Specification`. When the `Start` button is pressed, the function `start` calls the function `animate` with the net of the editor. The function `quit` terminates the complete application. Note that it is necessary to transfer the control to the `Tkinter` mainloop by the statement `control_window.mainloop()` after starting the editor and defining the buttons. Otherwise, the control program will terminate without any effect. The mainloop of `Tkinter` will transfer the control to the editor and to the functions `start` and `quit` by so-called *callback functions*. For details on `Tkinter` we refer to [Lut96].

By writing your own control program you can start several editors even for different net types and define interactions between different nets. For example, you could start an editor for high-level nets and another editor in which the application generates a process of this net.

3 Defining Net Types

The PNK can be used with any kind of Petri net; the net type is given as a parameter when a new instance of a net is created. In this section, we describe how to define a parameter for your favourite net type.

The parameter for a net type must provide the following information about the net type:

1. External and internal representation of markings of the net type.
2. Addition and subtraction functions for markings as well as some comparison functions.
3. External and internal representation of arc inscriptions.
4. A description of allowed transition modes.
5. A conversion of arc inscriptions into a corresponding marking for a given mode.
6. Some further extensions concerning the net as a whole—such as definitions of allowed variables and function symbols in arc inscriptions.

This information is provided by defining four classes which inherit from the predefined classes Marking (for 1. and 2.), Inscription (for 3.), Mode (for 4. and 5.), and Specification (for 6.). The class Specification plays a particular role since it is this class which is passed as a parameter to the PNK; therefore, Specification has references to the three other classes which make up a new Petri net type. These references are represented by attributes `Marking`, `Inscription`, and `Mode` of class Specification, which must be set accordingly.

Each of the four classes must provide some methods which are listed and described in Section 5.3.2. Moreover, method calls of these classes may raise some predefined exceptions: `ILLEGAL_MARKING`, `ILLEGAL_SUBTRACTION`, `ILLEGAL_INSCRIPTION`, `ILLEGAL_MODE`, `ILLEGAL_EXTENSION`.

Here, we present the definition of a new net type by help of an example. In this example, we define a simple version of high-level nets where each token is a natural number. We define the four classes HL_Specification, HL_Marking, HL_Inscription, and HL_Mode. After defining these classes, we can build some application for this Petri net type and start it by `Build_Application(HL_Specification,app_function)`.

In the implementation of this Petri net type, we exploit that Python is an interpreted programming language. For the declaration of functions, we use Python's syntax, which is compiled at runtime. You can find some more information on this net type and the used Python features in the German documentation [HHK+98].

```
DECLARATION         = 'Declaration'
VARIABLES           = 'Variables'

class HL_Specification(Specification):

  def __init__(self):
      self.Marking =     HL_Marking
      self.Inscription = HL_Inscription
      self.Modetype =    HL_Mode

      self.clear()

  def extensions(self):
      return repr([DECLARATION,VARIABLES])

  def set(self,extension,string):
      if extension == DECLARATION:
        self.Declaration = string
        self.Code = None
      elif extension == VARIABLES:
        self.Variables = eval(string)
      else:
        raise ILLEGAL_EXTENSION

  def get(self,extension):
      if extension == DECLARATION:
        return self.Declaration
      elif extension == VARIABLES:
        return repr(self.Variables)
      else:
        raise ILLEGAL_EXTENSION

  def clear(self):
      self.Declaration = ""
      self.Variables = []

      self.Code = None

  def check(self):
      # Checks whether the declaration of functions is syntactically correct
      # and compiles it to internal Code (self.Code)
      if self.Code:
        return 1
      else:
        try:
          self.Code = compile(self.Declaration,'<PNK-Declaration>','exec')
        except:
          self.Code = None
          return 0
        return 1
```

Table 3.1: The class HL_Specification

```
class HL_Marking(Marking):

  def __init__(self,specification):
      self.Specification = specification
      self.clear()

  def set(self,string):
      # Converts an external representation of a marking into an internal
      # repr.
      # Exampl. of an external representation of markings:
      #   "[1,4,3]" or "[]" or "[1,2,1]"
      self.Mark = {}
      try:
        ListMarking = eval(string)
        for value in ListMarking:
          if not self.legal(value): raise NO_LEGAL_MARKING
          if self.Mark.has_key(value):
            self.Mark[value] = self.Mark[value] + 1
          else:
            self.Mark[value] = 1
      except:
        self.Mark = {}
        raise NO_LEGAL_MARKING

  def get(self):
      # Converts the internal representation of a marking into
      # an external represenation (i.e. a string)      ListMarking  = []
      for value in self.Mark.keys():
        i = self.Mark[value]
        while i > 0:
          ListMarking.append(value)
          i = i - 1
      return repr(ListMarking)

  def clear(self):
      # Sets the marking to the empty marking
      self.Mark = {}

  def check(self):
      return 1
```

Table 3.2: The class HL_Marking (part 1)

```
def is_marked(self):
    # returns 1 (TRUE), if is not empty;  0 (FALSE) otherwise
    if len(self.Mark) > 0:
       return 1
    else:
       return 0

def contains(self,mark):
    # checks whether self >= mark
    for value in mark.Mark.keys():
      if not self.Mark.has_key(value): break
      elif mark.Mark[value] > self.Mark[value]: break
      else: continue
    else:
      return 1
    return 0

def add(self,mark):
    # Adds two markings self and mark.
    marking = HL_Marking()
    for value in self.Mark.keys():
      marking.Mark[value] = self.Mark[value]
    for value in mark.Mark.keys():
      if marking.Mark.has_key(value):
        marking.Mark[value] = marking.Mark[value] + mark.Mark[value]
      else:
        marking.Mark[value] = mark.Mark[value]
    self.set(marking.get())
    return marking

def subtract(self,mark):
    # subtracts mark from self; if the result would be negative it
    # raises an exception
    if self.contains(mark):
      marking = HL_Marking()
      for value in self.Mark.keys():
        marking.Mark[value] = self.Mark[value]
      for value in mark.Mark.keys():
        marking.Mark[value] = marking.Mark[value] - mark.Mark[value]
        if marking.Mark[value] == 0:
          del marking.Mark[value]
      self.set(marking.get())
      return marking
    else:
      raise ILLEGAL_SUBTRACTION

#--------------------------------------------------------------------
# Internal methods (need not be implementet for other Petri net types)
#--------------------------------------------------------------------

def legal(self,value):
    # Checks whether value is a legal value for a token:
      if value >= 0: return 1
    return 0
```

Table 3.3: The class HL_Marking (part 2)

```
class HL_Inscription(Inscription):

  def __init__(self,specification):
      self.Specification = specification
      self.clear()

  def set(self,inscription):
      # Converts an external representation of an arc inscription
      # into an internal one;  for this particular Petri net type,
      # there is no conversion
      self.Inscription = inscription

  def get(self):
      # Converts an internal representation of an arc inscription
      # into an external one
      return self.Inscription

  def clear(self):
      # Sets the default for an arc inscription
      self.Inscription = '[0]'

  def check(self):
      return 1
```

Table 3.4: The class HL_Inscription

```
class HL_Mode(Mode):

  def __init__(self,specification):
      self.Specification = specification
      self.clear()

  def get(self):
      return repr(self.Ass)

  def set(self,string):
      self.Ass = eval(string)
      self.Variables = self.Ass.keys()
      for variable in self.Variables:
        Marking = HL_Marking()
        if not Marking.legal(self.Ass[variable]):
          self.Ass = {}
          self.Variables = []
          raise ILLEGAL_MODE

  def clear(self):
      # Initializes the mode; here, all variables are set to 0
      self.Ass = {}
      Variables = self.Specification.Variables
      for variable in Variables:
         self.Ass[variable] = 0

      self.Variables = self.Ass.keys()

  def check(self):
      return 1
```

Table 3.5: The class HL_Mode (part 1)

```
# Methods exist_next and next allow to systematically generate all
# Modes starting from the initial one;  as long as there is another
# Mode exist_next returns 1 (TRUE)

def exist_next(self):
    if len(self.Variables) <> 0
      return 1
    else
      return 0

def next(self):
    if len(self.Variables) <> 0:
      sum = 0
      for variable in self.Variables:
        sum = self.Ass[variable] + sum
      if self.Ass[self.Variables[len(self.Variables)-1]] == sum:
        self.Ass[self.Variables[len(self.Variables)-1]] = 0
        self.Ass[self.Variables[0]] = sum + 1
      elif self.Ass[self.Variables[0]] >= 1:
        self.Ass[self.Variables[0]] = self.Ass[self.Variables[0]] - 1
        self.Ass[self.Variables[1]] = self.Ass[self.Variables[1]] + 1
      else:
        i = 1
        while self.Ass[self.Variables[i]] == 0: i = i + 1
        self.Ass[self.Variables[i+1]] = self.Ass[self.Variables[i+1]] + 1
        self.Ass[self.Variables[0]]   = self.Ass[self.Variables[i]] - 1
        self.Ass[self.Variables[i]]   = 0

def eval(self,inscr):
    # Evaluates an arc inscription in a given mode into a marking.
    if self.Specification.Check():
      try:
        Env = {}
        eval(self.Specification.Code,Env)
        return HL_Marking().set(repr(eval(inscr.Inscription,Env,self.Ass)))
      except:
        return None
    else:
      return None
```

Table 3.6: The class HL_Mode (part 2)

4 The Editor

Here, we briefly describe the use of the editor which is provided with the Petri Net Kernel. When an application such as the one presented in Chapter 2 is started, the user will see a menu bar with menus for editing nets and for starting the application function.

There are two menus for editing nets. The `File` menu allows to load and save nets as usual. The `Page` menu allows to create a new page. Moreover, you can hide or make visible pages by check buttons.

A net can be drawn on these different pages. For creating a place on a page, you need to click into this page with the left mouse button. It depends on the selection in the `Page` menu whether a transition or a place is created. The `Page` menu can be opened by pressing the right mouse button. An arc can be drawn by help of the middle mouse button by dragging the mouse from the source to the target object. The attributes of places, transitions, and arcs can be modified by corresponding menus which can be opened by clicking with the right mouse button into the corresponding object. Moreover, these menus provide functions for deleting objects. A place or a transition may be moved to another position by clicking into the object and moving the mouse accordingly.

All mentioned menus provide some more functionality, such as printing pages (i.e. writing the file to a PostScript file) etc. Here, we mention only two special features. Arcs of a net may be equipped with so-called *drag points* which allow to have arcs following a polygon. A drag point can be added from the `Arc` menu or from the menu of another drag point (remember that these menus are opened by clicking with the right mouse button into the corresponding object). The polygon may be smoothened by the corresponding function in the `Arc` menu. Drag points can be moved in the same way as places and transitions.

The second feature are *merge places*. A net may be split into several pages The correspondence between different pages may be established by merging places from different pages. This can be achieved by selecting the corresponding function in the `Place` menu. Note that you can merge arbitrary many places (even on the same page); all places are considered to be identical and have the same marking and the same initial marking. Together, the page concept and the merge places allow to graphically structure nets—this structure, however, is hidden from the application.

5 Details of Petri Net Kernel

5.1 Interfaces

The application interface of the Petri Net Kernel is available in two different ways (cf. Sect. 1.1). On the one hand, there is an object interface and on the other hand, there is a symbolic interface . The object interface deals with the object directly. This interface is more efficient then the symbolic interface.

The symbolic interface only uses the basic data types integer and string. It deals with objects via keys, which are integer numbers. Each object has its own key. The use of the symbolic interface simplifies and makes faster the communication between Petri Net Kernel and an application of it if they run in a computer network.

In Sect. 5.3.2, the classes which define a net type interface (cf. Chap. 1) are described.

5.2 Exception handling

In order to point out incorrect use of Petri Net Kernel to the application developer, the Petri Net Kernel raises exceptions of Python's *exception handling* (for details see e. g. [Lut96]). In Sect. 5.3 the description of those function which can raise an exception shows an exception type.

Tables 5.1 and 5.2 classify and describes the exception types and their expressions. Each exception type and each expression cause (together) an exception message to the function with incorrect use of Petri Net Kernel. The first exception message shows the name of the exception; the second one gives a hint to the cause of exception.

5.3 Detailed Description of the Petri Net Kernel

In the following, we describe the packages of Petri Net Kernel. The package *Petri net structure* (Sect. 5.3.1) provides the functionality to manage a Petri Net independent from tokens, markings, firing, and extensions. The package *Petri net type* (Sect. 5.3.2) gives a description of markings, arc inscriptions, firing modes, and extensions in a special Petri net type. If the distribution of Petri Net Kernel does not support a specific Petri net type, an application developer may implement his own package. The description of the package provides the name of functions which the developer must implement. Finally, the package *user interface* (Sect. 5.3.3) handles the messages between the user (via an editor) and applications.

Each method of each relevant class is described below. The description uses the identifiers listed in Tab. 5.3. These identifiers stand for the following objects or types. Optional arguments of methods are given in square brackets. The notation contains the type of the argument, its real identifier, and its default (for details to use optional arguments in methods see e. g. [Lut96]). The first parameter `self` of methods is omitted.

Table 5.1: Exception types and their expressions

Exception type		
Exception name	1. exception message	expression
OBJECT_ERROR	Object_Error	Object_Error
EDGE_ERROR	Edge_Error	Source_Error
		Target_Error
		Arc_Error
KEY_ERROR	Key_Error	Object_key_Error
		Node_key_Error
		Edge_key_Error
		Place_key_Error
		Transition_key_Error
TYPE_ERROR	Type_Error	String_Type_Error
FILEFORMAT_ERROR	Fileformat_Error	Fileformat_Error
		Fileformat_Error2
ILLEGAL_MARKING	Illegal Marking	
ILLEGAL_INSCRIPTION	Illegal Inscription	
ILLEGAL_SUBTRACTION	Subtraction Impossible	
ILLEGAL_MODE	Illegal Mode	
ILLEGAL_EXTENSION	Illegal Extension	
EVAL_IMPOSSIBLE	Evaluation Imossible	

Table 5.2: Exception expressions and their messages

Expression	2. exception message
Object_key_Error	Object_key does not exist!
Node_key_Error	Node_key does no exist!
Place_key_Error	Place_key does no exist!
Transition_key_Error	Transition_key does not exist!
Edge_key_Error	Edge_key does not exist!
Node_Error	This node does not exist!
Edge_Error	There isn't edge between!
Arc_Error	Arcs only exist between different types!
Source_Error	This source does not exist!
Target_Error	This target does not exist!
String_Type_Error	I need a string!
Int_Type_Error	I need an integer!
Fileformat_Error	Fileformat_Error found!
Fileformat_Error2	Object_id's not found

Table 5.3: Identifiers and objects used in method descriptions

Identifier	stands for object of a type or a class
id	`integer`
string	`string`
file	File
list_of_type	list of a type `type`
tupel_of(type1, type2, ...)	Tupel of elements of given types *type1*, *type2*, ...
graph	Graph, Net
object	Node, Edge, Transition, Place, Arc
node	Node, Transition, Place
edge	Edge, Arc
net	Net
place	Place
trans	Transition
spec	Specification
mark	Marking
mode	Mode
inscr	Inscription

The return value of the method is given below the name of the method. Exceptions which can be raised by the method are given at the end of its description.

5.3.1 Package *Petri net structure*

5.3.1.1 Class Net

This class handles the structural properties of a Petri net and the whole symbolic interface of the package *Petri net structure*.

Superclass : Graph

File : Graph.py, Netz.py

`Net([Specification: specification = ST_Specification] )`

Constructor
initializes a Petri net with the Petri net type `specification`. The net is initially empty.

`delete()`

deletes all objects of the net and finally the net itself.

`change_Name(`*string*`)`

changes the name of the net.
Error: `TYPE_ERROR:` *string*

`load(`*file*`)`

`net`

loads data from the open file `file` and returns a net. The existing net will be overwriten.
Error: `FILEFORMAT_ERROR: file`

`save(`*file*`)`

saves the net in the open file *file*. Only the initial marking of the net will be saved.

Object interface

`get_Objects()`
list_of_object — returns all objects (i. e. nodes *and* arcs) of the net as a list.

`get_Nodes()`
list_of_node — returns all nodes (i. e. places and transitions) of the net as a list.

`get_Edges()`
list_of_edge — returns all arcs of the net as a list.

`get_Places()`
list_of_place — returns all places of the net as a list.

`get_Transitions()`
list_of_trans — returns all transitions of the net as a list.

`get_Specification()`
spec — determines the type of the net and returns it as an object of the class Specification.

Symbolic interface

`get_Object_keys()`
list_of_id — returns the keys of all objects of the net as a list of keys.

`get_Object_key(`*object*`)`
id — returns the key of *object*.
Error: `OBJECT_ERROR:` *object*

`get_Object(`*id*`)`
object — checks if *id* is a valid key, then it returns the object of that key.
Error: `KEY_ERROR:` *id*

`get_Node_keys()`
list_of_id — returns the keys of all places and transitions of the net as a list of keys.

`change_Node_Name(`*id*`, `*string*`)`
changes the name of a place or of a transition with the key *id* to *string*.
Error: `KEY_ERROR:` *id*`; TYPE_ERROR:` *string*

`get_Node(`*id*`)`
node — checks whether *id* is a valid key of a place or of a transition. If so, it returns the corresponding object.
Error: `KEY_ERROR:` *id*

`get_Node_Name(`*id*`)`
string — returns the name of a place or of a transition with the key *id*.
Error: `KEY_ERROR:` *id*

`create_Place([String: name = ""], [integer: id = None], [String: init_mark = None])`

id — creates a place optionally with the name `name`, the key `id`, and the initial marking `init_mark`. It returns the key of the generated place.
Error: `TYPE_ERROR: name; ILLEGAL_MARKING: init_mark`

`get_Place_keys()`

list_of_id — returns the keys of all places of the net as a list of keys.

`get_Place(id)`

place — checks whether *id* is a valid key of a place. If so, it returns the corresponding place.
Error: `KEY_ERROR:` *id*

`delete_Place(id)`

deletes the place with key *id*.
Error: `KEY_ERROR:` *id*

`create_Transition([string: name = ""], [integer: id = None])`

id — creates a transition optionally with the name `name` and the key `id`. It returns the key of the generated transition.
Error: `TYPE_ERROR:` *string*

`get_Transition_keys()`

list_of_id — returns the keys of all transitions of the net as a list of keys.

`get_Transition(id)`

trans — checks whether *id* is a valid key of a transition. If so, it returns the corresponding transition.
Error: `KEY_ERROR:` *id*

`delete_Transition(id)`

deletes the transiton with key *id*.
Error: `KEY_ERROR:` *id*

`get_Node_Edges(id)`

list_of_id — returns the keys of all incoming and outgoing arcs of the place/transition with key *id* as a list of keys.
Error: `KEY_ERROR:` *id*

`get_Edges_in(id)`

list_of_id — returns the keys of all incoming arcs of the place/transition with key *id* as a list of keys.
Error: `KEY_ERROR:` *id*

`get_Edges_out(id)`

list_of_id — returns the keys of all outgoing arcs of the place/transition with key *id* as a list of keys.
Error: `KEY_ERROR:` *id*

`get_Preset_keys(id)`

list_of_id — returns the keys of all transitions/places in the preset of the place/transition with key *id* as a list of keys.
Error: `KEY_ERROR:` *id*

`get_Postset_keys(`*id*`)`

list_of_id — returns the keys of all transitions/places in the preset of the place/transition with key *id* as a list of keys.
Error: `KEY_ERROR:` *id*

`create_Arc(`*id1*`, `*id2*`, [integer: id = None], [string: inscr = None])`

id — creates an arc from place/transition with key *id1* to transition/place with key *id2* optionally with its own key `id` and the arc inscription `inscr`. It returns the key of the generated arc.
Error: `KEY_ERROR:` *id1*`, `*id2*`; EDGE_ERROR:` *id1*`, `*id2*`, [`*id1*`, `*id2*`]; ILLEGAL_INSCRIPTION: inscr`

`get_Edge_keys()`

list_of_id — returns the keys of all arcs of the net as a list of keys.

`get_Edge(`*id*`)`

edge — checks whether *id* is a valid key of an arc. If so, it returns the corresponding arc.
Error: `KEY_ERROR:` *id*

`get_Source(`*id*`)`

id — returns the source node of the arc with key *id*.
Error: `KEY_ERROR:` *id*

`get_Target(`*id*`)`

id — returns the target node of the arc with key *id*.
Error: `KEY_ERROR:` *id*

`delete_Edge(`*id*`)`

deletes the arc with key *id*.
Error: `KEY_ERROR:` *id*

`Specification_extensions()`

list_of_string — returns a list of all extension names of this net type.

`Specification_set(`*string1*`, `*string2*`)`

changes the content of the extension *string1* to *string2*.
Error: `ILLEGAL_EXTENSION:` *string1*`; TYPE_ERROR:` *string2*

`Specification_get(`*string*`)`

string — returns the content of the extension *string*.
Error: `ILLEGAL_EXTENSION:` *string*

`Specification_check()`

1 or 0 — checks whether the specification of the type of the net is correct. It returns 1 if the specification is correct and 0 else.

`Specification_clear()`

changes all extensions of the type of the net to its default.

`change_initial_Mark(`*id*`, `*string*`)`

changes the initial marking of the place with key *id* to the marking represented by *string*.
Error: `KEY_ERROR:` *id*`; ILLEGAL_MARKING:` *string*

`get_initial_Mark(`*id*`)`

string — returns a representation of the initial marking of the place with key *id*.
Error: `KEY_ERROR:` *id*

`clear_initial_Mark(`*id*`)`

changes the initial marking of the place with key *id* to the default (empty marking) of the used net type.
Error: `KEY_ERROR:` *id*

`change_current_Mark(`*id*`,` *string*`)`

changes the current marking of the place with key *id* to the marking represented by *string*.
Error: `KEY_ERROR:` *id*`;` `ILLEGAL_MARKING:` *string*

`get_current_Mark(`*id*`)`

string — returns a representation of the current marking of the place with key *id*.
Error: `KEY_ERROR:` *id*

`is_current_Mark_marked(`*id*`)`

1 oder 0 — checks whether the place with key *id* is currently marked. It returns 1 if the place is marked and 0 else.
Error: `KEY_ERROR:` *id*

`contains_current_Mark(`*id*`,` *string*`)`

1 oder 0 — checks whether the current marking of the place with key *id* contains a marking represented by *string*. It returns 1 if the place contains the marking *string* and 0 else.
Error: `KEY_ERROR:` *id*`;` `ILLEGAL_MARKING:` *string*

`add_current_Mark(`*id*`,` *string*`)`

string — adds the marking represented by *string* to the current marking of the place with key *id*. It returns the resulting current marking of that place as a string.
Error: `KEY_ERROR:` *id*`;` `ILLEGAL_MARKING:` *string*

`sub_current_Mark(`*id*`,` *string*`)`

string — subtracts the marking represented by *string* from the current marking of the place with key *id*. It returns the resulting current marking of that place as a string.
Error: `KEY_ERROR:` *id*`;` `ILLEGAL_MARKING:` *string*`;` `ILLEGAL_SUBTRACTION:` *string*

`clear_current_Mark(`*id*`)`

changes the current marking of the place with key *id* to default (empty marking) of used Petri net type.
Error: `KEY_ERROR:` *id*

`check_Marking(`*string*`)`

1 oder 0 — checks whether *string* is a valid marking of the used Petri net type. If so, it returns 1 and 0 else.

`change_Mode(`*id*`,` *string*`)`

changes the firing mode of the transition with key *id* to *string*.
Error: `KEY_ERROR:` *id*`;` `ILLEGAL_MODE:` *string*

`get_Mode(`*id*`)`

string — returns the firing mode of the transition with key *id* as a string.
Error: `KEY_ERROR:` *id*

`exist_next_Mode(`*id*`)`

1 or 0 — checks whether the transition with key *id* has a further firing mode. It returns 1 if there exists such a mode and 0 else.
Error: `KEY_ERROR:` *id*

`next_Mode(`*id*`)`

changes the firing mode of the transition with key *id* to its next mode.
Error: `KEY_ERROR:` *id*`;` `ILLEGAL_MODE:` *id*

`Mode_eval(`*id*`,` *string*`)`

string — evaluates the arc inscription *string* at the current mode of the transition with key *id*. It returns a marking of the net as a string.
Error: `ILLEGAL_INSCRIPTION:` *string*`;` `ILLEGAL_MARKING:` *string*`;` `KEY_ERROR:` *id*

`clear_Mode(`*id*`)`

changes the firing mode of the transition with key *id* to default of used net type.
Error: `KEY_ERROR:` *id*

`change_Inscription(`*id*`,` *string*`)`

changes the inscription of the arc with key *id* to *string*.
Error: `KEY_ERROR:` *id*`;` `ILLEGAL_INSCRIPTION:` *string*

`get_Inscription(`*id*`)`

string — returns the inscription of the arc with key *id*.
Error: `KEY_ERROR:` *id*

`check_Inscription(`*string*`)`

1 or 0 — checks whether *string* is a valid inscription of the used net type. It returns 1 if *string* is valid and 0 else.

`clear_Inscription(`*id*`)`

changes the inscription of the arc with key *id* to default of the used net type.
Error: `KEY_ERROR:` *id*

5.3.1.2 Class Node

The class Node describes methods which are basic for both places and transitions, of a Petri net. The classes Place and Transition are derived from that class; therefore, one can use these methods in these classes too.

File : Graph.py

`change_Name(`*string*`)`

changes the name of the node to *string*.
Error: `TYPE_ERROR:` *string*

`get_Name()`
string — returns the name of the node.

`get_Edges()`
list_of_edges — returns a list of all incoming and outgoing arcs of the node.

`get_Edges_in()`
list_of_edges — returns a list of all incoming arcs of the node.

`get_Edges_out()`
list_of_edges — returns a list of all outgoing arcs of the node.

`get_Preset()`
list_of_node — returns the preset of the node (the source nodes of incoming arcs) as a list.

`get_Postset()`
list_of_node — returns the postset of the node (the target nodes of outgoing arcs) as a list.

5.3.1.3 Class Place

This class describes places of a Petri net. They have an initial and a current marking.

Superclass : Node

File : Netz.py

`Place(`*net*`, `*string*`, [integer: id = None], [string: mark = None])`
Constructor
creates a place with name *string* in *net*. The place gets optionally the key `id` and the initial marking `mark`. The key `id` will be used only if in *net* there does not exists an object with identical key. If a marking for the place is not specified the default marking of the used net type will be used as the initial marking of that place.
Error: `TYPE_ERROR:` *string*`; ILLEGAL_MARKING: mark`

`delete()`
deletes the place and its incoming and outgoing arcs also in the net, to whom the place belonged.

`change_initial_Mark(`*mark*`)`
changes the initial marking of the place to *mark*.
Error: `ILLEGAL_MARKING:` *mark*

`get_initial_Mark()`
mark — returns the initial marking of the place.

`change_current_Mark(`*mark*`)`
changes the current marking of the place to *mark*.
Error: `ILLEGAL_MARKING:` *mark*

`get_current_Mark()`
mark — returns the current marking of the place.

5.3.1.4 Class Transition

This class describes transitions of a Petri net. They got a firing mode which is generated from the specification of the net.

Superclass : Node

File : Netz.py

`Transition(`*net*`, `*string*`, [integer: id = None])`
Constructor
creates a transition with the name *string* in *net*. The transitions gets optionally the key `id`. This key will be used if in *net* there does not exists an object with identical key.
Error: `TYPE_ERROR:` *string*

`delete()`
deletes the transition and its incoming and outgoing arcs also in the net, to whom it belonged.

`change_Mode(`*mode*`)`
changes the firing mode of the transition to *mode*.
Error: `ILLEGAL_MODE:` *mode*

`get_Mode()`
mode returns the current firing mode of the transition.

5.3.1.5 Class Arc

This class describes arcs of a Petri net. They only exist between a place and a transition. They may have an arc inscription.

Superclass : Edge

File : Graph.py, Netz.py

`Arc(`*net*`, `*node1*`, `*node2*`, [integer: id = None], [string: inscr = None])`
Constructor
checks whether *node1* and *node2* are nodes of different type (one has to be a place and the other one a transition). It initialies in *net* an arc between the mentioned nodes optionally with key `id` and the arc inscription `inscr`. The key `id` will be used if in *net* there does not exists an object with the identical key.
Error: `EDGE_ERROR:` *node1*`, `*node2* `[`*node1*`, `*node2*`]; ILLEGAL_INSCRIPTION: inscr`

`delete()`
deletes the arc also in its nodes and in the net, to whom it belonged.

`get_Source()`
node returns the source node of the arc.

`get_Target()`
node returns the target node of the arc.

`change_Inscription(`*inscr*`)`

changes the arc insription to *inscr*.
Error: `ILLEGAL_INSCRIPTION:` *inscr*

`get_Inscription()`

inscr returns the arc insription.

5.3.2 Package *Petri net types*

The classes of this package describe the used Petri net type. An application developer, who cannot use the types of the current distribution, has to develope his own package. The following gives hints to him, what he has to implement. He and other users can find here the description of the net type interface.

5.3.2.1 Class Specification

This class describes the used Petri net type. It contains attributes for classes of markings, firing modes, and arc inscriptions. Furthermore this class organizes the extensions of the net type as follows. Each extension has a name and contents. The name of the extension is the key to get its contents. For example the extension `VARIABLES` may contain all used names for variables of a high-level net.

For implementing examples see the implementation of the standard class Specification and the special classes of some implemented Petri net types. The specification class of a specific Petri net type should inherit from this class.

File : Specification.py

Attributes `Marking` holds the current class for markings.
`Inscription` holds the current class for arc insriptions.
`Mode` holds the current class for firing modes.

`Specification()`

Constructor
initialies a container class of a Petri net type. The attributes above will be bound to relevant classes.

`set(`*string1* `,` *string2*`)`

changes the contents of extension *string1* to *string2*.
Error: `ILLEGAL_EXTENSION:` *string1* `;` `TYPE_ERROR:` *string2*

`extensions()`

list_of_string returns all extension names of the net type as a list.

`get(`*string*`)`

string returns the contents of extension *string*.
Error: `ILLEGAL_EXTENSION:` *string*

`check()`

1 oder 0 checks whether the specification is correct. Maybe it compiles the specification. It returns 1 if the specification is correct and 0 else.

`clear()`

changes all extensions to default.

5.3.2.2 Class Marking

This class describes the collection of tokens on a place of a Petri net. The class of markings of a specific Petri net type should inherit from this class.

File : Specification.py

`Marking(`*spec*`, [string: initial = None])`

Constructor
initialies an object which functions as a marking. It is a marking of net type *spec*. It gets optionally `initial` as an initial value.
Error: `ILLEGAL_MARKING: initial`

`set(`*string*`)`

changes the marking to the equivalent of *string*.
Error: `ILLEGAL_MARKING:` *string*

`get()`

string returns the string equivalent of the marking.

`is_marked()`

1 oder 0 checks whether the marking does not hold the default (empty marking) of the net type. It returns 1 if the marking is not empty and 0 else.

`contains(`*mark*`)`

1 oder 0 checks whether the marking contains (it is greater or equal) an other marking *mark*. It returns 1 if the first marking is greater or equal and 0 else.

`add(`*mark*`)`

mark adds to the marking an other marking *mark*. It returns the object itself which contains now the sum of both markings.

`subtract(`*mark*`)`

mark subtracts from the marking an other marking *mark*. It returns the object itself which now contains the difference of both markings.
Error: `ILLEGAL_SUBTRACTION:` *mark*

`check()`

1 oder 0 checks whether the obejct contains a valid value of a marking. It returns 1 if the value is correct and 0 else.

`clear()`

changes the marking to the default (empty marking) of the used net type.

5.3.2.3 Class Mode

This class describes a firing mode of a transition. It will be used to transform an arc inscription to a marking of the used Petri net type under the firing mode. The class of modes of a specific Petri net type should inherit from this class.

File : Specification.py

`Mode(`*spec*`, [string: initial = None])`
Constructor
initialies an object which functions as firing mode of a net of type *spec*. It gets optionally an own firing mode `initial`.
Error: `ILLEGAL_MODE: initial`

`set(`*string*`)`
changes the firing mode to the equivalent of *string*.
Error: `ILLEGAL_MODE:` *string*

`get()`
string returns the string equivalent of the firing mode.

`exist_next()`
checks whether a further firing mode could be compute. It returns 1 if
1 oder 0 there exists a further firing mode and 0 else.

`next()`
computes a new firing mode and sets the object to the new one.
Error: `ILLEGAL_MODE`

`eval(`*inscr*`)`
computes from the arc inscription *inscr* an marking under the firing
mark mode. It returns a marking of the used Petri net type.
Error: `ILLEGAL_MARKING:` *inscr*

`check()`
checks whether the object contains a valid value of a firing mode. It
1 oder 0 returns 1 if the firing mode is correct and 0 else.

`clear()`
changes the firing mode to default of the net type.

5.3.2.4 Class Inscription

This class describes arc inscriptions. The class of inscriptions of a specific Petri net type should inherit from this class.

File : Specification.py

`Inscription(`*spec*`, [string: initial = None])`
Constructor
initialies an object which functions as an arc inscription of a net of type *spec*. It gets optionally an inscription `initial`.
Error: `ILLEGAL_INSCRIPTION: initial`

`set(`*string*`)`

changes the arc inscription to the equivalent of *string*.
Error: `ILLEGAL_INSCRIPTION:` *string*

`get()`

string returns the equivalent of the arc insciption.

`check()`

1 oder 0 checks whether the object contains a valid value of an arc inscription of the net type. It returns 1 if the arc inscription is correct and 0 else.

`clear()`

changes the arc inscription to the default of the net type.

5.3.3 The package *editor interface*—the class PNK

The class PNK describes the editor interface of the Petri Net Kernel. To use it there must be an editor which supports the methods of this class. The editor of the Petri Net Kernel distribution is such an editor. The methods effect that the editor shows the described actions.

Super class : Net

File : pnk.py

`PNK([Specification: specification = Specification])`

Constructor
initialies a Petri net and its editor interface optionally with `specification`.

`show_information(`*string*`)`

sends a message *string* to an editor.

`reset_emphasize()`

resets all emphasized net elements to their default face in editor.

`reset_annotations()`

deletes all annotations in editor.

`get_information(`*string*`)`

string sends a question *string* to the user via an editor and waits for reply.

Object interface

`emphasize_Nodes(`*list_of_node*`, [string: string = ""])`

sends a list of nodes and a comment `string` to an editor. The editor has to emphasize this set of nodes.

`emphasize_Arcs(`*list_of_edge*`, [string: string = ""])`

sends a list of arcs and a comment `string` to an editor. The editor has to emphasize this set of arcs.

`unemphasize_Nodes(`*list_of_node*`)`

sends a list of nodes to the editor. The editor has to reset the emphasize of the given set of nodes.

`unemphasize_Arcs(`*list_of_edge*`)`
: sends a list of arcs to the editor. The editor has to reset the emphasize of the given set of edges.

`annotate_Places(`*list_of_tupel_of(place, string)*`)`
: sends a list of places with annotations on each of them to the editor. The editor has to show these annotations.

`annotate_Transitions(`*list_of_tupel_of(trans, string)*`)`
: sends a list of transitions with annotations on each of them to the editor. The editor has to show these annotations.

`annotate_Arcs(`*list_of_tupel_of(edge, string)*`)`
: sends a list of arcs with annotations on each of them to the editor. The editor has to show these annotations.

`unannotate_Places(`*list_of_place*`)`
: sends a list of places to the editor. The editor has to delete the place annotations of these places.

`unannotate_Transitions(`*list_of_trans*`)`
: sends a list of transitions to the editor. The editor has to delete the transition annotations of these transitions.

`unannotate_Arcs(`*list_of_edge*`)`
: sends a list of arcs to the editor. The editor has to delete the arc annotations of these arcs.

`select_Place(`*list_of_place*`, [string: string = ""])`
: sends a list of places and a comment `string` to the editor. The editor returns a single place from the given set of places.

place

`select_Places(`*list_of_place*`, [string: string = ""])`
: sends a list of places and a comment `string` to the editor. The editor returns a subset of places from the given set of places.

list_of_place

`select_Transition(`*list_of_trans*`, [string: string = ""])`
: sends a list of transitions and a comment `string` to the editor. The editor returns a single place from the given set of transitions.

trans

`select_Transitions(`*list_of_trans*`, [string: string = ""])`
: sends a list of transitions and a comment `string` to the editor. The editor returns a subset of transitions from the given set of transitions.

list_of_trans

`select_Arc(`*list_of_edge*`, [string: string = ""])`
: sends a list of arcs and a comment `string` to the editor. The editor returns a single arc from the given set of arcs.

edge

`select_Arcs(`*list_of_edge*`, [string: string = ""])`
: sends a list of arcs and a comment `string` to the editor. The editor returns a subset of arcs from the given set of arcs.

list_of_edge

Symbolic interface

`emphasize_Node_keys(`*list_of_id*`, [string: string = ""])`
: sends a list of keys of nodes and a comment `string` to an editor. The editor has to emphasize the corresponding set of nodes.

`emphasize_Arc_keys(`*list_of_id*`, [string: string = ""])`
: sends a list of keys of edges and a comment `string` to an editor. The editor has to emphasize the corresponding set of arcs.

`unemphasize_Node_keys(`*list_of_id*`)`
: sends a list of keys of nodes to the editor. The editor has to reset the emphasize of the corresponding set of nodes.

`unemphasize_Arc_keys(`*list_of_id*`)`
: sends a list of keys of arcs to the editor. The editor has to reset the emphasize of the corresponding set of edges.

`annotate_Place_keys(`*list_of_tupel_of(id, string)*`)`
: sends a list of keys of places with annotations on each of the places to the editor. The editor has to show these annotations.

`annotate_Transition_keys(`*list_of_tupel_of(id, string)*`)`
: sends a list of keys of transitions with annotations on each of the transitions to the editor. The editor has to show these annotations.

`annotate_Arc_keys(`*list_of_tupel_of(id, string)*`)`
: sends a list of keys of arcs with annotations on each of the arcs to the editor. The editor has to show these annotations.

`unannotate_Place_keys(`*list_of_id*`)`
: sends a list of keys of places to the editor. The editor has to delete the place annotations of these places.

`unannotate_Transition_keys(`*list_of_id*`)`
: sends a list of keys of transitions to the editor. The editor has to delete the transition annotations of these transitions.

`unannotate_Arc_keys(`*list_of_id*`)`
: sends a list of keys of arcs to the editor. The editor has to delete the arc annotations of these arcs.

`select_Place_key(`*list_of_id*`, [string: string = ""])`
: sends a list of keys of places and a comment `string` to the editor. The

id
: editor returns a single key of a place from the given set of places.

`select_Place_keys(`*list_of_id*`, [string: string = ""])`
: sends a list of keys of places and a comment `string` to the editor. The editor returns a subset of places from the given set of places as a list of

list_of_id
: their keys.

`select_Transition_key(`*list_of_id*`, [string: string = ""])`

id — sends a list of keys of transitions and a comment `string` to the editor. The editor returns a single key of a transition from the given set of transitions.

`select_Transition_keys(`*list_of_id*`, [string: string = ""])`

list_of_id — sends a list of keys of transitions and a comment `string` to the editor. The editor returns a subset of transitions from the given set of transitions as a list of their keys.

`select_Arc_key(`*list_of_id*`, [string: string = ""])`

id — sends a list of keys of arcs and a comment `string` to the editor. The editor returns a single key of an arc from the given set of arcs.

`select_Arc_keys(`*list_of_id*`, [string: string = ""])`

list_of_id — sends a list of keys of arcs and a comment `string` to the editor. The editor returns a subset of arcs from the given set of arcs as a list of their keys.

Bibliography

[HHK+98] J. Hauptmann, B. Hohberg, E. Kindler, I. Schwenzer, and M. Weber. Der Petrinetz-Kern – Dokumentation der Anwendungs-Schnittstelle. Informatik-Berichte 98, Humboldt-Universität zu Berlin, February 1998.

[Kin97] Ekkart Kindler. Der Petrinetz-Kern: Ein einfaches Anwendungsbeispiel. In Jörg Desel, Ekkart Kindler, and Andreas Oberweis, editors, *Algorithmen und Werkzeuge für Petrinetze, 4. Workshop*, number 85 in Informatik-Berichte, pages 13–18. Humboldt-Universität, Institut für Informatik, October 1997.

[Lut96] Mark Lutz. *Programming Python.* O' Reilly, October 1996.

[Sch97] Ines Schwenzer. Konzeption für einen Petrinetz-Kern, 1997. Studienarbeit.

[Web97] Michael Weber. Der Petrinetz-Kern – Eine Aufteilung in Invariantes und Variables. In Jörg Desel, Ekkart Kindler, and Andreas Oberweis, editors, *Algorithmen und Werkzeuge für Petrinetze, 4. Workshop*, number 85 in Informatik-Berichte, pages 56–61, Humboldt-Universität zu Berlin, October 1997. Humboldt-Universität, Institut für Informatik.

Index

application interface, **5**, 19
application programmer, **5**
Arc, **28**
    Arc, **28**
    change_Inscription, **29**
    delete, **28**
    get_Inscription, **29**
    get_Source, **28**
    get_Target, **28**

Build_Application, 9

editor interface, **5**, 32
exception handling, 19

HL_Inscription, 15
HL_Marking, 13, 14
HL_Mode, 16, 17
HL_Specification, 12

ILLEGAL_EXTENSION, **11**
ILLEGAL_INSCRIPTION, **11**
ILLEGAL_MARKING, **11**
ILLEGAL_SUBTRACTION, **11**
Inscription, 11, **31**
    check, **32**
    clear, **32**
    get, **32**
    Inscription, **31**
    set, **32**

Marking, 11, **30**
    add, **30**
    check, **30**
    clear, **30**
    contains, **30**
    get, **30**
    is_marked, **30**
    Marking, **30**
    set, **30**
    subtract, **30**
Mode, 11, **31**
    check, **31**
    clear, **31**
    eval, **31**
    exist_next, **31**
    get, **31**
    Mode, **31**
    next, **31**
    set, **31**

Net, **21**
    add_current_Mark, **25**
    change_current_Mark, **25**
    change_initial_Mark, **24**
    change_Inscription, **26**
    change_Mode, **25**
    change_Name, **21**
    change_Node_Name, **22**
    check_Inscription, **26**
    check_Marking, **25**
    clear_current_Mark, **25**
    clear_initial_Mark, **25**
    clear_Inscription, **26**
    clear_Mode, **26**
    contains_current_Mark, **25**
    create_Arc, **24**
    create_Place, **23**
    create_Transition, **23**
    delete, **21**
    delete_Edge, **24**
    delete_Place, **23**
    delete_Transition, **23**
    exist_next_Mode, **26**
    get_current_Mark, **25**
    get_Edge, **24**
    get_Edge_keys, **24**
    get_Edges, **22**
    get_Edges_in, **23**
    get_Edges_out, **23**
    get_initial_Mark, **25**
    get_Inscription, **26**
    get_Mode, **26**
    get_Node, **22**
    get_Node_Edges, **23**
    get_Node_keys, **22**
    get_Node_Name, **22**
    get_Nodes, **22**
    get_Object, **22**
    get_Object_key, **22**
    get_Object_keys, **22**
    get_Objects, **22**
    get_Place, **23**
    get_Place_keys, **23**
    get_Places, **22**
    get_Postset_keys, **24**
    get_Preset_keys, **23**
    get_Source, **24**
    get_Specification, **22**
    get_Target, **24**
    get_Transition, **23**
    get_Transition_keys, **23**

get_Transitions, **22**
is_current_Mark_marked, **25**
load, **21**
Mode_eval, **26**
Net, **21**
next_Mode, **26**
save, **22**
Specification_check, **24**
Specification_clear, **24**
Specification_extensions, **24**
Specification_get, **24**
Specification_set, **24**
sub_current_Mark, **25**
net type interface, **5**, 19, 29
Node, **26**
change_Name, **26**
get_Edges, **27**
get_Edges_in, **27**
get_Edges_out, **27**
get_Name, **27**
get_Postset, **27**
get_Preset, **27**

object interface, **6**, 19

Petri Net Kernel, **5**, 19
Place, **27**
change_current_Mark, **27**
change_initial_Mark, **27**
delete, **27**
get_current_Mark, **27**
get_initial_Mark, **27**
Place, **27**
PNK, *siehe* Petri Net Kernel
PNK, **32**
annotate_Arc_keys, **34**
annotate_Arcs, **33**
annotate_Place_keys, **34**
annotate_Places, **33**
annotate_Transition_keys, **34**
annotate_Transitions, **33**
emphasize_Arc_keys, **34**
emphasize_Arcs, **32**
emphasize_Node_keys, **34**
emphasize_Nodes, **32**
get_information, **32**
PNK, **32**
reset_annotations, **32**
reset_emphasize, **32**
select_Arc, **33**
select_Arc_key, **35**
select_Arc_keys, **35**
select_Arcs, **33**
select_Place, **33**
select_Place_key, **34**
select_Place_keys, **34**
select_Places, **33**
select_Transition, **33**
select_Transition_key, **35**
select_Transition_keys, **35**
select_Transitions, **33**
show_information, **32**
unannotate_Arc_keys, **34**
unannotate_Arcs, **33**
unannotate_Place_keys, **34**
unannotate_Places, **33**
unannotate_Transition_keys, **34**
unannotate_Transitions, **33**
unemphasize_Arc_keys, **34**
unemphasize_Arcs, **33**
unemphasize_Node_keys, **34**
unemphasize_Nodes, **32**

Specification, 11, 21, **29**
check, **29**
clear, **30**
extensions, **29**
get, **29**
set, **29**
Specification, **29**
ST_Specification, 9
symbolic interface, **6**, 19

Transition, **28**
change_Mode, **28**
delete, **28**
get_Mode, **28**
Transition, **28**