

# checkpn Reference Manual

0.0a

Generated by Doxygen 1.3.8

Tue Nov 9 16:23:59 2004

## Contents

<b>1</b>	<b>checkpn Main Page</b>	<b>1</b>
<b>2</b>	<b>checkpn Hierarchical Index</b>	<b>6</b>
<b>3</b>	<b>checkpn Class Index</b>	<b>7</b>
<b>4</b>	<b>checkpn File Index</b>	<b>7</b>
<b>5</b>	<b>checkpn Class Documentation</b>	<b>8</b>
<b>6</b>	<b>checkpn File Documentation</b>	<b>35</b>

## 1 checkpn Main Page

### 1.1 Introduction

This document describes the implementation of a simple model checker called `checkpn` and based on the object-oriented model checking library `Spot`. It allows to check `LTL` formulae on ordinary `Petri nets`.

- The Petri net is given in a file respecting the syntax defined for the `Prod` tool (a basic example is given below).
- The LTL formula must respect the syntax defined by the `LTL parser` of `Spot`.
- The atomic propositions are simply the names of the Petri net places. In a given reachable marking, a proposition is satisfied if the corresponding place contains at least one token.

After a short description on how install the tool and use it, the main line followed for the implementation of `checkpn` is discussed. A last section is dedicated to an evaluation of the overhead induced by `Spot`.

### 1.2 Downloading and installation

The package can be download at <http://spot.lip6.fr/dl/checkpn-0.0a.tar.gz>.

Instructions for the installation are in the file named `INSTALL`.

### 1.3 Usage

The invocation command is of the form:

```
checkpn [OPTIONS...] petri_net_file
```

where `OPTIONS` are

Actions:

- `-c` display the number of states and edges of the reachability graph,

- `-e` display a sequence (if any) of the net satisfying the formula (implies `-f` or `-F`),
- `-fformula` check the formula `formula`,
- `-Fformula_file` check the formula read from `formula_file`,
- `-g` display the reachability graph of the Petri net,

Options for the translation of the formula into an automaton (implies `-f` or `-F`):

- `-b` branching postponement (`false` by default)
- `-l` fair-loop approximation (`false` by default)
- `-x` try to produce a more deterministic automaton (`false` by default)
- `-y` do not merge states with same symbolic representation (`true` by default)

For more explanations on the four last options of the tool, see the documentation of the parameters of `spot::ltl_to_tgba_fm()`.

A basic Petri net expressed with the syntax of the `Prod` tool and composed by four places and three transitions follows:

```
#place A mk(2*<.>)
#place B
#place CS
#place EX mk(<.>)

#trans t1
in {A:<.>;}
out {B:<.>;}
#endtr

#trans t2
in {B:<.>;EX:<.>;}
out {CS:<.>;}
#endtr

#trans t3
in {CS:<.>;}
out {A:<.>;EX:<.>;}
#endtr
```

In this format, `<.>` designates an uncolored token. Here, all the tokens have to be uncolored as `checkpn` is restricted to ordinary nets. In the initial marking, place A contains two tokens, EX only one and the other places are empty. The syntax used for transitions and their connectives is easy to deduce from this net description. As an example, the LTL formula  $G(B \Rightarrow F\ CS)$  respects the syntax of Spot and is valid for this net. An atomic proposition is satisfied in a given marking if the corresponding place contains at least one token. See the [LTL parser](#) documentation for more details about the LTL syntax.

```
> ./checkpn -e -f'G(B => F CS)' ../samples/simple.net
5 unique states visited
2 strongly connected components in search stack
automaton construction time: 0
checking time: 0

an accepting run exists
Prefix:
  G(FCS | !B) * [2.A + 1.EX]
  | <B:0, CS:0>
```

```

Cycle:
  FCS & G(FCS | !B) * [1.A + 1.B + 1.EX]
  | <B:1, CS:0>
  FCS & G(FCS | !B) * [2.B + 1.EX]
  | <B:1, CS:0>
  FCS & G(FCS | !B) * [1.B + 1.CS]
  | <B:1, CS:1> {Acc[CS]}
  G(FCS | !B) * [1.A + 1.B + 1.EX]
  | <B:1, CS:0>
  FCS & G(FCS | !B) * [2.B + 1.EX]
  | <B:1, CS:0>
  FCS & G(FCS | !B) * [1.B + 1.CS]
  | <B:1, CS:1>

```

As expected an example is found. At the opposite, the negation of the formula leads to an empty product automaton.

```

> ./checkpn -e -f '!G(B => F CS)' ../samples/simple.net
8 unique states visited
0 strongly connected components in search stack
automaton construction time: 0
checking time: 0

no accepting run found

```

## 1.4 Implementation

The implementation of the model checker has been realized in two steps. At first, we have developed a set of classes corresponding to the firing rule of the Petri net theory. Secondly, these classes have been interfaced with the Spot library to produce the final model checker.

### 1.4.1 Firing rule implementation

Three classes form the firing rule module:

- [marking](#) which encodes a Petri net state as an integer vector.
- [compressed\\_marking](#) which encodes a Petri net state as a bit vector for performance issues.
- [petri\\_net](#) which implements the firing rule of a Petri net and offers a parsing function.

These classes have been developed independently of Spot. The methods of the class [petri\\_net](#) are sufficient to produce the reachability graph step by step. The most important methods of this class are:

- `marking* petri_net::get_initial_marking() const;`
- `std::list<int>* petri_net::firable(const marking& m) const;`
- `marking* petri_net::successor(const marking& m, int t) const;`

Moreover, the class function [petri\\_net::parse\(\)](#) allows to create a [petri\\_net](#) from a file description.

All this stuff has been grouped in a library called libpn.

### 1.4.2 Interfacing Spot

The manual of Spot (see the [page](#)) explains how to interface it with a state space generator. The main goal is to masquerade the class `petri_net` as a `TGBA`. More precisely, three abstract classes of Spot have to be derived:

- `spot::state` representing a state of a TGBA,
- `spot::tgba_succ_iterator` allowing to iterate throw the successors of a state,
- and `spot::tgba` representing a TGBA.

In our model checker, the corresponding derivated classes are:

- `pn_state` which implements a `spot::state` by encapsulation of a `compressed_marking`,
- `pn_succ_iterator` implementing a `spot::tgba_succ_iterator`,
- `pn_tgba` which implements a `spot::tgba` by encapsulation of a `petri_net`.

Based on this three new classes our model checker consists in the simple following function `check()`. Notice that this function is a simplified version of the one (`model_check()`) integrated in the model checker.

```
void check(const petri_net* n, const spot::ltl::formula* f) {
    // n is a petri_net returned by petri_net::parse();
    // f is a formula returned by spot::ltl::parse();

    // ensure these atomic propositions correspond to place names
    // here the test is done a posteriori, we could also have checked
    // the validity of the AP during the parsing using a customized
    // environment.
    const string* s;
    if ((s = check_at_prop(n, f))) {
        std::cout << "the atomic proposition '" << *s
                    << "' does not correspond to a place name" << std::endl;
        return;
    }

    // roughly speaking, a bdd_dict object maps a BDD variable to an atomic
    // proposition and vice-versa
    spot::bdd_dict dict;

    // collect atomic propositions used in f
    spot::ltl::atomic_prop_set* sap = spot::ltl::atomic_prop_collect(f);

    // p is the tgba view of n
    // the atomic propositions of sap will be registered in dict by the
    // constructor
    pn_tgba p(n, sap, &dict);

    // a is the tgba view of f
    // a and p share the same dictionary
    spot::tgba* a = spot::ltl_to_tgba_fm(f, &dict);

    // prod is simply the product of a and p
    spot::tgba_product prod(a, &p);

    // construct an emptiness checker for prod
    spot::emptiness_check *ec= new spot::couvreur99_check(&prod);
    // check the emptiness
    spot::emptiness_check_result* res = ec->check();
}
```

```

if (res) {
    std::cout << "an accepting run exists" << std::endl;
    // construct and print a counter-example
    spot::tgba_run* run = res->accepting_run();
    if (run) {
        spot::print_tgba_run(std::cout, &prod, run);
        delete run;
    }
    else {
        std::cout << "an accepting run exists (use option '-e' to print it)"
                   << std::endl;
    }
    delete res;
}
else {
    std::cout << "no accepting run found" << std::endl;
}
// print some statistics
ec->print_stats(std::cout);

delete ec;
delete a;
delete sap;

return;
}

```

Notice that a formula given as a string can be easily translated in a **spot::ltl::formula** object by the function **spot::ltl::parse()**.

Some other examples of Spot usages are presented in the program.

- In `count_markings()`, the function **spot::stats\_reachable()** is called to display some statistics concerning the reachability graph of a Petri net.
- The class **spot::tgba\_reachable\_iterator\_breadth\_first** has been derivated in `marking_graph_visitor`. This last class implements a visitor displaying the reachability graph of a Petri net and its usage is illustrated in `print_reachability_graph()`.

## 1.5 Performance Issues

To evaluate the overhead induced by the Spot library, we have developped a simple tool based on the three basic Petri net classes (`marking`, `compressed_marking` and `petri_net`) presented above. This last tool, called `rg`, visits all the reachable markings of a given Petri net and stores them in a hash table.

The performances of `rg` have been compared with the ones of `checkpn` using the option `-c` (the reachability graph is traversed and the number of states and edges are computed).

The table 1.5 resumes these experimentations.

The overhead induces by Spot seems to be very important (more than 30% with respect to `rg` for at least one example). Independently of the indirection implied by the derivation of abstract classes, the two implementations differ by:

- The expansion of compressed marking (`compressed_marking::expand()`) is necessary in `checkpn` due to the interface of **spot::tgba**. Indeed, the method **spot::tgba::succ\_iter()** takes a **spot::state\*** as parameter. Because states are stored in a compressed form, we have to expand them for the computation of the firable transitions. In `rg`, this step is avoided. The stack contains only ordinary markings and they are compressed before to be stored in the hash table.

Model	# states	# edges	Time for checkpn (sec.)	Time for rg (sec.)	Cost of spot (%)
Fms2	3444	16311	0,06	0,05	16,66
Fms3	48590	297382	1,13	0,97	14,15
Fms4	438600	3166985	15,54	13,24	14,8
Kanban2	4600	28120	0,07	0,05	28,57
Kanban3	58400	446400	1,19	0,97	18,48
Kanban4	454475	3979850	11,22	9,43	15,95
Philo5	1364	6375	0,03	0,02	33,33
Philo6	5778	32406	0,16	0,13	18,75
Philo7	24476	160153	1,25	0,98	21,6
Philo8	103682	775336	13,16	9,69	26,36
Philo9	439204	3694923	172,19	116,58	32,29
Ring5	53856	263040	2,18	1,8	17,43
Ring6	575296	3341568	102,95	82,94	19,43
Mutex8	17497	81664	0,57	0,44	22,80
Mutex10	196831	1181000	27,36	19,18	29,89
Nb-philos20	15127	167240	5,35	3,99	25,42
Nb-philos25	167761	2318400	562,12	426,6	24,1

Table 1: Evaluation of the overhead induced by checkpn

- The interface of the class `spot::state` implied the implementation of a method `spot::state::compare()`. This method has to define a total order relation on states. In `rg`, such a method is not necessary. STL hash tables only impose to define a key equality function. We can alleged that such a function is less expansive that the previous one.

This is only a first conclusion, the investigation is in progress...

## 2 checkpn Hierarchical Index

### 2.1 checkpn Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

<b>compressed_marking</b>	<b>8</b>
<b>marking</b>	<b>9</b>
<b>petri_net</b>	<b>14</b>
spot::state[external]	
<b>pn_state</b>	<b>20</b>
spot::tgba[external]	
<b>pn_tgba</b>	<b>29</b>
spot::tgba_reachable_iterator[external]	
spot::tgba_reachable_iterator_breadth_first[external]	
<b>marking_graph_visitor</b>	<b>12</b>
spot::tgba_succ_iterator[external]	

<a href="#">pn_succ_iterator</a>	24
----------------------------------	----

## 3 checkpn Class Index

### 3.1 checkpn Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">compressed_marking</a> (Implementation of a compressed ordinary Petri net state (a bit vector) )	8
<a href="#">marking</a> (Implementation of an ordinary Petri net state (an int vector) )	9
<a href="#">marking_graph_visitor</a> (Implementation of a <code>spot::tgba_reachable_iterator_breadth_first</code> printing all reachable markings of a <a href="#">pn_tgba</a> as well as the firings )	12
<a href="#">petri_net</a> (A simple Petri net class )	14
<a href="#">pn_state</a> (Encapsulation of a <a href="#">compressed_marking</a> in a <code>spot::state</code> )	20
<a href="#">pn_succ_iterator</a> (Implementation of a <code>spot::tgba_succ_iterator</code> for a <a href="#">marking</a> )	24
<a href="#">pn_tgba</a> (Encapsulation of a <a href="#">petri_net</a> in a <code>spot::tgba</code> )	29

## 4 checkpn File Index

### 4.1 checkpn File List

Here is a list of all files with brief descriptions:

<a href="#">compressedmarking.hh</a>	35
<a href="#">graphvisitor.hh</a>	35
<a href="#">mainpage.dox</a>	36
<a href="#">marking.hh</a>	36
<a href="#">petrinet.hh</a>	36
<a href="#">pnstate.hh</a>	36
<a href="#">pnsucciter.hh</a>	36
<a href="#">pntgba.hh</a>	36
<a href="#">pntgbautils.hh</a>	36

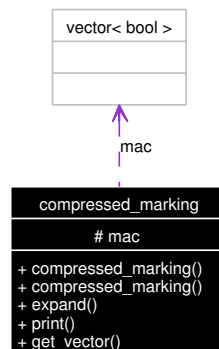


## 5 checkpn Class Documentation

### 5.1 compressed\_marking Class Reference

Implementation of a compressed ordinary Petri net state (a bit vector).

Collaboration diagram for compressed\_marking:



#### Public Member Functions

- `compressed_marking` (const `compressed_marking` &m)
- `compressed_marking` (const `marking` &m)  
*Construct a compression of m.*
- `marking * expand ()` const  
*Return the corresponding expanded marking.*
- void `print` (std::ostream &os) const  
*Basic printing method (as a vector of bits).*
- const std::vector< bool > & `get_vector ()` const  
*Return the std::vector<bool> coding the marking.*

#### Protected Attributes

- std::vector< bool > `mac`

#### 5.1.1 Detailed Description

Implementation of a compressed ordinary Petri net state (a bit vector).

Definition at line 32 of file `compressedmarking.hh`.

#### 5.1.2 Constructor & Destructor Documentation

##### 5.1.2.1 `compressed_marking::compressed_marking` (const `compressed_marking` &m)

**5.1.2.2 compressed\_marking::compressed\_marking (const marking & m)**

Construct a compression of *m*.

**5.1.3 Member Function Documentation****5.1.3.1 marking\* compressed\_marking::expand () const**

Return the corresponding expanded marking.

**5.1.3.2 const std::vector<bool>& compressed\_marking::get\_vector () const**

Return the std::vector<bool> coding the marking.

**5.1.3.3 void compressed\_marking::print (std::ostream & os) const**

Basic printing method (as a vector of bits).

**5.1.4 Member Data Documentation****5.1.4.1 std::vector<bool> compressed\_marking::mac [protected]**

Definition at line 49 of file compressedmarking.hh.

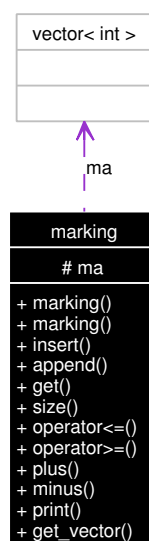
The documentation for this class was generated from the following file:

- [compressedmarking.hh](#)

**5.2 marking Class Reference**

Implementation of an ordinary Petri net state (an int vector).

Collaboration diagram for marking:



**Public Member Functions**

- **marking** (const **marking** &m)
- **marking** (int size=0)  
*Construct a null marking of size size.*
- void **insert** (int p, int n)  
*Set n tokens in place p.*
- void **append** (int n)  
*Set n tokens in a new place. This place takes **size()** as index .*
- int **get** (int p) const  
*Return the number of tokens in place p.*
- int **size** () const  
*Return the number of places on which the marking is defined.*
- bool **operator<=** (const **marking** &m) const  
*Return true if each place of the current marking contains as many or less tokens than m and false otherwise.*
- bool **operator>=** (const **marking** &m) const  
*Return true if each place of the current marking contains as many or more tokens than m and false otherwise.*
- void **plus** (const **marking** &m, **marking** &res) const  
*res recieves \*this + m.*
- void **minus** (const **marking** &m, **marking** &res) const  
*res recieves \*this - m.*
- void **print** (std::ostream &os) const  
*Basic printing method (as a vector of integers).*
- const std::vector< int > & **get\_vector** () const  
*Return the std::vector<int> coding the marking.*

**Protected Attributes**

- std::vector< int > **ma**

**5.2.1 Detailed Description**

Implementation of an ordinary Petri net state (an int vector).

Definition at line 30 of file marking.hh.

**5.2.2 Constructor & Destructor Documentation****5.2.2.1 marking::marking (const **marking** & m)**

### 5.2.2.2 marking::marking (int *size* = 0)

Construct a null marking of size *size*.

## 5.2.3 Member Function Documentation

### 5.2.3.1 void marking::append (int *n*)

Set *n* tokens in a new place. This place takes `size()` as index .

### 5.2.3.2 int marking::get (int *p*) const

Return the number of tokens in place *p*.

#### Precondition:

*p* must be a valid place identifier.

### 5.2.3.3 const std::vector<int>& marking::get\_vector () const

Return the std::vector<int> coding the marking.

### 5.2.3.4 void marking::insert (int *p*, int *n*)

Set *n* tokens in place *p*.

### 5.2.3.5 void marking::minus (const marking & *m*, marking & *res*) const

*res* receives *\*this* - *m*.

#### Precondition:

*\*this* must be equal or greater than *m*.

### 5.2.3.6 bool marking::operator<= (const marking & *m*) const

Return true if each place of the current marking contains as many or less tokens than *m* and false otherwise.

### 5.2.3.7 bool marking::operator>= (const marking & *m*) const

Return true if each place of the current marking contains as many or more tokens than *m* and false otherwise.

### 5.2.3.8 void marking::plus (const marking & *m*, marking & *res*) const

*res* receives *\*this* + *m*.

### 5.2.3.9 void marking::print (std::ostream & *os*) const

Basic printing method (as a vector of integers).

### 5.2.3.10 int marking::size () const

Return the number of places on which the marking is defined.

### 5.2.4 Member Data Documentation

#### 5.2.4.1 `std::vector<int>` `marking::ma` [protected]

Definition at line 74 of file marking.hh.

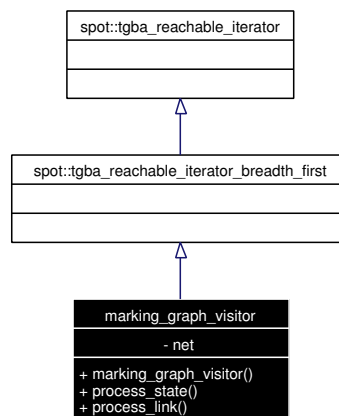
The documentation for this class was generated from the following file:

- [marking.hh](#)

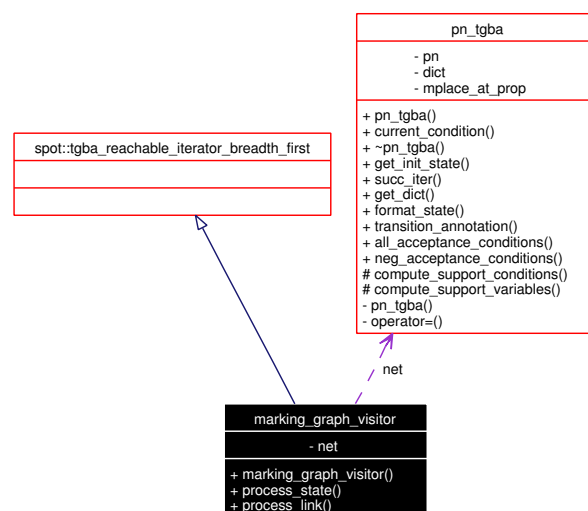
## 5.3 marking\_graph\_visitor Class Reference

Implementation of a `spot::tgba_reachable_iterator_breadth_first` printing all reachable markings of a [pn\\_tgba](#) as well as the firings.

Inheritance diagram for marking\_graph\_visitor:



Collaboration diagram for marking\_graph\_visitor:



**Public Member Functions**

- **marking\_graph\_visitor** (const **pn\_tgba** \*n)  
*Construct a visitor for the Petri net associated to n.*
- void **process\_state** (const **spot::state** \*s, int n, **spot::tgba\_succ\_iterator** \*si)  
*Called by **spot::tgba\_reachable\_iterator\_breadth\_first::run()** to process a state. The corresponding marking is printed.*
- void **process\_link** (int in, int out, const **spot::tgba\_succ\_iterator** \*si)  
*Called by **spot::tgba\_reachable\_iterator\_breadth\_first::run()** to process a fired transition. The corresponding edge is printed.*

**Private Attributes**

- const **pn\_tgba** \* **net**  
*Point to the visited **pn\_tgba**.*

**5.3.1 Detailed Description**

Implementation of a **spot::tgba\_reachable\_iterator\_breadth\_first** printing all reachable markings of a **pn\_tgba** as well as the firings.

Definition at line 33 of file graphvisitor.hh.

**5.3.2 Constructor & Destructor Documentation****5.3.2.1 marking\_graph\_visitor::marking\_graph\_visitor (const **pn\_tgba** \* n)**

Construct a visitor for the Petri net associated to *n*.

The pointer *n* is supposed to be valid for the whole life of the constructed instance.

```
marking_graph_visitor::marking_graph_visitor(const pn_tgba* n)
: spot::tgba_reachable_iterator_breadth_first(n), net(n) {
} //
```

**5.3.3 Member Function Documentation****5.3.3.1 void marking\_graph\_visitor::process\_link (int *in*, int *out*, const **spot::tgba\_succ\_iterator** \* *si*)**

Called by **spot::tgba\_reachable\_iterator\_breadth\_first::run()** to process a fired transition. The corresponding edge is printed.

**Parameters:**

- in* The source state number.
- out* The destination state number.
- si* The **pn\_succ\_iterator** positionned on the current transition.

```
void marking_graph_visitor::process_link(int in, int out, const spot::tgba_succ_iterator* si) {
    const pn_succ_iterator* i = dynamic_cast<const pn_succ_iterator*>(si);
    assert(i);
    std::cout << "\tfiring " << i->format_transition() << " from ";
    std::cout << in << " leads to " << out << std::endl;
} //
```

### 5.3.3.2 void marking\_graph\_visitor::process\_state (const spot::state \* *s*, int *n*, spot::tgba\_succ\_iterator \* *si*)

Called by `spot::tgba_reachable_iterator_breadth_first::run()` to process a state. The corresponding marking is printed.

#### Parameters:

- s* The current state.
- n* An unique number assigned to *s*.
- si* The `pn_succ_iterator` computed for *s*.

```
void marking_graph_visitor::process_state(const spot::state* s, int n, spot::tgba_succ_iterator* si) {
    std::cout << "marking " << n << " : " << net->format_state(s) << std::endl;
} //
```

### 5.3.4 Member Data Documentation

#### 5.3.4.1 const `pn_tgba*` `marking_graph_visitor::net` [private]

Point to the visited `pn_tgba`.

Definition at line 73 of file `graphvisitor.hh`.

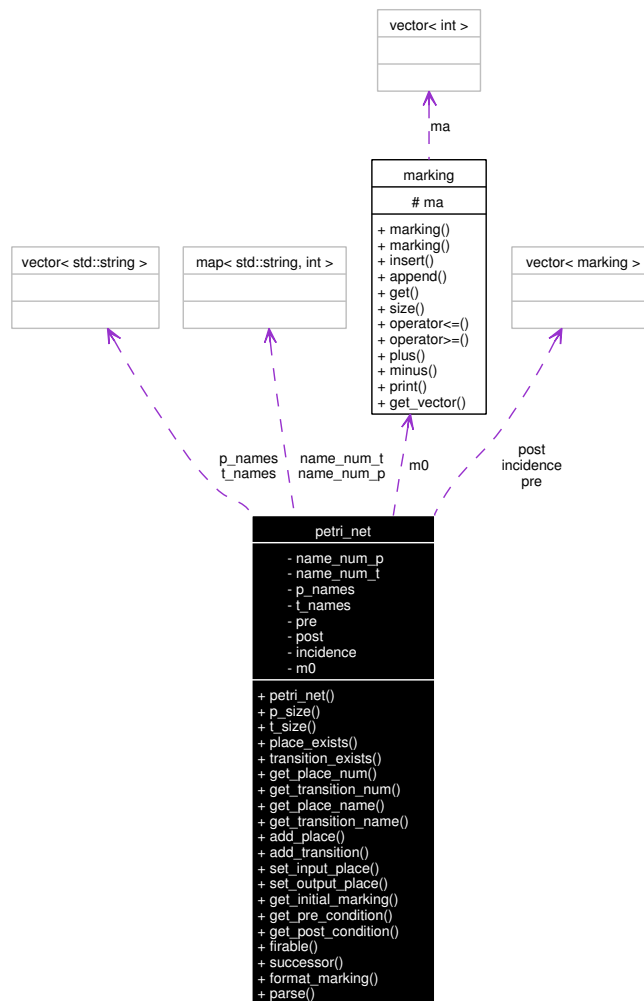
The documentation for this class was generated from the following file:

- `graphvisitor.hh`

## 5.4 petri\_net Class Reference

A simple Petri net class.

Collaboration diagram for `petri_net`:



## Public Member Functions

- `petri_net ()`  
Construct an empty petri net.
- `int p_size () const`  
Return the number of places in the net.
- `int t_size () const`  
Return the number of transitions in the net.
- `bool place_exists (const std::string &name) const`  
Return true if a place of name name is present in the net.
- `bool transition_exists (const std::string &name) const`  
Return true if a transition of name name is present in the net.
- `int get_place_num (const std::string &name) const`



*Return the unique identifier of the place name.*

- `int get_transition_num (const std::string &name) const`  
*Return the unique identifier of the transition name.*
- `const std::string & get_place_name (int p) const`  
*Return the name of the place identified by p.*
- `const std::string & get_transition_name (int t) const`  
*Return the name of the transition identified by t.*
- `void add_place (const std::string &name, int initial_marking=0)`  
*Create a new place of name name and of initial marking initial\_marking.*
- `void add_transition (const std::string &name)`  
*Create a new transition of name name.*
- `void set_input_place (const std::string &place_name, const std::string &trans_name, int valuation=1)`  
*Define the place place\_name as an input of the transition transition\_name with the valuation valuation.*
- `void set_output_place (const std::string &place_name, const std::string &trans_name, int valuation=1)`  
*Define the place place\_name as an output of the transition transition\_name with the valuation valuation.*
- `marking * get_initial_marking () const`  
*Get the initial marking of the Petri net.*
- `const marking & get_pre_condition (int t) const`  
*Get the precondition of t.*
- `const marking & get_post_condition (int t) const`  
*Get the postcondition of t.*
- `std::list< int > * firable (const marking &m) const`  
*Return the list of identifiers of the transitions which are firable from the marking m.*
- `marking * successor (const marking &m, int t) const`  
*Return the marking reached when firing the transition identified by t from the marking m.*
- `std::string format_marking (const marking &m) const`  
*Format the marking m as a string for printing.*

#### Static Public Member Functions

- `petri_net * parse (const char *file_name)`  
*Create a petri net from the file file\_name.*

### Private Attributes

- `std::map< std::string, int > name_num_p`  
*map between the names of the places and their integer numbers.*
- `std::map< std::string, int > name_num_t`  
*map between the names of the transitions and their integer numbers.*
- `std::vector< std::string > p_names`  
*names of the places.*
- `std::vector< std::string > t_names`  
*names of the transitions.*
- `std::vector< marking > pre`  
*pre, post and incidence matrix.*
- `std::vector< marking > post`  
*pre, post and incidence matrix.*
- `std::vector< marking > incidence`  
*pre, post and incidence matrix.*
- `marking m0`  
*initial marking.*

### Friends

- `std::ostream & operator<< (std::ostream &, const petri_net &)`  
*Print the Petri net in 'prod' format.*

#### 5.4.1 Detailed Description

A simple Petri net class.

Definition at line 35 of file petrinet.hh.

#### 5.4.2 Constructor & Destructor Documentation

##### 5.4.2.1 petri\_net::petri\_net ()

Construct an empty petri net.

#### 5.4.3 Member Function Documentation

##### 5.4.3.1 void petri\_net::add\_place (const std::string & name, int initial\_marking = 0)

Create a new place of name *name* and of initial marking *initial\_marking*.

The place name must be distinct of the existing ones and the initial marking be positive.

**5.4.3.2 void petri\_net::add\_transition (const std::string & name)**

Create a new transition of name *name*.

The place name must be distinct of the existing ones.

**5.4.3.3 std::list<int>\* petri\_net::firable (const marking & m) const**

Return the list of identifiers of the transitions which are firable from the marking *m*.

The list has been allocated with `new`. It is the responsibility of the caller to `delete` it when no longer needed.

**5.4.3.4 std::string petri\_net::format\_marking (const marking & m) const**

Format the marking *m* as a string for printing.

**5.4.3.5 marking\* petri\_net::get\_initial\_marking () const**

Get the initial marking of the Petri net.

The marking has been allocated with `new`. It is the responsibility of the caller to `delete` it when no longer needed.

**5.4.3.6 const std::string& petri\_net::get\_place\_name (int p) const**

Return the name of the place identified by *p*.

*p* must be a valid identifier.

**5.4.3.7 int petri\_net::get\_place\_num (const std::string & name) const**

Return the unique identifier of the place *name*.

The place must exist.

**5.4.3.8 const marking& petri\_net::get\_post\_condition (int t) const**

Get the postcondition of *t*.

*t* must be a valid identifier.

**5.4.3.9 const marking& petri\_net::get\_pre\_condition (int t) const**

Get the precondition of *t*.

*t* must be a valid identifier.

**5.4.3.10 const std::string& petri\_net::get\_transition\_name (int t) const**

Return the name of the transition identified by *t*.

*t* must be a valid identifier.

**5.4.3.11 int petri\_net::get\_transition\_num (const std::string & name) const**

Return the unique identifier of the transition *name*.

The place must exist.

#### 5.4.3.12 int petri\_net::p\_size () const

Return the number of places in the net.

#### 5.4.3.13 petri\_net\* petri\_net::parse (const char \*file\_name) [static]

Create a petri net from the file *file\_name*.

Return a null pointer in case of error.

#### 5.4.3.14 bool petri\_net::place\_exists (const std::string &name) const

Return true if a place of name *name* is present in the net.

#### 5.4.3.15 void petri\_net::set\_input\_place (const std::string &place\_name, const std::string &trans\_name, int valuation = 1)

Define the place *place\_name* as an input of the transition *transition\_name* with the valuation *valuation*.

The place and the transition must exist and the valuation be positive.

#### 5.4.3.16 void petri\_net::set\_output\_place (const std::string &place\_name, const std::string &trans\_name, int valuation = 1)

Define the place *place\_name* as an output of the transition *transition\_name* with the valuation *valuation*.

The place and the transition must exist and the valuation be positive.

#### 5.4.3.17 marking\* petri\_net::successor (const marking &m, int t) const

Return the marking reached when firing the transition identified by *t* from the marking *m*.

*t* must be firable from *m*. The marking has been allocated with `new`. It is the responsibility of the caller to `delete` it when no longer needed.

#### 5.4.3.18 int petri\_net::t\_size () const

Return the number of transitions in the net.

#### 5.4.3.19 bool petri\_net::transition\_exists (const std::string &name) const

Return true if a transition of name *name* is present in the net.

### 5.4.4 Friends And Related Function Documentation

#### 5.4.4.1 std::ostream& operator<< (std::ostream &, const petri\_net &) [friend]

Print the Petri net in 'prod' format.

### 5.4.5 Member Data Documentation

#### 5.4.5.1 `std::vector<marking> petri_net::incidence` [private]

pre, post and incidence matrix.

Definition at line 156 of file `petrinet.hh`.

#### 5.4.5.2 `marking petri_net::m0` [private]

initial marking.

Definition at line 158 of file `petrinet.hh`.

#### 5.4.5.3 `std::map<std::string, int> petri_net::name_num_p` [private]

map between the names of the places and their integer numbers.

Definition at line 148 of file `petrinet.hh`.

#### 5.4.5.4 `std::map<std::string, int> petri_net::name_num_t` [private]

map between the names of the transitions and their integer numbers.

Definition at line 150 of file `petrinet.hh`.

#### 5.4.5.5 `std::vector<std::string> petri_net::p_names` [private]

names of the places.

Definition at line 152 of file `petrinet.hh`.

#### 5.4.5.6 `std::vector<marking> petri_net::post` [private]

pre, post and incidence matrix.

Definition at line 156 of file `petrinet.hh`.

#### 5.4.5.7 `std::vector<marking> petri_net::pre` [private]

pre, post and incidence matrix.

Definition at line 156 of file `petrinet.hh`.

#### 5.4.5.8 `std::vector<std::string> petri_net::t_names` [private]

names of the transitions.

Definition at line 154 of file `petrinet.hh`.

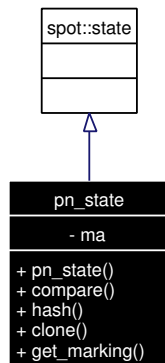
The documentation for this class was generated from the following file:

- [petrinet.hh](#)

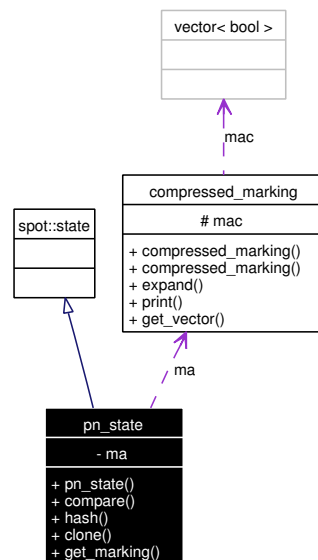
## 5.5 pn\_state Class Reference

Encapsulation of a [compressed\\_marking](#) in a `spot::state`.

Inheritance diagram for pn\_state:



Collaboration diagram for pn\_state:



## Public Member Functions

- `pn_state` (const marking &m)  
*Construct a pn\_state encapsulating the marking m.*
- int `compare` (const state \*other) const  
*Compare two states (issued of a same Petri net).*
- size\_t `hash` () const  
*Hash a state.*
- state \* `clone` () const  
*Duplicate a state.*

- `const compressed_marking & get_marking () const`  
*Return the encapsulated compressed marking.*

### Private Attributes

- `compressed_marking ma`  
*The encapsulated marking is stored compressed.*

### 5.5.1 Detailed Description

Encapsulation of a `compressed_marking` in a `spot::state`.

Definition at line 31 of file `pnstate.hh`.

### 5.5.2 Constructor & Destructor Documentation

#### 5.5.2.1 pn\_state::pn\_state (const marking & m)

Construct a `pn_state` encapsulating the marking `m`.

```
pn_state::pn_state(const marking& m) : spot::state(), ma(m) {
} //
```

### 5.5.3 Member Function Documentation

#### 5.5.3.1 state\* pn\_state::clone () const [virtual]

Duplicate a state.

```
spot::state* pn_state::clone() const {
    pn_state *m = new pn_state(*this);
    return m;
} //
```

Implements `spot::state`.

#### 5.5.3.2 int pn\_state::compare (const state \* other) const [virtual]

Compare two states (issued of a same Petri net).

This method returns an integer less than, equal to, or greater than zero if *this* is found, respectively, to be less than, equal to, or greater than *other* according to bit vector lexicographic order.

This method should not be called to compare states from different automata.

```
int pn_state::compare(const state* other) const {
    const pn_state* m = dynamic_cast<const pn_state*>(other);
    assert(m);

    const std::vector<bool>& b1 = ma.get_vector();
    const std::vector<bool>& b2 = m->ma.get_vector();
    if (b1.size() < b2.size())
```

```

        return -1;
    if (b1.size() > b2.size())
        return 1;

    std::pair<std::vector<bool>::const_iterator,
std::vector<bool>::const_iterator>
        res = std::mismatch(b1.begin(), b1.end(), b2.begin());

    if (res.first==b1.end())
        return 0;
    if (*(res.first) && !*(res.second))
        return 1;
    return -1;
} //

```

Implements `spot::state`.

### 5.5.3.3 `const compressed_marking& pn_state::get_marking() const`

Return the encapsulated compressed marking.

```

const compressed_marking& pn_state::get_marking() const {
    return ma;
} //

```

### 5.5.3.4 `size_t pn_state::hash() const [virtual]`

Hash a state.

```

size_t pn_state::hash() const {
    const int max = sizeof(size_t)*8;
    size_t res = 0;
    const std::vector<bool>& b = ma.get_vector();
    int min = b.size();

    if (min>max)
        min=max;
    std::vector<bool>::const_iterator i = b.begin();

    for (; min>0; --min, ++i)
        if (*i)
            res |= 1<<min;
    return res;
} //

```

Implements `spot::state`.

## 5.5.4 Member Data Documentation

### 5.5.4.1 `compressed_marking pn_state::ma [private]`

The encapsulated marking is stored compressed.

Definition at line 78 of file `pnstate.hh`.

The documentation for this class was generated from the following file:

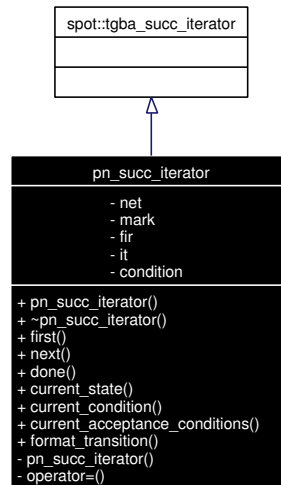
- [pnstate.hh](#)



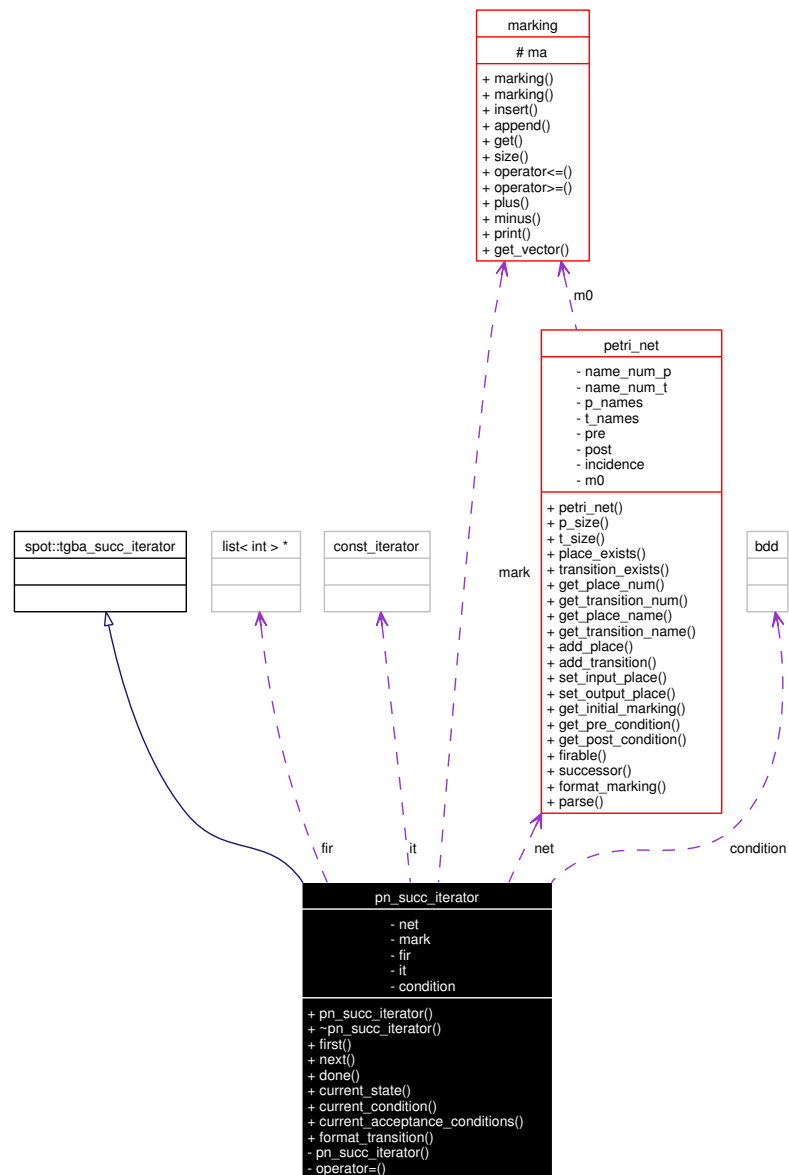
## 5.6 pn\_succ\_iterator Class Reference

Implementation of a `spot::tgba_succ_iterator` for a *marking*.

Inheritance diagram for `pn_succ_iterator`:



Collaboration diagram for `pn_succ_iterator`:



## Public Member Functions

- `pn_succ_iterator` (const `petri_net` &n, const `marking` \*m, const `bdd` &cond)

Construct an iterator over the successors of the marking m in the Petri net n. The marking m satisfies the atomic propositions represented by cond.

- virtual `~pn_succ_iterator` ()

Destructor.

- void `first` ()

Position the iterator on the first successor (if any).

- void `next` ()

*Jump to the next successor (if any).*

- bool `done` () const  
*Check whether the iteration is finished.*
- `spot::state * current_state` () const  
*Get the state of the current successor.*
- bdd `current_condition` () const  
*Get the condition on the transition leading to this successor. This model checker is base on state properties and then all the outgoing arcs are labelled by the conditions of the source state.*
- bdd `current_acceptance_conditions` () const  
*Get the acceptance conditions on the transition leading to this successor. Here no acceptance conditions are taken into account. The method returns false.*
- std::string `format_transition` () const  
*Return the transition name of the current successor for printing.*

### Private Member Functions

- `pn_succ_iterator` (const `pn_succ_iterator` &s)  
*not implemented (assert(false))*
- `pn_succ_iterator & operator=` (const `pn_succ_iterator` &s)  
*not implemented (assert(false))*

### Private Attributes

- const `petri_net` & `net`  
*Designate the associated Petri net.*
- const `marking` \* `mark`  
*Point to the marking for which we iterate.*
- std::list< int > \* `fir`  
*Point to the list of transitions firable from \*mark in net.*
- std::list< int >::const\_iterator `it`  
*Designate the current successor.*
- bdd `condition`  
*Condition satisfied by \*mark.*

### 5.6.1 Detailed Description

Implementation of a `spot::tgba_succ_iterator` for a *marking*.

Definition at line 36 of file `pnsucciter.hh`.

### 5.6.2 Constructor & Destructor Documentation

#### 5.6.2.1 pn\_succ\_iterator::pn\_succ\_iterator (const petri\_net & n, const marking \* m, const bdd & cond)

Construct an iterator over the successors of the marking *m* in the Petri net *n*. The marking *m* satisfies the atomic propositions represented by *cond*.

Notice that the constructed instance takes the control of the marking pointed by *m*. In particular, this marking will be deallocated automatically at the destruction.

```
pn_succ_iterator::pn_succ_iterator(const petri_net& n,
                                   const marking* m, const bdd& cond)
: net(n), mark(m), fir(n.firable(*m)), condition(cond) {
    it = fir->end(); // => done()
} //
```

#### 5.6.2.2 virtual pn\_succ\_iterator::~~pn\_succ\_iterator () [virtual]

Destructor.

```
pn_succ_iterator::~~pn_succ_iterator() {
    delete mark;
    delete fir;
} //
```

#### 5.6.2.3 pn\_succ\_iterator::pn\_succ\_iterator (const pn\_succ\_iterator & s) [private]

not implemented (`assert(false)`)

### 5.6.3 Member Function Documentation

#### 5.6.3.1 bdd pn\_succ\_iterator::current\_acceptance\_conditions () const [virtual]

Get the acceptance conditions on the transition leading to this successor. Here no acceptance conditions are taken into account. The method returns false.

Implements `spot::tgba_succ_iterator`.

#### 5.6.3.2 bdd pn\_succ\_iterator::current\_condition () const [virtual]

Get the condition on the transition leading to this successor. This model checker is base on state properties and then all the outgoing arcs are labelled by the conditions of the source state.

```
bdd pn_succ_iterator::current_condition() const {
    assert(!done());
    return condition;
} //
```

Implements `spot::tgba_succ_iterator`.

**5.6.3.3 spot::state\* pn\_succ\_iterator::current\_state () const [virtual]**

Get the state of the current successor.

The state has been allocated with new. It is the responsibility of the caller to delete it when no longer needed.

```
spot::state* pn_succ_iterator::current_state() const {
    assert(!done());
    marking* m = net.successor(*mark, *it);
    pn_state* res = new pn_state(*m);
    delete m;
    return res;
} //
```

Implements **spot::tgba\_succ\_iterator**.

**5.6.3.4 bool pn\_succ\_iterator::done () const [virtual]**

Check whether the iteration is finished.

```
bool pn_succ_iterator::done() const {
    return it==fir->end();
} //
```

Implements **spot::tgba\_succ\_iterator**.

**5.6.3.5 void pn\_succ\_iterator::first () [virtual]**

Position the iterator on the first successor (if any).

```
void pn_succ_iterator::first() {
    it = fir->begin();
} //
```

Implements **spot::tgba\_succ\_iterator**.

**5.6.3.6 std::string pn\_succ\_iterator::format\_transition () const**

Return the transition name of the current successor for printing.

```
std::string pn_succ_iterator::format_transition() const {
    assert(!done());
    return net.get_transition_name(*it);
} //
```

**5.6.3.7 void pn\_succ\_iterator::next () [virtual]**

Jump to the next successor (if any).

```
void pn_succ_iterator::next() {
    assert(!done());
    it++;
} //
```

Implements **spot::tgba\_succ\_iterator**.

**5.6.3.8** `pn_succ_iterator& pn_succ_iterator::operator= (const pn_succ_iterator & s)`  
[private]

not implemented (assert(false))

#### 5.6.4 Member Data Documentation

**5.6.4.1** `bdd pn_succ_iterator::condition` [private]

Condition satisfied by *mark*.

Definition at line 131 of file pnsucciter.hh.

**5.6.4.2** `std::list<int>* pn_succ_iterator::fir` [private]

Point to the list of transitions firable from *mark* in *net*.

Definition at line 125 of file pnsucciter.hh.

**5.6.4.3** `std::list<int>::const_iterator pn_succ_iterator::it` [private]

Designate the current successor.

Definition at line 128 of file pnsucciter.hh.

**5.6.4.4** `const marking* pn_succ_iterator::mark` [private]

Point to the marking for which we iterate.

Definition at line 122 of file pnsucciter.hh.

**5.6.4.5** `const petri_net& pn_succ_iterator::net` [private]

Designate the associated Petri net.

Definition at line 119 of file pnsucciter.hh.

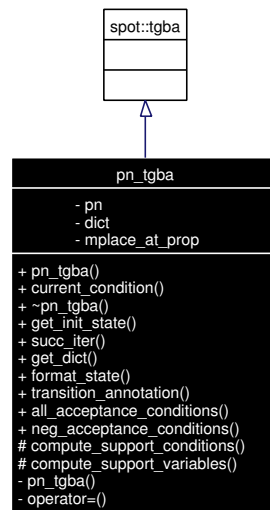
The documentation for this class was generated from the following file:

- [pnsucciter.hh](#)

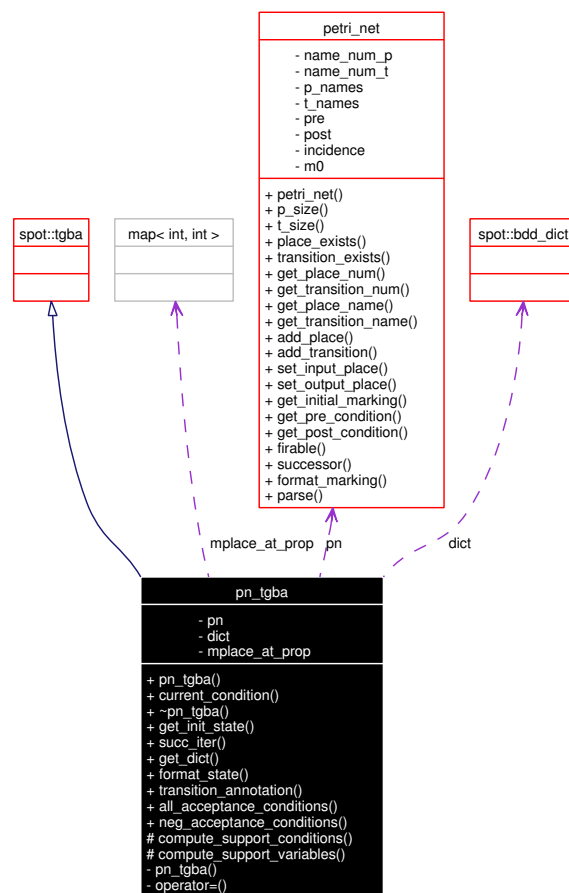
## 5.7 pn\_tgba Class Reference

Encapsulation of a *petri\_net* in a *spot::tgba*.

Inheritance diagram for pn\_tgba:



Collaboration diagram for `pn_tgba`:



## Public Member Functions

- **pn\_tgba** (const **petri\_net** \*pn, const **spot::ltd::atomic\_prop\_set** \*sap=0, **spot::bdd\_dict** \*dict=0)  
*Construct a tgba view of pn where the set of atomic propositions which are observed are in sap. The constructor registers these propositions in dict.*
- **bdd current\_condition** (const **marking** &m) const  
*Return the bdd corresponding to the values of the observed atomic propositions in the marking m.*
- virtual **~pn\_tgba** ()  
*Unregister all the used propositions.*
- **spot::state** \* **get\_init\_state** () const  
*Get the initial state of the automaton.*
- **spot::tgba\_succ\_iterator** \* **succ\_iter** (const **spot::state** \*local\_state, const **spot::state** \*, const **spot::tgba** \*) const  
*Get an iterator over the successors of local\_state.*
- **spot::bdd\_dict** \* **get\_dict** () const  
*Get the dictionary associated to the automaton.*
- std::string **format\_state** (const **spot::state** \*state) const  
*Format the state as a string for printing.*
- std::string **transition\_annotation** (const **spot::tgba\_succ\_iterator** \*t) const  
*Format the pointed transition as a string for printing.*
- **bdd all\_acceptance\_conditions** () const  
*Return the empty set (false) as the Petri net accepts all infinite sequences. Take care that blocking sequences are not taken into account here.*
- **bdd neg\_acceptance\_conditions** () const  
*Return true.*

## Protected Member Functions

- **bdd compute\_support\_conditions** (const **spot::state** \*state) const  
*Do the actual computation of tgba::support\_conditions(). Return true.*
- **bdd compute\_support\_variables** (const **spot::state** \*state) const  
*Do the actual computation of tgba::support\_variables(). Return true.*

## Private Member Functions

- **pn\_tgba** (const **pn\_tgba** &p)  
*not implemented (assert(false))*



- `pn_tgba & operator= (const pn_tgba &p)`  
*not implemented (assert(false))*

### Private Attributes

- `const petri_net & pn`  
*Reference the encapsulated Petri net.*
- `spot::bdd_dict * dict`  
*Point to the associated dictionnary.*
- `std::map< int, int > mplace_at_prop`

### 5.7.1 Detailed Description

Encapsulation of a `petri_net` in a `spot::tgba`.

Definition at line 43 of file `pntgba.hh`.

### 5.7.2 Constructor & Destructor Documentation

#### 5.7.2.1 `pn_tgba::pn_tgba (const petri_net * pn, const spot::ltl::atomic_prop_set * sap = 0, spot::bdd_dict * dict = 0)`

Construct a tgba view of `pn` where the set of atomic propositions which are observed are in `sap`. The constructor registers these propositions in `dict`.

`sap` can be a null pointer if the tgba will not be used for checking. In this case, `dict` can also be a null pointer. The pointers `pn` and `dict` (if not null) are supposed to be valide during all the live of the constructed instance.

```
pn_tgba::pn_tgba(const petri_net* p,
                const spot::ltl::atomic_prop_set* sap,
                spot::bdd_dict* dic) : pn(*p) {
    assert(!sap || dic);

    dict = dic;
    if (sap) {
        spot::ltl::atomic_prop_set::iterator it;
        for(it=sap->begin(); it!=sap->end(); ++it) {
            mplace_at_prop[pn.get_place_num((*it)->name())] =
                dict->register_proposition(*it, this);
        }
    }
} //
```

#### 5.7.2.2 `virtual pn_tgba::~~pn_tgba () [virtual]`

Unregister all the used propositions.

```
pn_tgba::~~pn_tgba() {
    if (dict)
        dict->unregister_all_my_variables(this);
} //
```

**5.7.2.3 pn\_tgba::pn\_tgba (const pn\_tgba & p) [private]**

not implemented (assert(false))

**5.7.3 Member Function Documentation****5.7.3.1 bdd pn\_tgba::all\_acceptance\_conditions () const [virtual]**

Return the empty set (false) as the Petri net accepts all infinite sequences. Take care that blocking sequences are not taken into account here.

Implements **spot::tgba**.

**5.7.3.2 bdd pn\_tgba::compute\_support\_conditions (const spot::state \* state) const [protected]**

Do the actual computation of tgba::support\_conditions(). Return true.

**5.7.3.3 bdd pn\_tgba::compute\_support\_variables (const spot::state \* state) const [protected]**

Do the actual computation of tgba::support\_variables(). Return true.

**5.7.3.4 bdd pn\_tgba::current\_condition (const marking & m) const**

Return the bdd corresponding to the values of the observed atomic propositions in the marking *m*.

```
bdd pn_tgba::current_condition(const marking& m) const {
    bdd res = bddtrue;
    std::map<int, int>::const_iterator it;
    for(it=mplace_at_prop.begin(); it!=mplace_at_prop.end(); it++)
        if (m.get(it->first) > 0)
            res &= bdd_ithvar(it->second);
        else
            res &= bdd_nithvar(it->second);
    return res;
} //
```

**5.7.3.5 std::string pn\_tgba::format\_state (const spot::state \* state) const**

Format the state as a string for printing.

```
std::string pn_tgba::format_state(const spot::state* state) const {
    const pn_state* s = dynamic_cast<const pn_state*>(state);
    assert(s);
    marking* m = s->get_marking().expand();
    std::string res = pn.format_marking(*m);
    delete m;
    return res;
} //
```

**5.7.3.6 spot::bdd\_dict\* pn\_tgba::get\_dict () const [virtual]**

Get the dictionary associated to the automaton.

```
spot::bdd_dict* pn_tgba::get_dict() const {
    return dict;
} //
```

Implements **spot::tgba**.

#### 5.7.3.7 spot::state\* pn\_tgba::get\_init\_state() const [virtual]

Get the initial state of the automaton.

The state has been allocated with `new`. It is the responsibility of the caller to delete it when no longer needed.

```
spot::state* pn_tgba::get_init_state() const {
    marking* m0 = pn.get_initial_marking();
    pn_state* res = new pn_state(*m0);
    delete m0;
    return res;
} //
```

Implements **spot::tgba**.

#### 5.7.3.8 bdd pn\_tgba::neg\_acceptance\_conditions() const [virtual]

Return true.

Implements **spot::tgba**.

#### 5.7.3.9 pn\_tgba& pn\_tgba::operator=(const pn\_tgba & p) [private]

not implemented (assert(false))

#### 5.7.3.10 spot::tgba\_succ\_iterator\* pn\_tgba::succ\_iter (const spot::state \* local\_state, const spot::state \*, const spot::tgba \*) const

Get an iterator over the successors of *local\_state*.

The iterator has been allocated with `new`. It is the responsibility of the caller to delete it when no longer needed.

The two last parameters are not used here

```
spot::tgba_succ_iterator* pn_tgba::succ_iter (const spot::state* local_state,
    const spot::state*, const spot::tgba*) const {
    const pn_state* s = dynamic_cast<const pn_state*>(local_state);
    assert(s);
    marking* m = s->get_marking().expand();
    return new pn_succ_iterator(pn, m, current_condition(*m));
} //
```

#### 5.7.3.11 std::string pn\_tgba::transition\_annotation (const spot::tgba\_succ\_iterator \* t) const

Format the pointed transition as a string for printing.

**Parameters:**

*t* a non-done [pn\\_succ\\_iterator](#) for this automata

```
std::string pn_tgba::transition_annotation(const spot::tgba_succ_iterator* t)
const {
    assert(!t->done());
    return (dynamic_cast<const pn_succ_iterator*>(t))->format_transition();
} //
```

## 5.7.4 Member Data Documentation

### 5.7.4.1 `spot::bdd_dict*` `pn_tgba::dict` [private]

Point to the associated dictionary.

Definition at line 150 of file pntgba.hh.

### 5.7.4.2 `std::map<int, int>` `pn_tgba::mplace_at_prop` [private]

Map the indexes of places used as atomic propositions to the corresponding indexes of bdd variables.

Definition at line 154 of file pntgba.hh.

### 5.7.4.3 `const petri_net&` `pn_tgba::pn` [private]

Reference the encapsulated Petri net.

Definition at line 147 of file pntgba.hh.

The documentation for this class was generated from the following file:

- [pntgba.hh](#)

## 6 checkpn File Documentation

### 6.1 compressedmarking.hh File Reference

#### Classes

- class [compressed\\_marking](#)  
*Implementation of a compressed ordinary Petri net state (a bit vector).*

### 6.2 graphvisitor.hh File Reference

#### Classes

- class [marking\\_graph\\_visitor](#)  
*Implementation of a `spot::tgba_reachable_iterator_breadth_first` printing all reachable markings of a `pn_tgba` as well as the firings.*

## 6.3 `mainpage.dox` File Reference

## 6.4 `marking.hh` File Reference

### Classes

- class `marking`  
*Implementation of an ordinary Petri net state (an int vector).*

## 6.5 `petrinet.hh` File Reference

### Classes

- class `petri_net`  
*A simple Petri net class.*

## 6.6 `pnstate.hh` File Reference

### Classes

- class `pn_state`  
*Encapsulation of a `compressed_marking` in a `spot::state`.*

## 6.7 `pnsucciter.hh` File Reference

### Classes

- class `pn_succ_iterator`  
*Implementation of a `spot::tgba_succ_iterator` for a `marking`.*

## 6.8 `pntgba.hh` File Reference

### Classes

- class `pn_tgba`  
*Encapsulation of a `petri_net` in a `spot::tgba`.*

## 6.9 `pntgbautils.hh` File Reference

### Functions

- void `count_markings` (const `petri_net` \*n)  
*Display some statistics concerning the Petri net n.*

- void `print_reachability_graph` (const `petri_net` \*n)  
*Display the reachability graph of the Petri net n.*
- const std::string \* `check_at_prop` (const `petri_net` \*p, const `spot::ltl::formula` \*f)  
*Check if the atomic propositions in f are places of the Petri net p. Return a pointer on the first atomic proposition which is not a place name if any and 0 otherwise.*
- void `model_check` (const `petri_net` \*n, const `spot::ltl::formula` \*f, bool ce\_expected, bool fm\_exprop\_opt=false, bool fm\_symb\_merge\_opt=true, bool post\_branching=false, bool fair\_loop\_approx=false)  
*Check if the Petri net n can produce at least one infinite sequence accepted by the formula f.*

## 6.9.1 Function Documentation

### 6.9.1.1 const std::string\* check\_at\_prop (const `petri_net` \*p, const `spot::ltl::formula` \*f)

Check if the atomic propositions in *f* are places of the Petri net *p*. Return a pointer on the first atomic proposition which is not a place name if any and 0 otherwise.

```
const std::string* check_at_prop(const petri_net* p,
                                const spot::ltl::formula* f) {
    spot::ltl::atomic_prop_set* sap = spot::ltl::atomic_prop_collect(f);

    if (sap) {
        spot::ltl::atomic_prop_set::iterator it;
        for(it=sap->begin(); it!=sap->end(); ++it) {
            if(!p->place_exists( (*it)->name() )) {
                return &((*it)->name());
            }
        }
    }
    return 0;
} //
```

### 6.9.1.2 void count\_markings (const `petri_net` \*n)

Display some statistics concerning the Petri net *n*.

```
void count_markings(const petri_net* n) {
    double start_time, visit_time;

    pn_tgba p(n);

    start_time = ((double)(clock()) / CLOCKS_PER_SEC);

    spot::tgba_statistics st = spot::stats_reachable(&p);

    visit_time = ((double)(clock()) / CLOCKS_PER_SEC) - start_time;

    std::cout << "the state graph is composed of " << st.states;
    std::cout << " states and " << st.transitions << " edges" << std::endl;
    std::cout << "visiting time: " << visit_time << std::endl;
} //
```

**6.9.1.3 void model\_check** (const [petri\\_net](#) \* *n*, const [spot::ltl::formula](#) \* *f*, bool *ce\_expected*, bool *fm\_exprop\_opt* = false, bool *fm\_symb\_merge\_opt* = true, bool *post\_branching* = false, bool *fair\_loop\_approx* = false)

Check if the Petri net *n* can produce at least one infinite sequence accepted by the formula *f*.

If this is the case and *ce\_expected* is true, such a sequence is displayed. In other case, a message indicating if the test is satisfied or not is printed. The four last parameters specify options for the translation of the formula *f* in a TGBA as indicated for the function [spot::ltl\\_to\\_tgba\\_fm](#).

```
void model_check(const petri_net* n, const spot::ltl::formula* f,
                bool ce_expected, bool fm_exprop_opt, bool fm_symb_merge_opt,
                bool post_branching, bool fair_loop_approx) {
    double start_time, build_time, check_time;

    const std::string* s;
    if ((s = check_at_prop(n, f)) {
        std::cout << "the atomic proposition '" << *s
                    << "' does not correspond to a place name" << std::endl;
        return;
    }

    spot::bdd_dict dict;

    spot::ltl::atomic_prop_set* sap = spot::ltl::atomic_prop_collect(f);

    pn_tgba p(n, sap, &dict);

    start_time = (double)(clock()) / CLOCKS_PER_SEC;
    spot::tgba* a = spot::ltl_to_tgba_fm(f, &dict, fm_exprop_opt,
                                         fm_symb_merge_opt, post_branching, fair_loop_approx);

    spot::tgba_product prod(a, &p);

    spot::emptiness_check *ec= new spot::couvreur99_check(&prod);

    build_time = ((double)(clock()) / CLOCKS_PER_SEC) - start_time;
    spot::emptiness_check_result* res = ec->check();
    check_time = ((double)(clock()) / CLOCKS_PER_SEC) - (start_time + build_time);

    ec->print_stats(std::cout);
    std::cout << "automaton construction time: " << build_time << std::endl;
    std::cout << "checking time: " << check_time << std::endl;
    std::cout << std::endl;

    if (res) {
        if (ce_expected) {
            std::cout << "an accepting run exists" << std::endl;
            spot::tgba_run* run = res->accepting_run();
            if (run)
            {
                spot::print_tgba_run(std::cout, &prod, run);
                delete run;
            }
        }
        else {
            std::cout << "an accepting run exists (use option '-e' to print it)"
                        << std::endl;
        }
        delete res;
    }
    else {
        std::cout << "no accepting run found" << std::endl;
    }

    delete ec;
}
```

```
    delete a;  
    delete sap;  
  
    return;  
} //
```

#### 6.9.1.4 void print\_reachability\_graph (const petri\_net \* n)

Display the reachability graph of the Petri net *n*.

```
void print_reachability_graph(const petri_net* n) {  
  
    pn_tgba p(n);  
    marking_graph_visitor mgv(&p);  
    mgv.run();  
} //
```



## Index

- ~pn\_succ\_iterator
  - pn\_succ\_iterator, [27](#)
- ~pn\_tgba
  - pn\_tgba, [32](#)
- add\_place
  - petri\_net, [17](#)
- add\_transition
  - petri\_net, [17](#)
- all\_acceptance\_conditions
  - pn\_tgba, [33](#)
- append
  - marking, [11](#)
- check\_at\_prop
  - pntgbautils.hh, [37](#)
- clone
  - pn\_state, [22](#)
- compare
  - pn\_state, [22](#)
- compressed\_marking, [8](#)
  - compressed\_marking, [8](#)
  - expand, [9](#)
  - get\_vector, [9](#)
  - mac, [9](#)
  - print, [9](#)
- compressedmarking.hh, [35](#)
- compute\_support\_conditions
  - pn\_tgba, [33](#)
- compute\_support\_variables
  - pn\_tgba, [33](#)
- condition
  - pn\_succ\_iterator, [29](#)
- count\_markings
  - pntgbautils.hh, [37](#)
- current\_acceptance\_conditions
  - pn\_succ\_iterator, [27](#)
- current\_condition
  - pn\_succ\_iterator, [27](#)
  - pn\_tgba, [33](#)
- current\_state
  - pn\_succ\_iterator, [27](#)
- dict
  - pn\_tgba, [35](#)
- done
  - pn\_succ\_iterator, [28](#)
- expand
  - compressed\_marking, [9](#)
- fir
  - pn\_succ\_iterator, [29](#)
- firable
  - petri\_net, [18](#)
- first
  - pn\_succ\_iterator, [28](#)
- format\_marking
  - petri\_net, [18](#)
- format\_state
  - pn\_tgba, [33](#)
- format\_transition
  - pn\_succ\_iterator, [28](#)
- get
  - marking, [11](#)
- get\_dict
  - pn\_tgba, [33](#)
- get\_init\_state
  - pn\_tgba, [34](#)
- get\_initial\_marking
  - petri\_net, [18](#)
- get\_marking
  - pn\_state, [23](#)
- get\_place\_name
  - petri\_net, [18](#)
- get\_place\_num
  - petri\_net, [18](#)
- get\_post\_condition
  - petri\_net, [18](#)
- get\_pre\_condition
  - petri\_net, [18](#)
- get\_transition\_name
  - petri\_net, [18](#)
- get\_transition\_num
  - petri\_net, [18](#)
- get\_vector
  - compressed\_marking, [9](#)
  - marking, [11](#)
- graphvisitor.hh, [35](#)
- hash
  - pn\_state, [23](#)
- incidence
  - petri\_net, [20](#)
- insert
  - marking, [11](#)
- it
  - pn\_succ\_iterator, [29](#)
- m0
  - petri\_net, [20](#)

- ma
  - marking, 12
  - pn\_state, 23
- mac
  - compressed\_marking, 9
- mainpage.dox, 36
- mark
  - pn\_succ\_iterator, 29
- marking, 9
  - append, 11
  - get, 11
  - get\_vector, 11
  - insert, 11
  - ma, 12
  - marking, 10
  - minus, 11
  - operator<=, 11
  - operator>=, 11
  - plus, 11
  - print, 11
  - size, 11
- marking.hh, 36
- marking\_graph\_visitor, 12
  - marking\_graph\_visitor, 13
  - net, 14
  - process\_link, 13
  - process\_state, 14
- minus
  - marking, 11
- model\_check
  - pntgbautils.hh, 37
- mplace\_at\_prop
  - pn\_tgba, 35
- name\_num\_p
  - petri\_net, 20
- name\_num\_t
  - petri\_net, 20
- neg\_acceptance\_conditions
  - pn\_tgba, 34
- net
  - marking\_graph\_visitor, 14
  - pn\_succ\_iterator, 29
- next
  - pn\_succ\_iterator, 28
- operator<<
  - petri\_net, 19
- operator<=
  - marking, 11
- operator=
  - pn\_succ\_iterator, 28
  - pn\_tgba, 34
- operator>=
  - marking, 11
- p\_names
  - petri\_net, 20
- p\_size
  - petri\_net, 19
- parse
  - petri\_net, 19
- petri\_net, 14
  - add\_place, 17
  - add\_transition, 17
  - fireable, 18
  - format\_marking, 18
  - get\_initial\_marking, 18
  - get\_place\_name, 18
  - get\_place\_num, 18
  - get\_post\_condition, 18
  - get\_pre\_condition, 18
  - get\_transition\_name, 18
  - get\_transition\_num, 18
  - incidence, 20
  - m0, 20
  - name\_num\_p, 20
  - name\_num\_t, 20
  - operator<<, 19
  - p\_names, 20
  - p\_size, 19
  - parse, 19
  - petri\_net, 17
  - place\_exists, 19
  - post, 20
  - pre, 20
  - set\_input\_place, 19
  - set\_output\_place, 19
  - successor, 19
  - t\_names, 20
  - t\_size, 19
  - transition\_exists, 19
- petrinet.hh, 36
- place\_exists
  - petri\_net, 19
- plus
  - marking, 11
- pn
  - pn\_tgba, 35
- pn\_state, 20
  - clone, 22
  - compare, 22
  - get\_marking, 23
  - hash, 23
  - ma, 23
  - pn\_state, 22
- pn\_succ\_iterator, 24
  - ~pn\_succ\_iterator, 27

- condition, [29](#)
- current\_acceptance\_conditions, [27](#)
- current\_condition, [27](#)
- current\_state, [27](#)
- done, [28](#)
- fir, [29](#)
- first, [28](#)
- format\_transition, [28](#)
- it, [29](#)
- mark, [29](#)
- net, [29](#)
- next, [28](#)
- operator=, [28](#)
- pn\_succ\_iterator, [27](#)
- pn\_tgba, [29](#)
  - ~pn\_tgba, [32](#)
  - all\_acceptance\_conditions, [33](#)
  - compute\_support\_conditions, [33](#)
  - compute\_support\_variables, [33](#)
  - current\_condition, [33](#)
  - dict, [35](#)
  - format\_state, [33](#)
  - get\_dict, [33](#)
  - get\_init\_state, [34](#)
  - mplace\_at\_prop, [35](#)
  - neg\_acceptance\_conditions, [34](#)
  - operator=, [34](#)
  - pn, [35](#)
  - pn\_tgba, [32](#)
  - succ\_iter, [34](#)
  - transition\_annotation, [34](#)
- pnstate.hh, [36](#)
- pnsucciter.hh, [36](#)
- pntgba.hh, [36](#)
- pntgbautils.hh, [36](#)
  - check\_at\_prop, [37](#)
  - count\_markings, [37](#)
  - model\_check, [37](#)
  - print\_reachability\_graph, [39](#)
- post
  - petri\_net, [20](#)
- pre
  - petri\_net, [20](#)
- print
  - compressed\_marking, [9](#)
  - marking, [11](#)
- print\_reachability\_graph
  - pntgbautils.hh, [39](#)
- process\_link
  - marking\_graph\_visitor, [13](#)
- process\_state
  - marking\_graph\_visitor, [14](#)
- set\_input\_place
  - petri\_net, [19](#)
- set\_output\_place
  - petri\_net, [19](#)
- size
  - marking, [11](#)
- succ\_iter
  - pn\_tgba, [34](#)
- successor
  - petri\_net, [19](#)
- t\_names
  - petri\_net, [20](#)
- t\_size
  - petri\_net, [19](#)
- transition\_annotation
  - pn\_tgba, [34](#)
- transition\_exists
  - petri\_net, [19](#)