

# PetriDotNet 1.5

András Vörös	Dániel Darvas	Ákos Hajdu
Attila Jámbor	Attila Klenik	Kristóf Marussy
Vince Molnár	Tamás Bartha	István Majzik*

User Manual

---

\*E-mail of the developer team: [petridotnet@inf.mit.bme.hu](mailto:petridotnet@inf.mit.bme.hu)



# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	System Requirements . . . . .	7
<b>2</b>	<b>Basic Operation</b>	<b>9</b>
2.1	Starting the Tool . . . . .	9
2.2	Load, Save . . . . .	9
2.3	Creating and Editing Petri Nets . . . . .	10
2.4	Editing Hierarchical Petri Nets . . . . .	13
2.5	Coloured Petri Nets . . . . .	16
2.5.1	Editing Coloured Petri Nets . . . . .	16
2.5.2	Handling Colour Sets . . . . .	16
2.5.3	Handling Coloured Tokens . . . . .	20
2.5.4	Edge Expressions, Guards and Variables . . . . .	21
2.5.5	Limitations . . . . .	23
2.6	Petri Net Simulation . . . . .	23
<b>3</b>	<b>CTL Model Checking</b>	<b>27</b>
3.1	Unbounded CTL Model Checking . . . . .	27
3.2	Bounded CTL Model Checking . . . . .	30
<b>4</b>	<b>Stochastic Analysis Module</b>	<b>33</b>
4.1	Performance measures . . . . .	33
4.2	Model and reward settings . . . . .	35
4.2.1	Sensitivity Variables . . . . .	35
4.2.2	Sensitivity Variable Configurations . . . . .	35
4.2.3	Transition Rates . . . . .	36
4.2.4	State Rewards . . . . .	37
4.2.5	Event Rewards . . . . .	39
4.2.6	Composite Rewards . . . . .	41
4.2.7	Reference consistency . . . . .	42
4.3	Formalizing the performance measures . . . . .	42
4.4	Reward evaluation . . . . .	43
4.4.1	Analysis Configuration . . . . .	43
4.4.2	Selecting the rewards . . . . .	46
4.4.3	Running the analysis . . . . .	48
4.5	Mean Time to First Failure calculation . . . . .	48

<b>5</b>	<b>CEGAR-based Reachability Analysis</b>	<b>51</b>
5.1	Overview of the algorithms . . . . .	51
5.1.1	Abstraction . . . . .	51
5.1.2	CEGAR approach on Petri nets . . . . .	51
5.2	Usage . . . . .	53
5.2.1	Overview of the GUI . . . . .	53
5.2.2	Information about the net . . . . .	54
5.2.3	Parameters of the reachability problem . . . . .	55
5.2.4	Configuration of the algorithm . . . . .	56
5.2.5	Examination of the result of the algorithm . . . . .	58
<b>6</b>	<b>Quick Introduction to Plug-in Development</b>	<b>61</b>

# List of Figures

2.1	Empty editor window. . . . .	9
2.2	Main window with a loaded Petri net. . . . .	10
2.3	The toolbox in Design mode. . . . .	11
2.4	Drawing edges between Petri net elements. . . . .	12
2.5	Edit helper tool. . . . .	12
2.6	Properties panel. . . . .	13
2.7	Example Petri net with coarse transition. . . . .	14
2.8	Example subnet. . . . .	14
2.9	The flat Petri net equivalent to Figure 2.7 and 2.8. . . . .	15
2.10	The <i>Coloured token types</i> window. . . . .	16
2.11	The <i>Token type editor</i> window for single token types. . . . .	17
2.12	The <i>Token type editor</i> window filled with example data. . . . .	18
2.13	The <i>Token type editor</i> window for complex token types. . . . .	19
2.14	The <i>Add and remove coloured tokens</i> window. . . . .	20
2.15	The <i>Coloured tokens</i> window. . . . .	21
2.16	The <i>Variables</i> window. . . . .	22
2.17	The <i>Net analysis</i> window. . . . .	23
2.18	Settings of the interactive simulation. . . . .	24
2.19	Settings of the non-interactive simulation. . . . .	25
2.20	Large scale statistics plug-in. . . . .	25
3.1	Form to select decomposition strategy. . . . .	28
3.2	The <i>State space generation finished</i> window. . . . .	29
3.3	CTL expression editor. . . . .	30
3.4	CTL expression editor with an example expression. . . . .	31
3.5	Example CTL model checking result. . . . .	31
3.6	Settings of bounded CTL model checking. . . . .	32
4.1	The example hybrid cloud model. . . . .	34
4.2	A simple load balancer model. . . . .	34
4.3	Sensitivity variables tab. . . . .	35
4.4	Sensitivity variable configurations tab. . . . .	36
4.5	Transition rates tab. . . . .	37
4.6	State rewards tab. . . . .	38
4.7	Place-based reward window. . . . .	38
4.8	CTL editor window. . . . .	39
4.9	Arithmetic expression editor window. . . . .	40
4.10	Event rewards tab. . . . .	40

4.11	Event rewards tab. . . . .	41
4.12	Invalid transition rate warning. . . . .	42
4.13	Analysis configuration form. . . . .	43
4.14	Reward selection form for steady state analysis. . . . .	47
4.15	Reward selection form for transient analysis. . . . .	47
4.16	The running analysis. . . . .	48
4.17	The finished analysis. . . . .	49
5.1	Petri net CEGAR algorithm . . . . .	52
5.2	Solution space of the state equation . . . . .	53
5.3	Main window of the CEGAR plug-in. . . . .	54
5.4	Place names with IDs. . . . .	54
5.5	Marking editor dialogue. . . . .	55
5.6	Predicates tab. . . . .	56
5.7	Predicate editor. . . . .	56
5.8	Configuration dialogue. . . . .	57
5.9	Result of a problem, where the target is reachable. . . . .	58
5.10	Place selector dialogue. . . . .	59

# Chapter 1

## Introduction

PetriDotNet is a tool to edit, simulate and analyse Petri nets. It has been developed at the Fault Tolerant Systems Research Group (FTSRG) of the Budapest University of Technology and Economics (BUTE), Hungary to provide an easily usable and extensible tool.

### 1.1 System Requirements

PetriDotNet requires the Microsoft .NET framework (4.5 or newer) on Windows operating system (Vista or newer). During the development we have tried to maintain compatibility with the Mono framework in order to support Linux and Mac OS X operating systems, however, running PetriDotNet with Mono framework is not tested, therefore not supported.

Editing and simulation smaller Petri nets (containing hundreds of nodes) have a negligible memory consumption (10–20 MB). The tool is not optimised to graphically show or to edit huge Petri nets (with thousands of nodes). However, it is possible to open these nets in order to run the analysis modules on them.

The program does not need any installation. After extracting the compressed file to any directory, the program is ready to be used by running the `PetriDotNet.exe`.





## Chapter 2

# Basic Operation

This chapter discusses the basic operations in the PetriDotNet tool. It aims to be an easy-to-use tool, thus learning its basic functionality should be simple. In the following, we introduce the reader to the basic operations of PetriDotNet.

### 2.1 Starting the Tool

After starting the tool by executing `PetriDotNet.exe`, the tool opens with an empty editor window (see Figure 2.1).

The PetriDotNet tool can be operated from the main menu or by clicking on the toolbar icons. After starting the tool, the user can create a new, empty Petri net (`File / New`) or load an existing net (`File / Open`).

### 2.2 Load, Save

The tool natively supports two different file formats.

- **.pnml** This is a standardised, XML-based file format, supported by several Petri net editor tools. This is the preferred file format to store Petri nets.
- **.pn** This is a binary file format. Its advantage is the faster loading time, however it is a non-standard format.

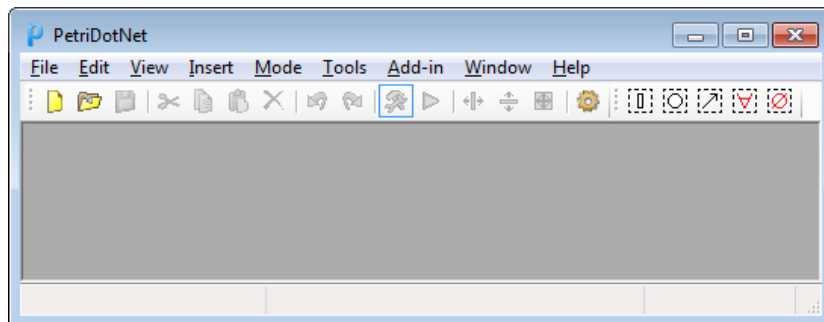


Figure 2.1: Empty editor window.

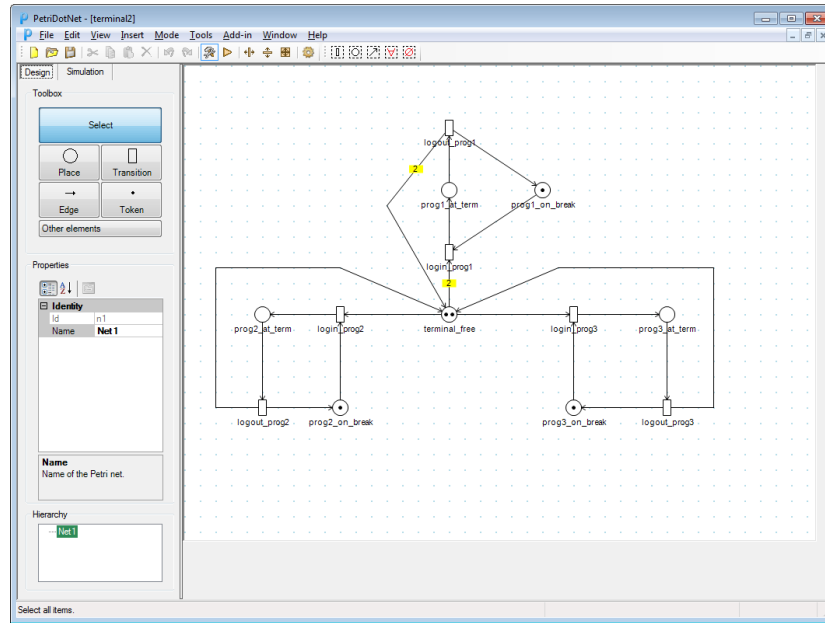


Figure 2.2: Main window with a loaded Petri net.

Besides these file formats, PetriDotNet can provide export or import functionalities via plug-ins. Currently there is a possibility to export the edited Petri nets into other Petri net formalisms, such as to the syntax of the GPenSim<sup>1</sup>, LivePN<sup>2</sup> and .pnt format of INA<sup>3</sup> (Integrated Net Analyzer) tools. Also, the Petri nets can be translated into the input format of SAL<sup>4</sup> (Symbolic Analysis Laboratory). Furthermore, importation from the .net textual Petri net file format, used by e.g. the INA tool, is also provided.

## 2.3 Creating and Editing Petri Nets

PetriDotNet provides a multi-document interface (MDI) for editing, i.e. multiple Petri nets can be opened in different windows inside the main PetriDotNet tool window. The user can change the currently edited Petri net or rearrange the Petri net windows in the **Window** menu.

The size of the editor pane (where the Petri net can be edited) is fixed. It can be changed by various commands: the **View / Fit to net** (*Alt+F7*) resizes the editor pane to the size of the currently edited Petri net, while the **ViewGrow width** (*Alt+Right*) increases the width, the **ViewGrow height** (*Alt+Down*) increases the height of the editor pane. These commands can be directly accessed from the toolbar. The size of the editor pane is automatically increased when a Petri net element is placed at the edge of the editor pane.

By default the tool is in *Design* mode. In this mode the left side of the editor

<sup>1</sup><http://www.davidrajuh.net/gpensim/>

<sup>2</sup>?**TODO**:What is this?

<sup>3</sup><http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/>

<sup>4</sup><http://sal.csl.sri.com/>

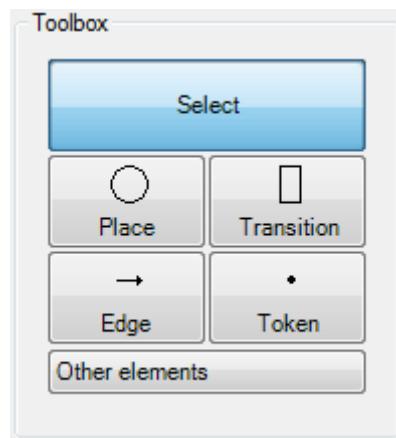


Figure 2.3: The toolbox in Design mode.

window contains a *Toolbox*, where the Petri net elements (place, transition, edge, token, etc.) to be inserted can be selected. To insert a Petri net element, first the element type has to be selected in the *Toolbox*, then the element can be inserted by clicking on the editor pane. By using the *Token* tool, the initial token count of a selected place can be increased by one.

To insert an edge, select first the source node by clicking on it, then click on the target node (thus the edges are not drawn by click-and-drag mechanism). Edge mid-points can be added by clicking on an empty part of the editor pane after before clicking on the target node of the edge.

In *Select* mode (after clicking on the [Select] button in the *Toolbox*), the Petri net elements can be selected and their properties can be changed. After selecting an edge, its mid-points are visible and editable. If the position of a mid-point should be changed, it can be simply dragged with the cursor after selecting the edge. If a mid-point should be deleted, right click on the mid-point after selecting the edge and click on the **Delete edge point** menu item. Mid-points can be inserted after an edge is inserted. To do this, press [Ctrl] and click on the edge where the mid-point should be placed.

If the *Enable edit helper tool* feature is enabled in the *View* tab of the *Settings* panel, a small toolbar becomes visible when a place or transition is selected (see Figure 2.5). This allows to easily insert a new item, connected to the selected node.

The properties of the selected node can be changed in the *Properties* panel on the sidebar (see Figure 2.6). A short description is visible about the selected property. The properties of the Petri net elements can be edited directly from the editor pane, by right clicking on the element and selecting the **Properties** menu item.

Some of the properties can be changed without using the *Properties* panel. Using the previously mentioned *Token* tool, the initial token count of the places can be increased. The token count of the selected place can also be increased or decreased by one by using the *Ctrl+Num+* or *Ctrl+Num-* hotkeys.

The same hotkeys work for selected edges, in this case they increase or decrease the weight of the selected edges by one.

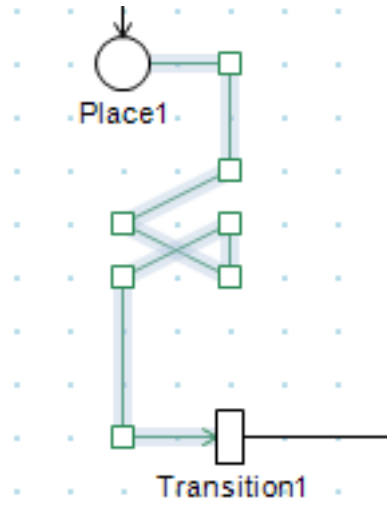


Figure 2.4: Drawing edges between Petri net elements.

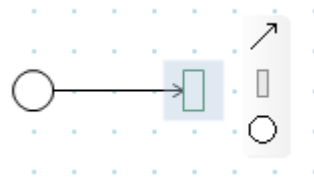


Figure 2.5: Edit helper tool.

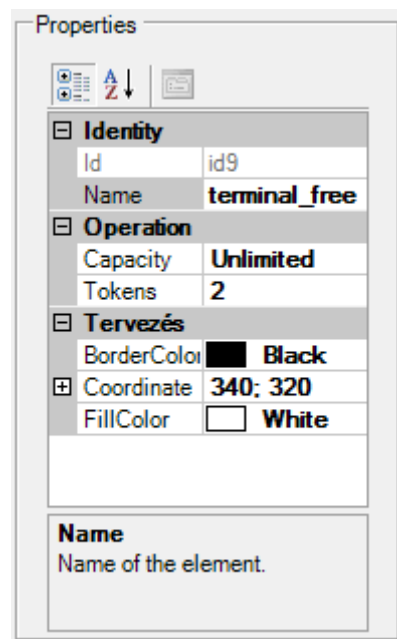


Figure 2.6: Properties panel.

Another property that can be quickly changed without using the *Properties* panel is the orientation of a transition (vertical or horizontal). To change the orientation of a transition, press *Ctrl* and click on the transition whose orientation should be changed. The orientation of a transition is only a visual property and it does not affect the behaviour of the Petri net.

To improve the readability of the Petri net, the name of a transition or place is not visible by default. The visibility of the names can be controlled using the *Name Visibility* property in the *Properties* panel. When the name of a node is manually changed, the name visibility will be set to visible. The names can be made visible using a hotkey too: press *Alt*, then click on the nodes.

The names of the transitions and places can be shown on the top, bottom, left or right side of the element. This can be changed by modifying the *Text Align* property in the *Properties* panel or by using the following hotkeys: *Ctrl+Up*, *Ctrl+Down*, *Ctrl+Left*, *Ctrl+Right*.

## 2.4 Editing Hierarchical Petri Nets

In order to help the readability of the edited Petri nets, PetriDotNet supports hierarchical modelling. The hierarchy (the tree structure of nets and subnets) can be seen in the *Hierarchy* box. By clicking on one of the nets in the *Hierarchy* box the selected (sub)net will be displayed. The subnets are handled via *coarse transitions* (CTs). A coarse transition enables to refine a modelled event or process (originally modelled by a single transition) into a subnet.

Consequently, each subnet is represented by a CT in its parent net. The CT behaves “syntactically” as a transition, thus it can only be connected to places.

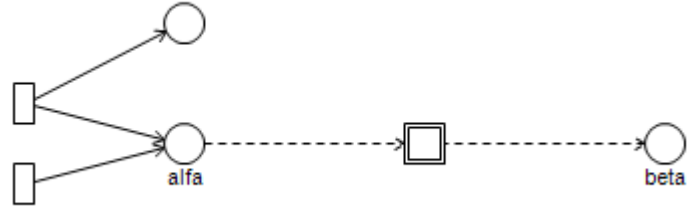


Figure 2.7: Example Petri net with coarse transition.

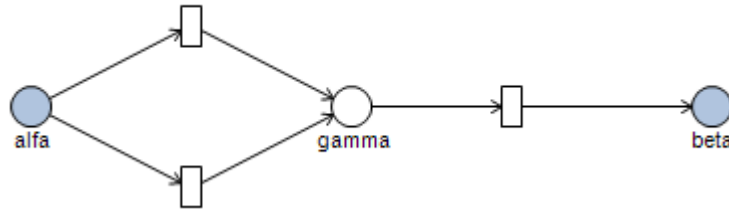


Figure 2.8: Example subnet.

When an edge is drawn between a place  $p$  and a coarse transition  $t$ , a reference place is created referring to  $p$  in the subnet represented by  $t$ . The behaviour of the reference places are exactly the same as their parents. By default the reference places are coloured to blue, but this can be overwritten by the user.

When an edge is drawn between a place  $p$  and a coarse transition  $t$ , the user might not want to create a new reference place in the subnet of  $t$ . Alternatively, one of the existing places of the subnet can be transformed into a reference place. To do this, hold the *Ctrl* key while connecting the place and the coarse transition, then select from the displayed list which place of the subnet should be transformed into a reference place.

The edges between the places and the CTs in the parent net are only to visualize connections. As these are not real Petri net edges, they are displayed as dashed lines and they have no multiplicity. The arrows denote the directions of the edges in the subnet.

**Example 2.1** Consider the example Petri net in Figure 2.7. The coarse transition represents the subnet shown in Figure 2.8. This hierarchical Petri net is equivalent to the flat Petri net in Figure 2.9.

The coarse transitions can be copied. When a CT is copied, the represented subnet is copied too. This subnet will have exactly the same reference places, except if some of the parents of the reference places were selected while copying. If a parent place is selected while copying, they will be copied in the parent net, and in the copied subnet the reference places will point to the copies in the parent net. Check Table 2.1 for an example.

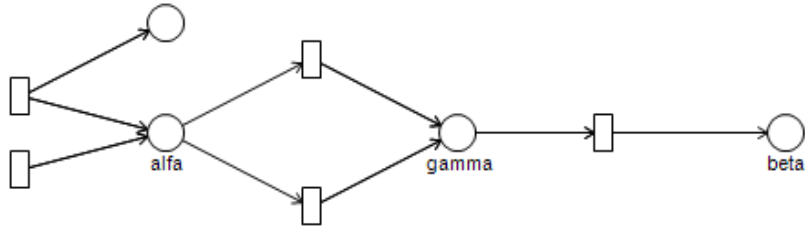


Figure 2.9: The flat Petri net equivalent to Figure 2.7 and 2.8.

Table 2.1: Example for copying coarse transitions

Selected elements before copying	Result of the copy operation

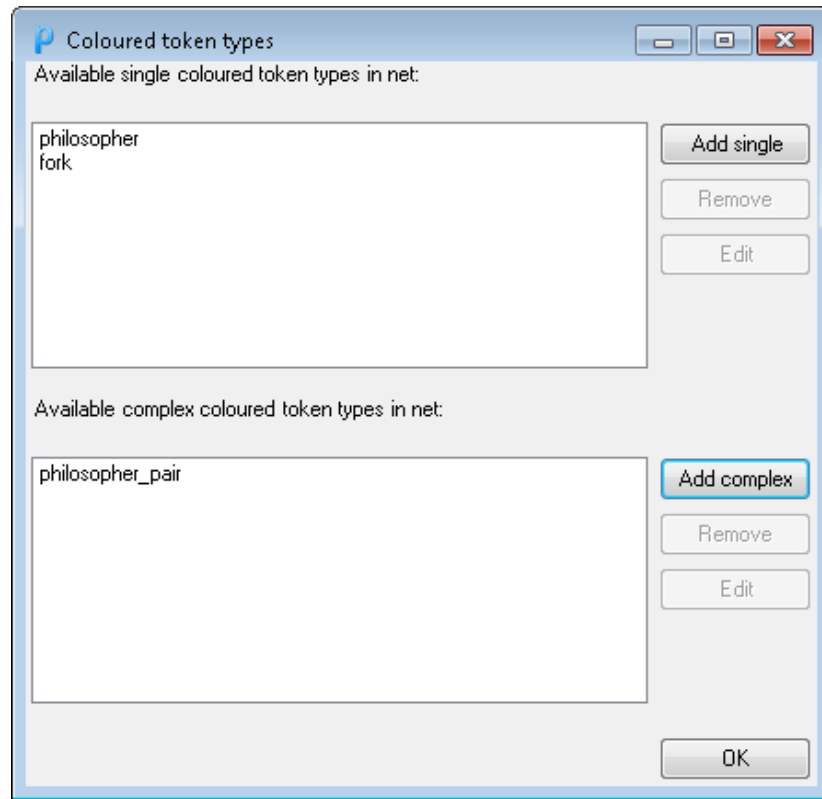


Figure 2.10: The *Coloured token types* window.

## 2.5 Coloured Petri Nets

### 2.5.1 Editing Coloured Petri Nets

The coloured Petri nets can be edited similarly to the uncoloured Petri nets. To insert a coloured place or coloured transition, click on the [Other elements] button in the *Toolbox*, then select the Coloured Place or Coloured Transition item. The edges between coloured places and coloured transitions can be drawn using the *Edge* tool.

### 2.5.2 Handling Colour Sets

The coloured Petri nets rely on the defined colour sets (or token types). The token types can be defined in the CPN / Token types menu item.

The *Coloured token types* window (Figure 2.10) shows the defined token types and allows the user to define new token types. Two token types are distinguished: *single* and *complex* (compound) token types. The simple token types are enumerations, while the complex token types are cross products of single token types.



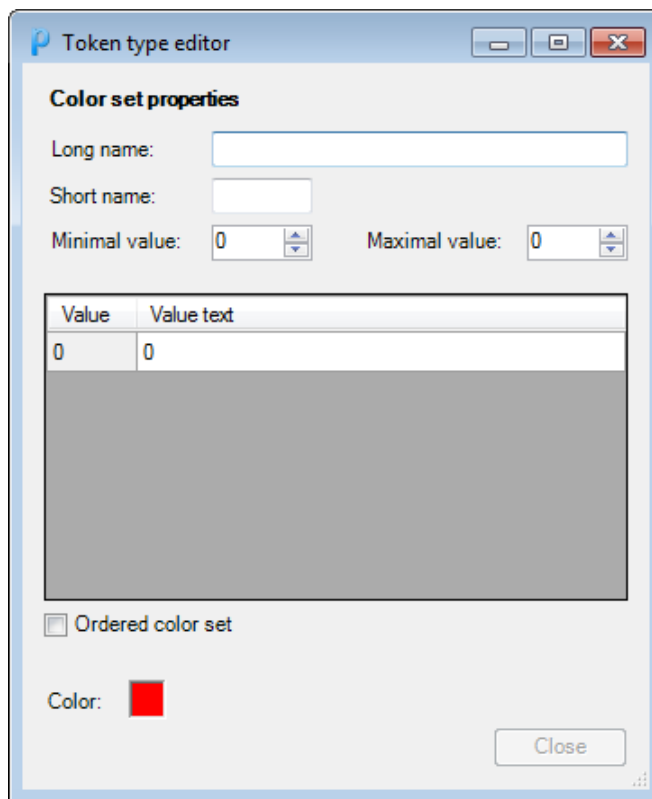


Figure 2.11: The *Token type editor* window for single token types.

**Adding Single Token Types.** By clicking on the [Add single] button, a new single token type can be added. To define a new colour set, the following information is needed (Figure 2.11):

- *Long name*: the long name of the token type.
- *Short name*: the short name of the token type.
- *Minimal value*: the minimum integer value in the colour set.
- *Maximal value*: the maximum integer value in the colour set.
- *Ordered color set*: if this checkbox is checked, the *succ* (successor) operation is defined for the elements of the colour set based in the items' integer values.
- *Color*: the color used on the graphical user interface to show the tokens of this type.

Furthermore, the user can assign names to each items by modifying the *Value text* in the table.

**Example 2.2** To represent Dijkstra's famous dining philosophers model using coloured Petri nets, a philosopher token type is needed, as shown in Figure 2.12.

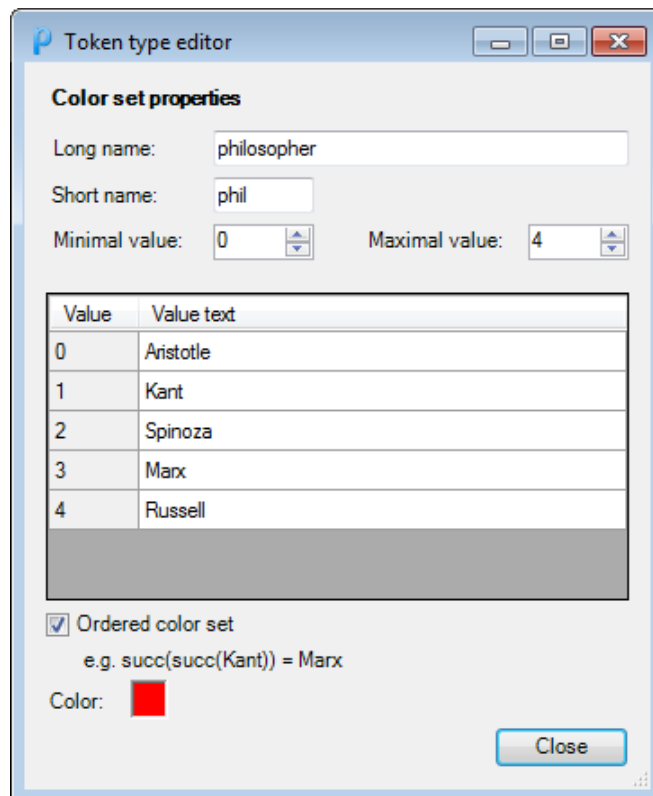


Figure 2.12: The *Token type editor* window filled with example data.

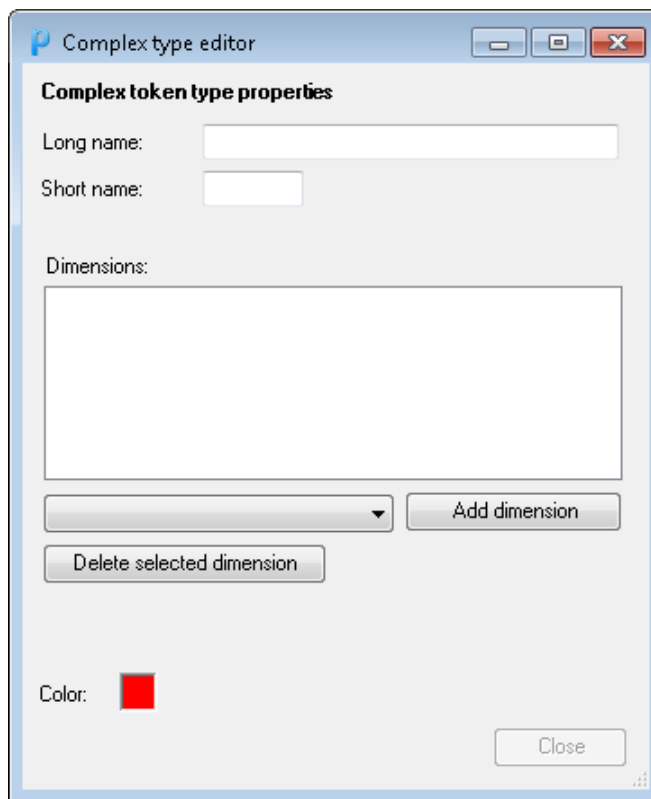


Figure 2.13: The *Token type editor* window for complex token types.

**Adding Complex Token Types.** By clicking on the [Add complex] button in the *Coloured token types* window, a new complex token type can be added. To define a new complex token type, the following information is needed (Figure 2.13):

- *Long name*: the long name of the token type.
- *Short name*: the short name of the token type.
- *Dimensions*: the ordered list of the single token types. New item can be added by selecting a single token type and clicking on the [Add dimension] button. The selected token type can be removed by clicking on [Delete selected dimension].
- *Color*: the color used on the graphical user interface to show the tokens of this type.

The token type of a selected coloured place can be easily set or modified by selecting a token type in the *Coloured set* section of the *Properties* box. In a valid coloured Petri net, each coloured place should have an assigned token type. If a coloured place has no defined token type, it is drawn in red.

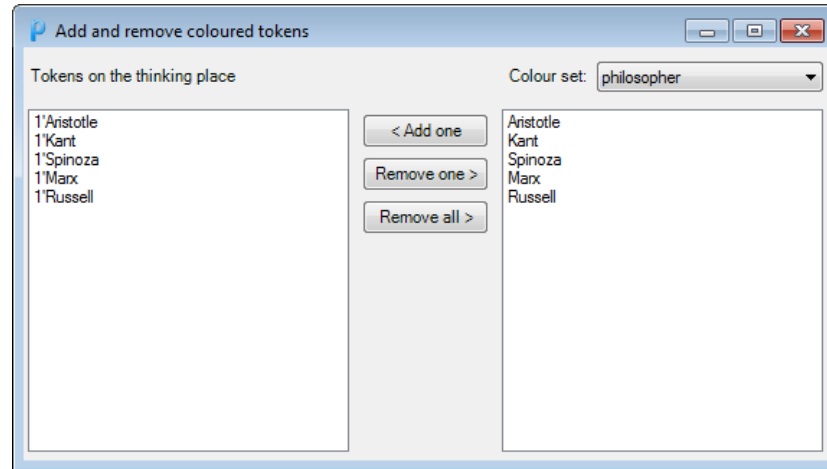


Figure 2.14: The *Add and remove coloured tokens* window.

### 2.5.3 Handling Coloured Tokens

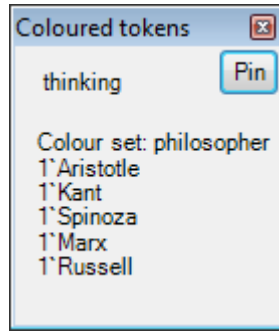
The initial tokens of a coloured place can be changed in the *Initial tokens* row of the *Properties* box, by clicking on the [...] button. This opens the *Add and remove coloured tokens* window (Figure 2.14). The left side of the window shows the current initial tokens of the selected coloured place, together with their multiplicity. The right side of the window shows the tokens available in the selected token type. To add a new token, select one or more tokens from the list on the right side, then click on the [**< Add one**]. This will add one of each selected tokens to the initial token set of the selected coloured place. To remove initial tokens, select the tokens in the list on the left, then click on the [**Remove one >**]. This reduces the multiplicity of each selected tokens by one. Clicking on the [**Remove all >**] removes all the selected tokens.

The initial tokens of a coloured place can also be given in the *Properties* box in a textual format, using the following format: `<multiplicity>'<value_vector>`. The definition of `<value_vector>` can be found in Section 2.5.4. The `all` is a special word, meaning all elements of the selected token type. If the initial token set contains multiple tokens, they can be separated using a `++` sign.

**Example 2.3** *The following examples are valid textual representations of initial token sets:*

- `1'Aristotle`
- `2'[0,3]++1'[0,0]`
- `all`

The editor shows only the number of tokens visually. To get more information, select the **View / Show coloured tokens form** command. This opens a small window (Figure 2.15) that shows more information about the place that is under the mouse cursor at the moment. This window can be used both in *Design* and *Simulation* modes.

Figure 2.15: The *Coloured tokens* window.

### 2.5.4 Edge Expressions, Guards and Variables

The dynamic behaviour of a coloured Petri net relies on the expressions written on the coloured edges. These edge expressions control the flow of tokens.

The edge expression of a selected edge can be set or modified in the *Arc expression* row of the *Properties* box. An edge expression can only contain variables and constants. If the edge expression contains multiple elements, they have to be separated by ++ characters. The valid edge expressions are defined as follows:

```

<EdgeExpr> ::= <value_vector> |<EdgeExpr> '++' <value_vector>
<value_vector> ::= '[' <value> ',' <value> ',' ... ']'
<value> ::= <variable_name> |<constant>

```

Another tool to control the token flow is the usage of guard expressions, assigned to coloured transitions. A coloured transition can fire only if its guard expression is satisfied. The guard expression of a selected coloured transition can be set or modified in the *Guard* row of the *Properties* box. The syntax of the guard expressions is the following:

```

<GuardExpr> ::= <expression> |<GuardExpr> <bool_op> <GuardExpr>
<bool_op> ::= '&&' |'||'
<comp_op> ::= '>' |'>=' |'<' |'<=' |'=' |'!=' |'<'
<expression> ::= <gvalue> <comp_op> <integer> |
                <gvalue> <comp_op> <gvalue>
<gvalue> ::= <variable_name> |'succ' <gvalue>

```

**Example 2.4** *Examples for syntactically correct guard expressions:*

- $fb=succ\ fj$
- $fb > 2$
- $fb > 2 \ \&\&\ fj < 1$

The variables used in guard expressions have to be defined explicitly. This can be done by selecting the CPN / Variables menu item that opens the *Variables* window (Figure 2.16).

In the *Variables* window, the defined variables and their types are listed. New variable can be added by clicking on [Add]. The selected variable can be deleted by clicking on [Remove]. The type of each variable should be a single token type.

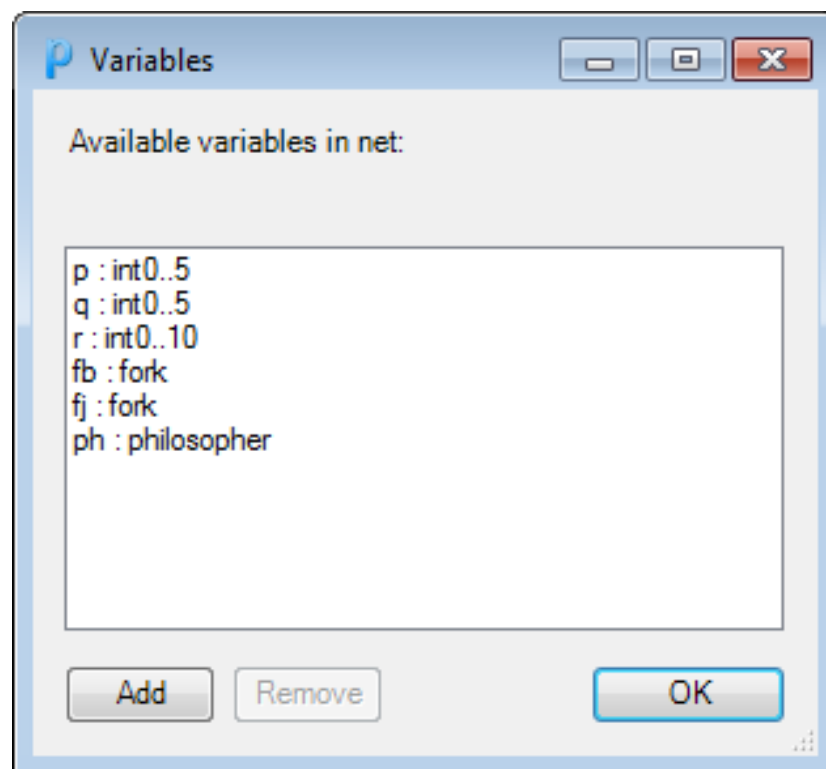
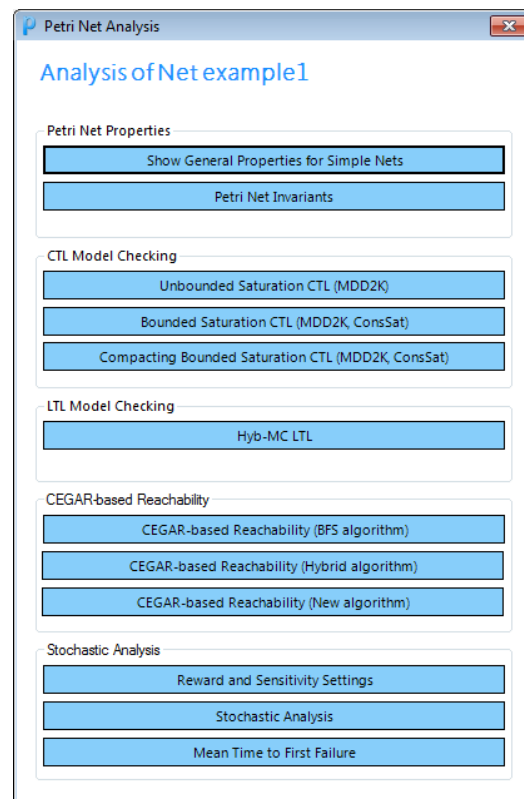


Figure 2.16: The *Variables* window.

Figure 2.17: The *Net analysis* window.

### 2.5.5 Limitations

Hierarchical coloured Petri nets are not supported currently . Coloured and uncoloured nodes cannot be connected.

## 2.6 Petri Net Simulation

There are two main ways to use a Petri net: simulation (or token game) and analysis (e.g. checking dynamic properties, computing invariants, perform model checking). PetriDotNet has a built-in support for simulation and various analysis algorithms can be used as plug-ins. This section discusses the simulation capabilities of PetriDotNet. The analysis plug-ins are discussed in detail in the following chapters. Most of them can be accessed in the **Add-in / Net analysis** menu (Figure 2.17).

In the *Simulation* mode the user can intuitively check the behaviour of a Petri net. Two types of simulation can be used in PetriDotNet: an interactive, step-by-step simulation and a non-interactive simulation (execution).

In case of the step-by-step simulation (*Step by step* mode, Figure 2.18) the currently enabled transitions are denoted by a red border. In this mode the enabled transitions can be fired one-by-one, by clicking on them or by choosing from the list on the panel on the left. By clicking on the **[Step one]** button, a

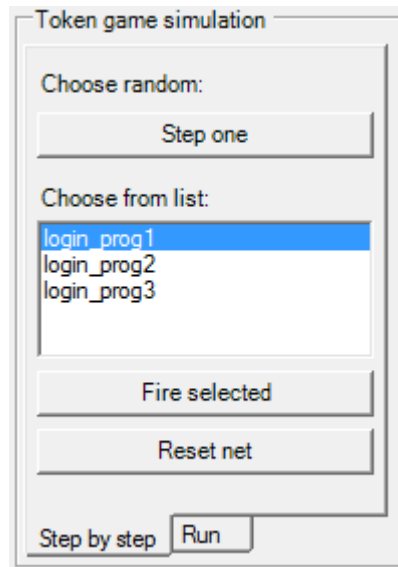


Figure 2.18: Settings of the interactive simulation.

randomly chosen enabled transition will be fired. The [Reset net] button will reset the net to the initial marking.

The execution mode can be accessed on the *Run* tab (Figure 2.19). In this mode a randomly chosen enabled transition is fired automatically at regular intervals. This interval can be set using the *Tick interval* input. The execution can be started by clicking on the [Run] button. The conflicts of the Petri net are resolved non-deterministically. The *Transitions fired last* field shows the last fired transitions.

Non-interactive simulation can also be performed using the Large scale statistics plug-in (Add-in / Statistics / Large scale statistics, Figure 2.20). The execution can be started by clicking on the [Run] button. This will start a firing sequence containing a pre-defined number of firings (*Number of firings* setting). After the execution, statistics about the firing sequence are presented in the window.



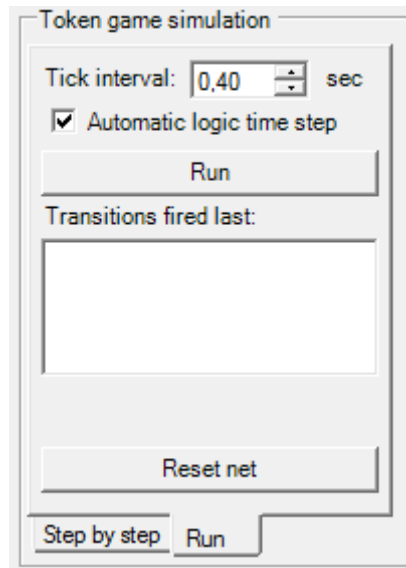


Figure 2.19: Settings of the non-interactive simulation.

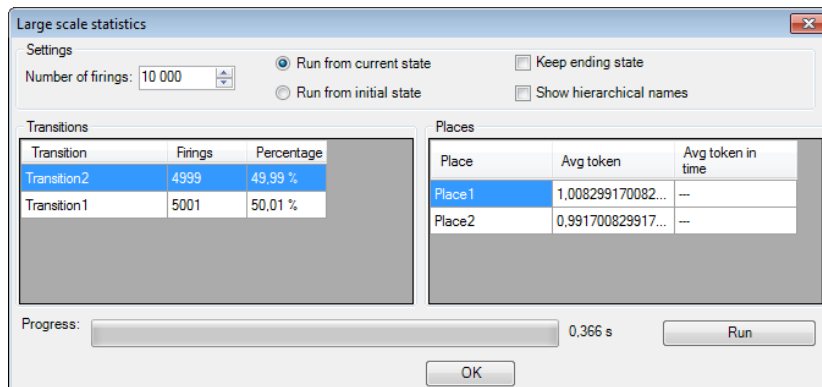


Figure 2.20: Large scale statistics plug-in.



## Chapter 3

# CTL Model Checking

This chapter overviews the CTL model checking capabilities of PetriDotNet. The CTL model checking algorithms rely on saturation, a symbolic state space exploration and model checking algorithm [1] for finite Petri nets. This chapter discusses the usage of the algorithms. The theoretical details of the implemented algorithms can be read in [10, 9, 2].

Saturation-based model checking algorithms typically have three steps:

1. **Model decomposition.** In this step, the model is decomposed into “levels”. One model level will be represented by one level in the decision diagram.
2. **State space exploration.** In this step, the reachable state space of the model is explored and symbolically stored.
3. **Evaluation of the requirement.** In this step, the given requirement (in our case: CTL expression) is evaluated.

The implementation used in PetriDotNet follows these steps.

PetriDotNet provides both unbounded and bounded CTL model checking algorithms. The details of these algorithms are discussed in the following.

### 3.1 Unbounded CTL Model Checking

Unbounded CTL model checking can be executed by clicking on the [Unbounded Saturation CTL] button in the *Petri Net Analysis* window (Add-in / Net analysis, Figure 2.17).

**Model Decomposition.** After clicking on the [Unbounded Saturation CTL] button, first the model decomposition has to be given. There are three possibilities for the decomposition (Figure 3.1):

- Setting the decomposition manually ([Manually]),
- Load a previously saved decomposition ([Load from file]),
- Generate a decomposition automatically using experimental P-Invariants based heuristics ([P-Invariants]).

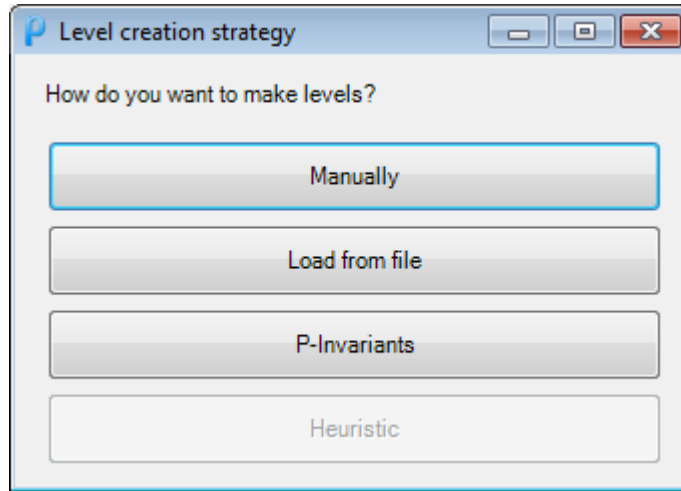


Figure 3.1: Form to select decomposition strategy.

If the manual decomposition definition option was selected, the user has to set up the decomposition. First, some places have to be selected. They will be assigned to Level 1. This choice is confirmed by clicking on the [OK] button. Then the places for Level 2 have to be selected. This continues while there are unassigned places. By selecting a number in the input box and clicking on the [Every N] button the places will be split into levels: the 1st, 2nd, ...,  $N$ th places will be assigned to Level 1, the  $N + 1$ th, ...,  $2N$ th places to Level 2, etc.

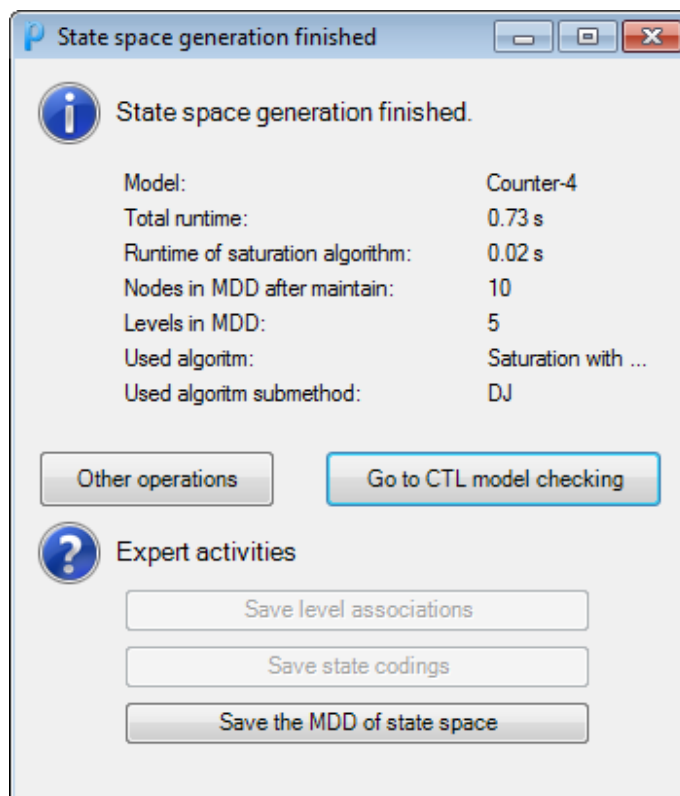
**CTL Expressions.** After the level decomposition is done, the state space exploration algorithm is executed. After that a window (Figure 3.2) notifies the user that the state space exploration is finished. It shows some basic metrics, such as the runtime (*Runtime of saturation algorithm*) and the size of the decision diagram describing the state space of the Petri net (*Nodes in MDD after maintain*). By clicking on [Go to CTL model checking] button the CTL expression to be checked can be given and evaluated.

The following grammar defines the syntax of the accepted CTL expressions ( $\langle \text{Expr} \rangle$ ).

```

<Expr> ::= <CTL_expr> |
          <place_expr> |
          '!' <Expr> |
          '(' <Expr> ')' |
          <Expr> <bool_op> <Expr>
<CTL_expr> ::= <CTL_un_op> '(' <Expr> ')' |
              <CTL_bin_op> '(' <Expr> <bin_separator> <Expr> ')'
<CTL_un_op> ::= 'EX' | 'EF' | 'EG' | 'AX' | 'AF' | 'AG'
<CTL_bin_op> ::= 'EU' | 'EW' | 'ER' | 'AU' | 'AW' | 'AR'
<bin_sep> ::= 'u' | 'w' | 'r'
<place_expr> ::= <place_identifier> <comp_op> <integer>
<bool_op> ::= '&&' | '||'
<comp_op> ::= '>' | '>=' | '<' | '<=' | '=' | '!=' | '<>'

```

Figure 3.2: The *State space generation finished* window.

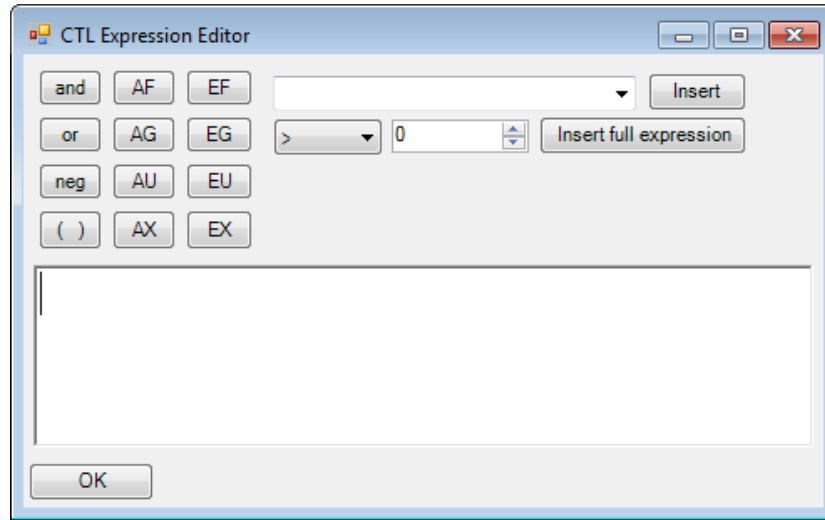


Figure 3.3: CTL expression editor.

The `<place_identifier>` denotes the name or ID of a place. To help the user, a CTL expression editor with syntax highlighting is available whenever a CTL expression has to be provided (Figures 3.3 and 3.4). A complete `<place_expr>` expression can be inserted to the CTL expression editor field by selecting a place the drop-down list, setting the comparison operator and the integer, and finally clicking on the [Insert full expression].

After clicking on the [OK] button, the given CTL expression is evaluated. The result is presented in a simple message box (Figure 3.5). By clicking [OK] on this message box, the CTL expression editor is shown again, where another CTL expression can be specified for evaluation. In this way multiple CTL expressions can be checked after a single state space exploration step. By closing the CTL expression editor, the main window of PetriDotNet is accessible again.

## 3.2 Bounded CTL Model Checking

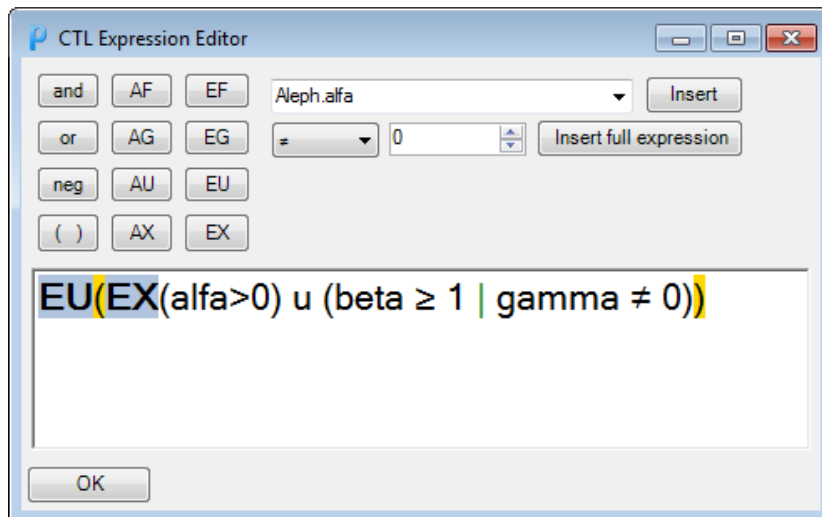


Figure 3.4: CTL expression editor with an example expression.

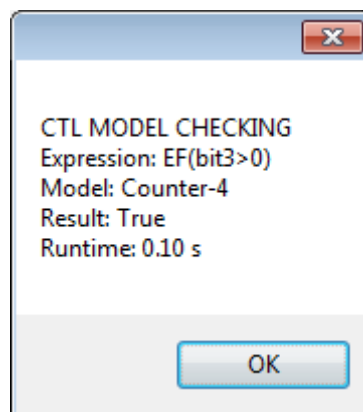


Figure 3.5: Example CTL model checking result.

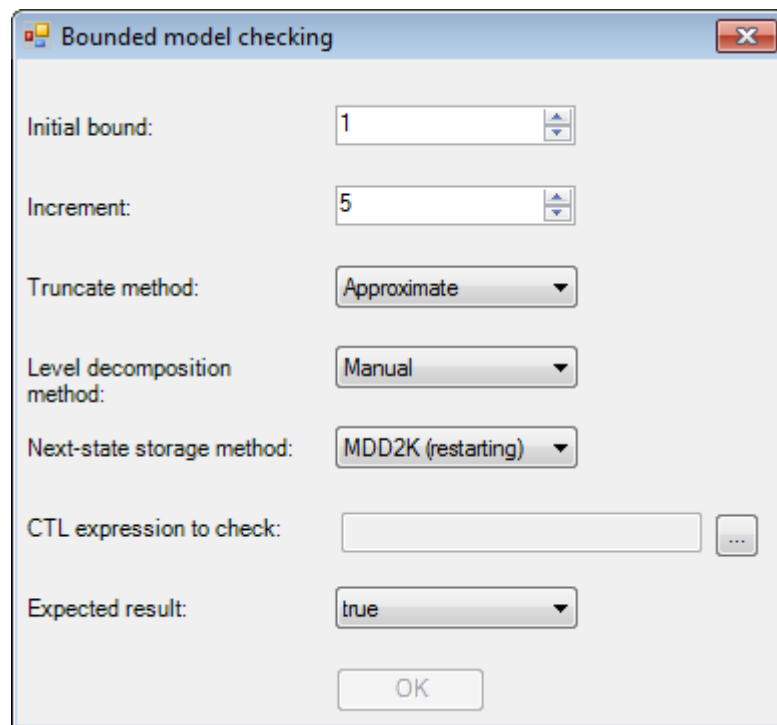


Figure 3.6: Settings of bounded CTL model checking.



## Chapter 4

# Stochastic Analysis Module

The capabilities of the stochastic analysis component will be demonstrated with the help of the example model depicted in Figure 4.1. In this manual we will only introduce the features provided by the user interface. For more details please refer to [7].

**Example 4.1 Hybrid cloud** *The system processes a finite number of incoming jobs (to ensure a finite state space) with the help of a private or a public cloud chosen by a simple probability based load balancer. The submodel for the load balancer is shown in Figure 4.2.*

*We assume that there are infinite servers available in the public cloud that never fail. In the private cloud we have a finite number of resources and there is a chance that a server will fail and needs to be repaired.*

### 4.1 Performance measures

Before presenting the various stochastic analysis features of the tool we gather some interesting question about the model we would like to answer with stochastic analysis.

1. In steady state how much time does it take to process an incoming job?
2. In steady state how much is the operational cost of the system?
3. How will these values change if the rate of the incoming jobs or the settings of the load balancer changes?
4. How many jobs can be processed in the first hour?
5. What is the mean time to first failure (MTFF) and mean uptime (MUT) of the system?

In the next sections we will present the various features of the tool that will aid us in answering these questions.

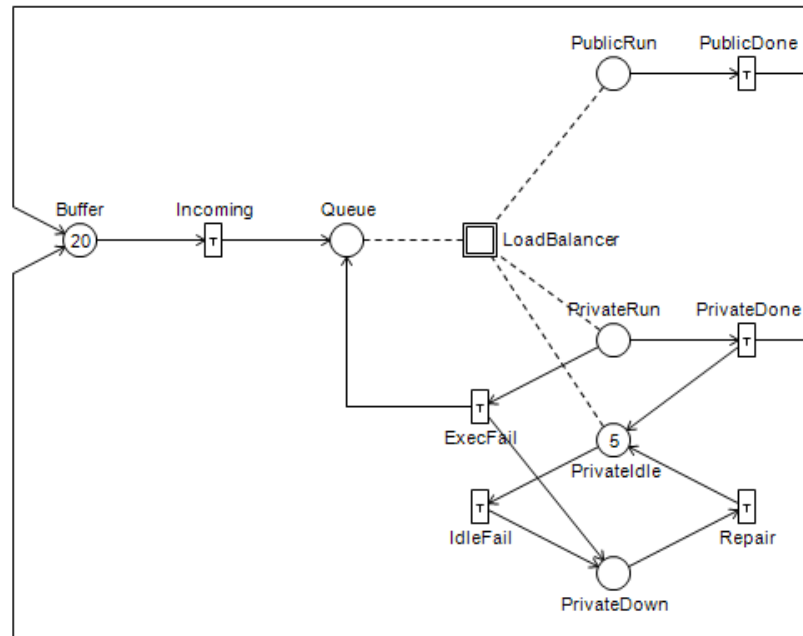


Figure 4.1: The example hybrid cloud model.

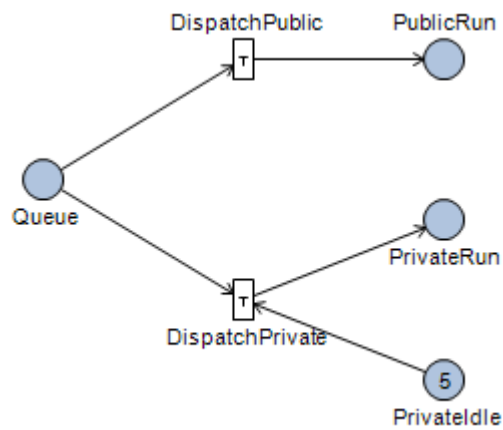


Figure 4.2: A simple load balancer model.

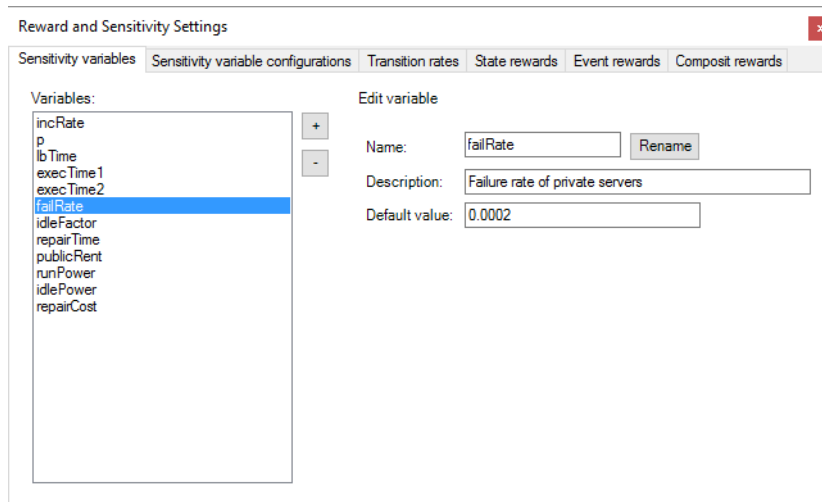


Figure 4.3: Sensitivity variables tab.

## 4.2 Model and reward settings

In order to answer the questions in Section 4.1 we need to define some variables, set the transitions rates and formalize the questions as reward expressions. These steps can be performed by selecting the `Add-in / NET ANALYSIS` menu item and clicking the `[Reward and Sensitivity Settings]` button in the shown form.

### 4.2.1 Sensitivity Variables

On the first tab on the settings form we can define variables for the model. We can use these variables later in reward and transition rate expressions as well as for sensitivity analysis. The settings tab is shown in Figure 4.3.

The existing sensitivity variables are listed on the left side of the form. We can add or remove variables with the `[+]` and `[-]` buttons, respectively. Once a variable is selected we can set its name, description (optional) and default value. The new name can be applied with the `[Rename]` button. The description and the default value are applied automatically as typed. In Figure 4.3 the selected sensitivity variable will denote the failure rate of the private servers.

### 4.2.2 Sensitivity Variable Configurations

On the second tab (shown in Figure 4.4) we can create different configurations for the sensitivity variables defined in the previous tab. At the top of the form the selected value of the combobox denotes the currently active configuration for the variables. The empty selection corresponds to the configuration containing the default values of the variables set in the previous tab.

The available configurations (beside the default configuration) are listed on the left side of the form. We can add or remove configurations with the `[+]` and `[-]` buttons, respectively. Once a configuration is selected we can set its name and the new values of the existing variables. The new name can be applied with

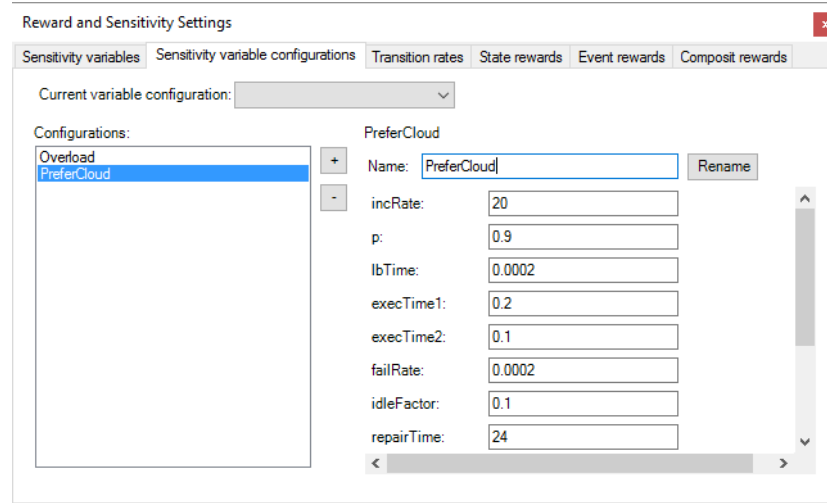


Figure 4.4: Sensitivity variable configurations tab.

the **[Rename]** button. The values of the variables are applied automatically as typed.

In Figure 4.4 the *PreferCloud* configuration is highlighted (but not selected!) which changes the default value of the  $p$  variable so that the load balancer will choose the public cloud resources with higher probability.

### 4.2.3 Transition Rates

On the third tab (shown in Figure 4.5) we can set the rate functions for the transitions of the Petri net. The transitions are listed on the left side of the form. By default the transitions don't have exponentially distributed firing delays associated with them, so this form provides a quick way to set this for every transition by clicking the **[Set all to exponential]** button.

Once a transition is selected additional options become available on the right side. We can set an exponential distribution for the selected transition or remove the distribution by clicking the **[Set to exponential]** or **[Clear]** buttons, respectively.

Once the distribution is set we can provide the rate of the transition as an arithmetic expression. In this expression we can use:

1. The common arithmetic operators ( $+$ ,  $-$ ,  $^$ ,  $*$ ,  $/$  and  $//$  for integer division)
2. Some predefined functions ( $exp()$ ,  $lg()$ ,  $lb()$ ,  $ln()$ ,  $log()$ ,  $sin()$ ,  $cos()$ )
3. The defined sensitivity variables

The available variables and functions can be inserted from the combobox below using the **[Insert]** button or typed directly into the expression editor textbox.

The provided expression can be checked for errors without applying it using the **[Check]** button. The expression can be applied as the rate function using

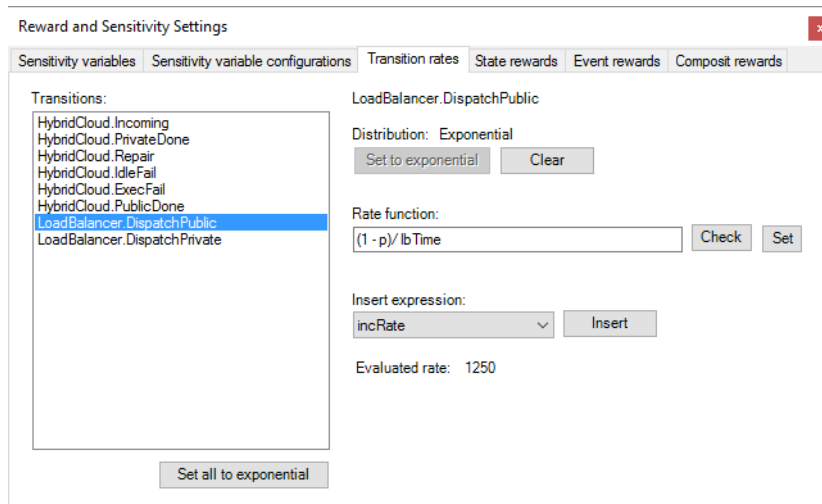


Figure 4.5: Transition rates tab.

the [Set] button. After setting the rate function it is evaluated and the result is displayed based on the currently selected variable configuration.

**Important: in order to perform stochastic analysis, every transition needs to have an exponentially distributed firing rate that evaluates to a nonzero positive number!**

#### 4.2.4 State Rewards

On the fourth tab (shown in Figure 4.6) we can define state reward configurations whose expected value can be evaluated during the stochastic analysis. Defining a state reward configuration means that we assign a value (i.e. a reward) to certain states of the system. The expected value of the configuration is calculated as the weighted sum of the assigned state rewards using the probabilities that the system is in the given state as weights.

The defined configurations are listed on the left side of the form. We can add or remove configurations with the [+] or [-] buttons, respectively. Once a configuration is selected we can set its name and add multiple reward assignments to the configuration which makes it possible to describe complex metrics with a single configuration. The new name can be applied with the [Rename] button.

The tool provides multiple ways to add a reward assignment to the configuration. The added assignments can be edited or removed with the [...] or [-] buttons, respectively.

##### Place-based assignment

Using a place-based assignment we essentially say that to every state (Petri net marking) of the model assign the following reward: the number of tokens on that place in that state multiplied by the given constant number.

We can add a place-based assignment using the [Place] button. In the appearing window (shown in Figure 4.7) we can select the place and set the

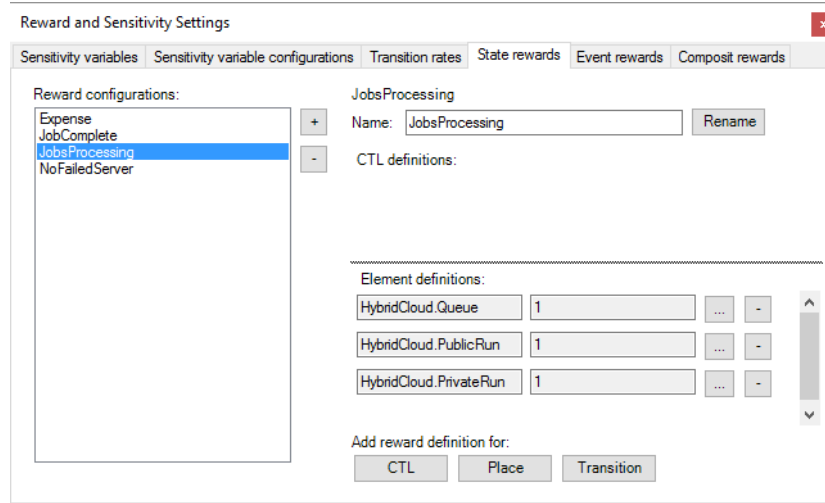


Figure 4.6: State rewards tab.

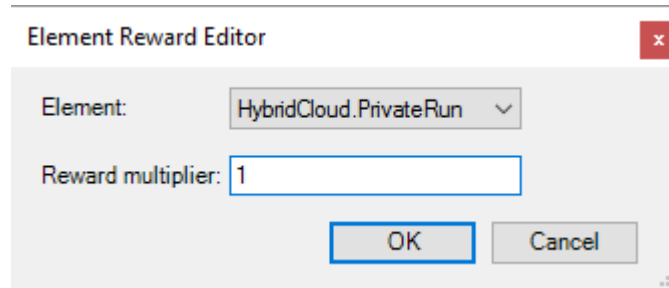


Figure 4.7: Place-based reward window.

constant multiplier for the assignment.

### Transition-based assignment

Using a transition-based assignment we essentially say that to every state of the model *where that transition is enabled* assign the following reward: the rate of the transition multiplied by the given constant number.

We can add a transition-based assignment using the [Transition] button. In the appearing window (same as Figure 4.7) we can select the transition and set the constant multiplier for the assignment.

### CTL-based assignment

Using a CTL-based assignment we essentially say that to every state of the model *where the CTL expression is true* assign the following reward: the evaluated value of an arbitrary arithmetic expression. In this expression we can use:

1. The common arithmetic operators (+, -, ^, \*, / and // for integer division)

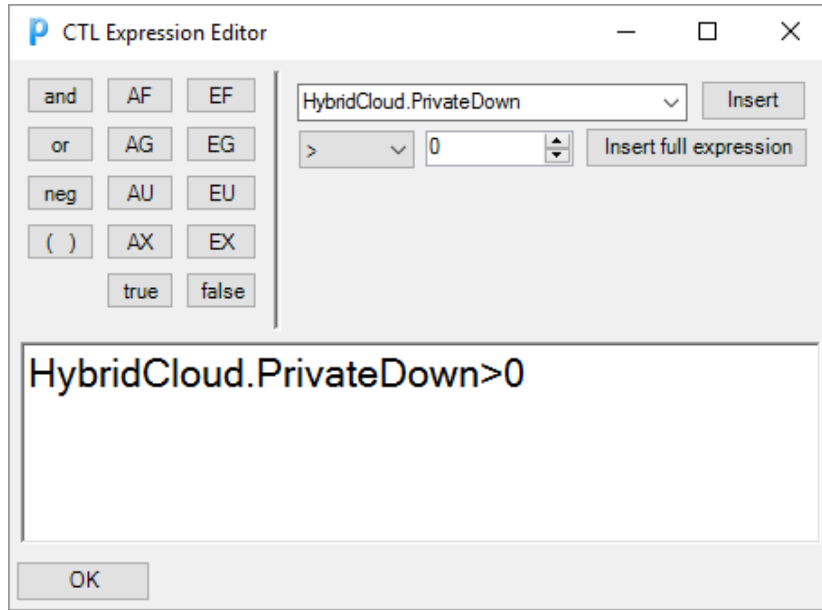


Figure 4.8: CTL editor window.

2. Some predefined functions ( $exp()$ ,  $lg()$ ,  $lb()$ ,  $ln()$ ,  $log(,)$ ,  $sin()$ ,  $cos()$ )
3. The defined sensitivity variables
4. The rate of a transition (accessed with the  $rate()$  function and the transition name)
5. The number of tokens on a place in the currently evaluated state (accessed with the place name)

We can add a CTL-based assignment using the [CTL] button. In the appearing window (shown in Figure 4.8) we can type the CTL expression. The buttons on the left can be used to insert basic elements (like operators) and the [Insert] or [Insert full expression] buttons can be used to insert place names or full logical expressions (given by the states of the surrounding controls), respectively. The CTL expression can be applied by clicking the [OK] button.

After setting the CTL expression another window pops up (shown in Figure 4.9) where we can set the arithmetic expression for the reward assignment. The available variables and functions can be inserted from the combobox below using the [Insert] button or typed directly into the expression editor textbox. The provided expression can be checked for errors without applying it using the [Check] button. The checked expression can be applied using the [OK] button.

### 4.2.5 Event Rewards

On the fifth tab (shown in Figure 4.10) we can define event reward configurations whose expected value can be evaluated during the stochastic analysis. Defining

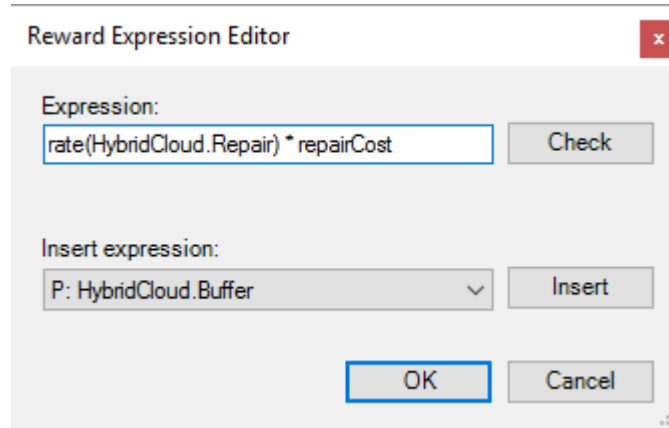


Figure 4.9: Arithmetic expression editor window.

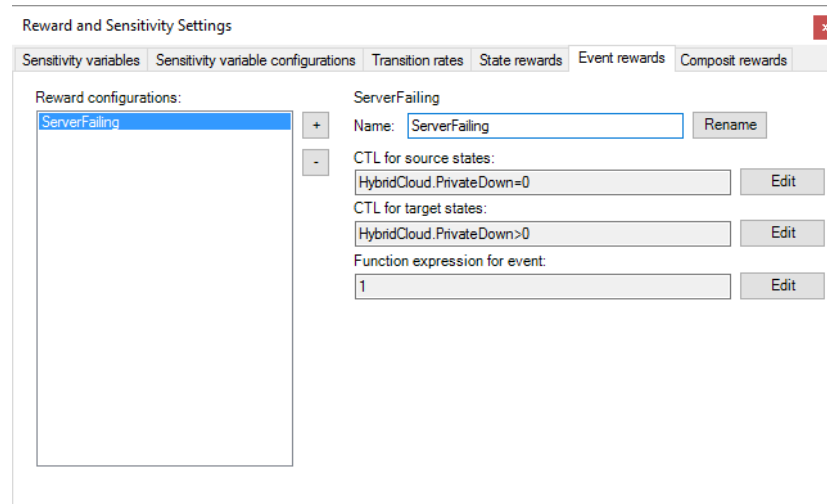


Figure 4.10: Event rewards tab.

an event reward configuration means that we assign a value (i.e. a reward) to certain state transitions of the system. The expected value of the configuration is calculated as the weighted sum of the assigned event rewards (first multiplied by the rate of the occurring event) using the probabilities that the system is in the given source state (i.e. the event can occur) as weights.

The defined configurations are listed on the left side of the form. We can add or remove configurations with the **[+]** or **[-]** buttons, respectively. Once a configuration is selected we can set its name and add multiple reward assignments to the configuration which makes it possible to describe complex metrics with a single configuration. The new name can be applied with the **[Rename]** button.

An event reward configuration consists of three parts. A CTL expression describing the set of source states, a CTL expression describing the set of target states and an arithmetic expression that will be evaluated on the matching



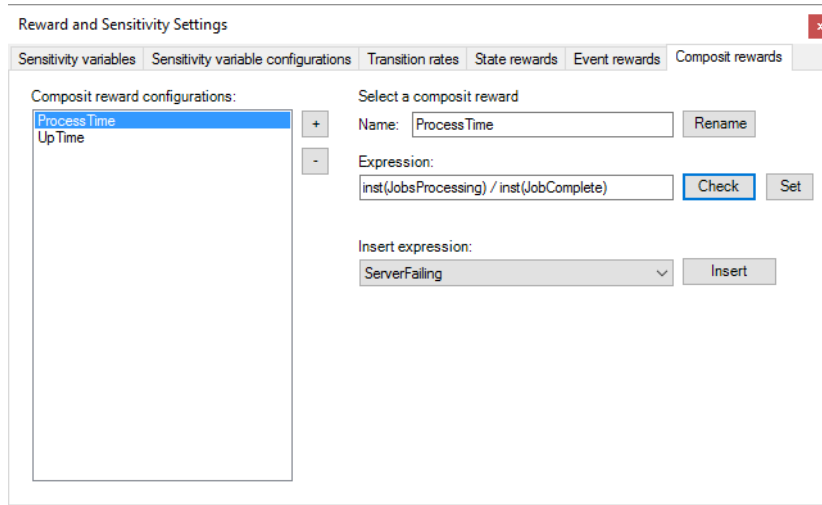


Figure 4.11: Event rewards tab.

events. An event is processed if and only if its source state is in the set of source states and its target state is in the set of target states described by the CTL expressions. The three expressions can be edited by clicking the [Edit] button and using the forms shown in Figures 4.8 and 4.9.

#### 4.2.6 Composite Rewards

On the sixth tab (shown in Figure 4.11) we can define composite rewards. A composite reward is simply an arithmetic expression that can contain expected values of other rewards as variables. This is useful for performing some post-processing on already calculated reward values.

The defined composite rewards are listed on the left side of the form. We can add or remove configurations with the [+] or [-] buttons, respectively. Once a configuration is selected we can set its name and the arithmetic expression. The new name can be applied with the [Rename] button. In the expression we can use:

1. The common arithmetic operators (+, -, ^, \*, / and // for integer division)
2. Some predefined functions ( $exp()$ ,  $lg()$ ,  $lb()$ ,  $ln()$ ,  $log(,)$ ,  $sin()$ ,  $cos()$ )
3. The defined sensitivity variables
4. The defined state or event reward configurations

The available variables and functions can be inserted from the combobox below using the [Insert] button or typed directly into the expression editor textbox. The provided expression can be checked for errors without applying it using the [Check] button. The expression can be applied using the [Set] button.

A reward configuration can be evaluated in two ways resulting in different values. Because of this we need to distinguish these values in the expression using

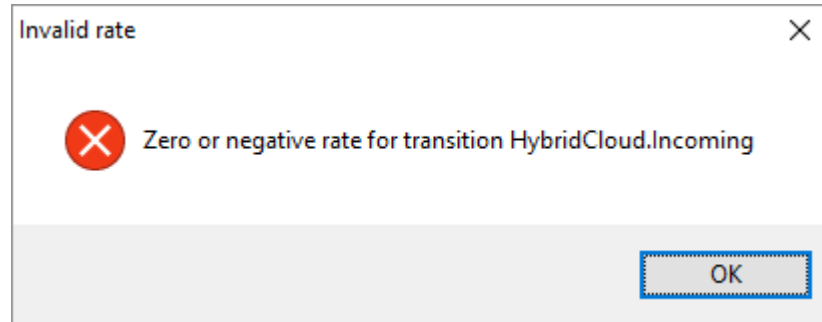


Figure 4.12: Invalid transition rate warning.

the *inst()* and *acc()* functions to reference the instantaneous and accumulated values of rewards, respectively.

#### 4.2.7 Reference consistency

Almost every arithmetic expression can refer to some kind of variable, e.g. a sensitivity variable, a reward configuration, a place name or a transition name (referred to as variables from now on). These are all values that can be modified through the user interface of the tool, thus every expression that references them must be updated. For sensitivity variables and reward configurations this update is performed automatically.

- If a sensitivity variable is renamed then every reference to it is renamed.
- If a new sensitivity variable is added to the model then this variable will be added to every existing sensitivity variable configuration with its default value.
- If a sensitivity variable is deleted then every reference to this variable will be set to zero. If this results in an invalid transition rate then a warning window similar to Figure 4.12 will be shown. It's recommended to manually check every expression referring the deleted variable.
- If a reward configuration is renamed then every reference to it is renamed.
- If a reward configuration is deleted then every reference to this configuration will be set to zero. It's recommended to manually check every expression referring the deleted configuration.

**Modifications to place and transition properties (e.g. deleting or renaming) are currently not applied to the expressions automatically!**

### 4.3 Formalizing the performance measures

In this section we will construct the variables and reward configurations necessary to answer the questions presented in Section 4.1.

**TODO:**Describing the needed variables and reward expressions.

Figure 4.13: Analysis configuration form.

## 4.4 Reward evaluation

After the parameters and rewards of the model are defined we can run the stochastic analysis. This can be done through the `Add-in / NET ANALYSIS` menu item and clicking the `[Stochastic Analysis]` button in the shown form.

### 4.4.1 Analysis Configuration

After clicking the `[Stochastic Analysis]` button the form in Figure 4.13 is shown. In this window we can configure the algorithms used during the stochastic analysis. The previously saved configuration are listed on the left side of the form. We can add or delete configurations using the `[+]` or `[-]`, respectively. The details of the selected configuration are shown on the right side of the form. The changes of a configuration can be saved using the `[Save]` button. The selected configuration can be run using the `[Next/Run]` button.

#### Overview

The analysis workflow consists of four main steps. Every step but the last can be configured to achieve the most suitable workflow, thus the best performance for the stochastic analysis.

1. **State space exploration:** The reachable state space of the model is explored.
2. **Generator matrix composing:** The matrix describing the stochastic model is composed using the reachable state space.

3. **Numerical solution:** The state of the system at a required time is calculated from the composed generator matrix.
4. **Reward evaluation:** The required reward configurations are evaluated based on the numerical solution of the system.

### Configuration details

In the next list we give a detailed description of the available configuration points and their possible values. We also highlight the effects of some settings to the analysis where its important.

**TODO:** Additional notes to the numerical algorithms

**Terminate workflow after** Determines the last step of the analysis after which the workflow will terminate. Affects the available settings on the form.

- *State space exploration:* The workflow will explore the state space of the system then terminate.
- *Generator composing:* The workflow will also compose the generator matrix from the state space then terminate.
- *Stochastic analysis:* The workflow will also calculate the system state from the generator matrix then terminate.
- *Run whole workflow:* The workflow will also evaluate the reward configurations based on the system state.

**State space exploration** Determines the algorithm used during the state space exploration of the system.

- *Explicit:* A simple graph traversal algorithm will be used. Not efficient for bigger state spaces.
- *Symbolic:* The saturation algorithm will be used. Recommended for huge state spaces.

**Generator operation configuration** Determines the way matrix algorithms will be executed during the generator matrix composing step.

- *Sequential:* The matrix algorithms will be executed sequentially.
- *Parallel:* The matrix algorithms may be executed in parallel depending on the size of the matrix.

**Generator matrix storage** Determines the storage structure for the generator matrix.

- *Dense:* The generator matrix will be stored as a dense matrix. Only recommended for small matrices.
- *Sparse:* The generator matrix will be stored as a sparse matrix. Recommended for bigger matrices.
- *Block Kronecker Decomposition:* The generator matrix will be stored in a decomposed form. Recommended for really huge matrices. Can change the convergence of some algorithms.

**Components of model** Determines the decomposition method of the model in case of symbolic state space exploration or decomposed generator matrix storage.

- *One coarse transition as a component:* Every top level coarse transition will serve as a component plus the top level places as the last component.
- *K places as a component:* The first  $K$  places will make up the first component, the second  $K$  places will make up the second component, and so on.

**Analysis operation configuration** Determines the way matrix algorithms will be executed during the numerical solution step.

- *Sequential:* The matrix algorithms will be executed sequentially.
- *Parallel:* The matrix algorithms may be executed in parallel depending on the size of the matrix.

**Analysis type** Determines the system state that will be calculated during the numerical solution step.

- *Steady State:* The steady state of the system will be calculated (i.e. at the time  $t = \infty$ ).
- *Transient:* The transient state of the system will be calculated at the given time  $t$ .

**Algorithm type** Determines the numerical solver used during the numerical solution step of the analysis.

- *Bi-CGSTAB:* Iterative Krylov-subspace method for solving linear system of equations. Usually has a fast convergence but also has a high memory requirement. An error tolerance or a maximum number of iterations can be set as the stopping criteria. Can be used for steady state analysis and as an inner algorithm.
- *Gauss-Seidel:* Iterative method for solving linear system of equations. Usually has a slower convergence but also has a smaller memory requirement. An error tolerance or a maximum number of iterations can be set as the stopping criteria. Can be used for steady state analysis and as an inner algorithm.
- *Jacobi:* Iterative method for solving linear system of equations. Usually has a slower convergence but also has a smaller memory requirement. An error tolerance or a maximum number of iterations can be set as the stopping criteria. Can be used for steady state analysis and as an inner algorithm.
- *Power iteration:* Iterative method for solving linear system of equations. Usually has a slower convergence but also has a smaller memory requirement. An error tolerance or a maximum number of iterations can be set as the stopping criteria. Can be used for steady state analysis and as an inner algorithm.

- *LU decomposition*: Direct method for solving linear system of equations. Calculates the solution with machine precision. Error-prone due to numerical instability. Recommended only for small matrices. Can be used for steady state analysis and as an inner algorithm.
- *Group Gauss-Seidel*: Block matrix based version of the Gauss-Seidel algorithm. An error tolerance or a maximum number of iterations can be set as the stopping criteria. Can be used for steady state analysis. Needs an inner algorithm to calculate the solution.
- *Group Jacobi*: Block matrix based version of the Jacobi algorithm. An error tolerance or a maximum number of iterations can be set as the stopping criteria. Can be used for steady state analysis. Needs an inner algorithm to calculate the solution.
- *Uniformization*: Uses an approximation to calculate the transient state of the system at a given  $t$  time. Its performance is affected by the magnitude of transition rates. An error tolerance plus a maximum number of iterations can be set as the stopping criteria. Preferred method for transient analysis.
- *TR-BDF2*: Integrator algorithm used for calculating the transient state of the system at a given  $t$  time. Its performance is independent of the model parameters but has a high execution time. An error tolerance plus a maximum number of iterations can be set as the stopping criteria. Needs an inner algorithm to calculate the solution.

If anything but the *Run whole workflow* option is selected as the end of the analysis workflow, or there isn't any reward configuration defined for the model then clicking the [Run] button will start the analysis.

#### 4.4.2 Selecting the rewards

If the *Run whole workflow* option is selected as the end of the analysis and there are available reward configurations for the model then clicking the [Next] button will display the window shown in Figure 4.14. The available reward configurations are displayed in the left side of the window. The selected reward configurations (that will be calculated during the analysis) are displayed in the middle of the window. A reward can be selected or unselected using the [→] or [←] buttons, respectively. The available rewards and their corresponding options vary based on the selected analysis type.

The window in Figure 4.14 shows the available settings in case of steady state analysis. When a reward is selected its expected instantaneous value (for the state of the system at  $t = \infty$  time) will be computed automatically. Furthermore the tool can calculate the sensitivity (i.e. the derivative) of this value for the selected sensitivity variables. In case of steady state analysis the composite rewards that reference accumulated reward values are not visible since they can't be calculated.

In case of transient analysis the options for the selected rewards are different, as shown in Figure 4.15. Beside the expected instantaneous value at time  $t$  it is also possible to calculate the expected accumulated value of the reward until time  $t$ . For transient analysis the sensitivity calculation for rewards are currently not supported.

Rewards to Calculate

Available rewards:

- JobComplete
- JobsProcessing
- ServerFailing
- Process Time
- Up Time

Selected rewards:

- Expense
- NoFailedServer

Expense

Instantaneous reward sensitivities:

- incRate
- p
- lbTime
- execTime1
- execTime2
- failRate
- idleFactor
- repairTime
- publicRent

OK Cancel

Figure 4.14: Reward selection form for steady state analysis.

Rewards to Calculate

Available rewards:

- JobComplete
- JobsProcessing
- ServerFailing
- Process Time
- Up Time

Selected rewards:

- Expense
- NoFailedServer

Expense

- Instantaneous reward
- Accumulated reward

OK Cancel

Figure 4.15: Reward selection form for transient analysis.

```

GS Sparse
Info Beginning saturation...
Info Saturation finished.

Info States in MDD: 4 151
Info Nodes in MDD: 33

Symbolic State Space DONE      32

Info Memory dump at 60
Info Before generator:342398320 (410248)

MDD Converter STARTED
Info Converted 32 nodes
MDD Converter DONE      3

EDD Converter STARTED
Info Converted 30 nodes with 4151 states
EDD Converter DONE      4

Symbolic Sparse Generator STARTED
Info Filling in the sparse generator matrix...
Info Computing the diagonal...
Info Sparse generator matrix of size (4 151 X 4 151)
created with 25 650 nonzero elements.
Symbolic Sparse Generator DONE      16

Info Memory dump at 133
Info Before probability vector:344526416 (2128096)

Gauss-Seidel STARTED

```

Figure 4.16: The running analysis.

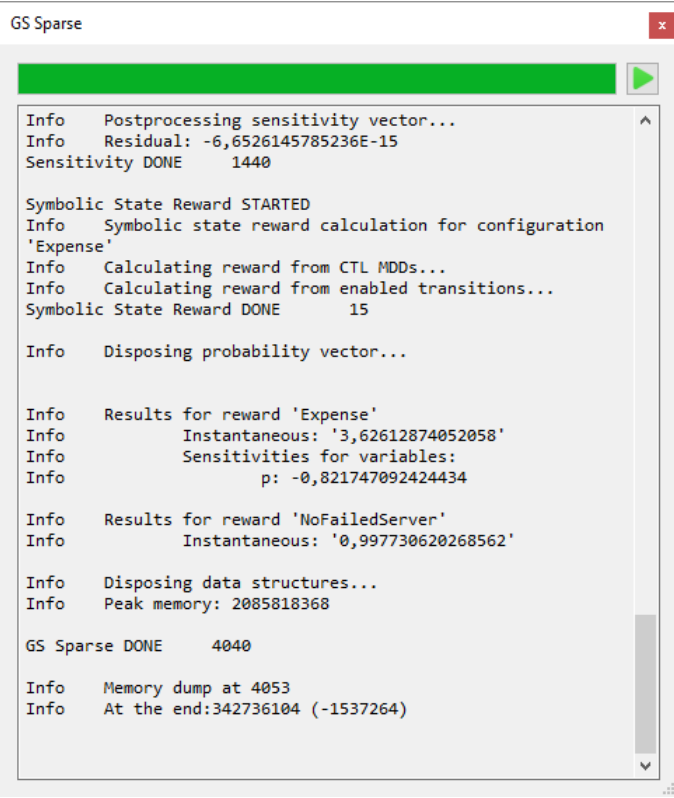
### 4.4.3 Running the analysis

After selecting the rewards to calculate clicking the [OK] button will start the analysis. The progress and logs of the algorithms used during the analysis are displayed to the user as shown in Figure 4.16. Most of the algorithms support interruption thus the analysis can be stopped using the [X] button next to the progressbar. After the analysis the results of the selected rewards and their sensitivities for the selected variables are displayed near the end of the log as shown in Figure 4.17.

## 4.5 Mean Time to First Failure calculation

**TODO:**Describing the MTFE configs





```
GS Sparse
Info Postprocessing sensitivity vector...
Info Residual: -6,6526145785236E-15
Sensitivity DONE 1440

Symbolic State Reward STARTED
Info Symbolic state reward calculation for configuration
'Expense'
Info Calculating reward from CTL MDDs...
Info Calculating reward from enabled transitions...
Symbolic State Reward DONE 15

Info Disposing probability vector...

Info Results for reward 'Expense'
Info Instantaneous: '3,62612874052058'
Info Sensitivities for variables:
Info p: -0,821747092424434

Info Results for reward 'NoFailedServer'
Info Instantaneous: '0,997730620268562'

Info Disposing data structures...
Info Peak memory: 2085818368

GS Sparse DONE 4040

Info Memory dump at 4053
Info At the end:342736104 (-1537264)
```

Figure 4.17: The finished analysis.



## Chapter 5

# CEGAR-based Reachability Analysis

This chapter gives an overview on the usage and capabilities of the CEGAR-based algorithms. These algorithms aim to solve the following two problems:

- *reachability*, i.e., to decide if a target marking is reachable from the initial marking,
- *submarking coverability*, i.e., to decide if a marking satisfying a set of linear predicates is reachable from the initial marking.

Section 5.1 is an excerpt of [4], giving a short overview of the CEGAR approach. More detailed information can be found in [5, 3, 6]. Section 5.2 presents the usage of the tool based on [3].

### 5.1 Overview of the algorithms

#### 5.1.1 Abstraction

Abstraction is a general mathematical approach for solving hard problems. The abstract model has a less detailed state space representation by hiding the irrelevant details. However, due to the abstraction, some action of the abstract model may not be realisable in the original model. In this case, the abstraction has to be refined. This approach is called the “counterexample-guided abstraction refinement” (CEGAR).

#### 5.1.2 CEGAR approach on Petri nets

The CEGAR approach was first described for the reachability analysis of Petri nets by Wimmel and Wolf [11]. Figure 5.1 shows an overview of the algorithm and each step is detailed in this section.

**Initial abstraction.** The input of the algorithm is a reachability problem whether a marking  $m'$  is reachable from the initial marking  $m_0$  in the Petri net  $PN$ . The reachability problem is first transformed into the initial abstraction,

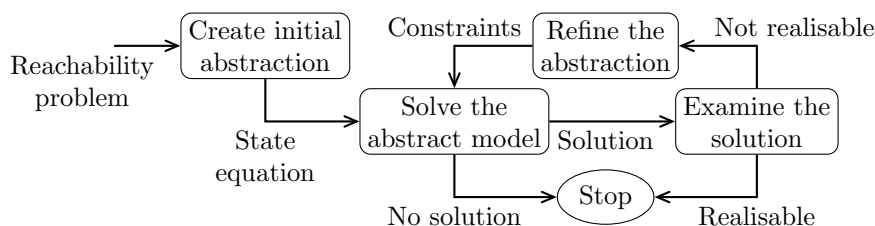


Figure 5.1: Petri net CEGAR algorithm

namely the state equation of the form  $m_0 + Cx = m'$ , where  $C$  is the *incidence matrix* of  $PN$ .

**Solving the abstract model.** Solving the abstract model (i.e., the state equation) is an integer linear programming problem. The ILP solver yields a minimal solution with respect to a cost function. In the original algorithm [11] the sum of the firing count of transitions is minimised in order to obtain trajectories with the shortest length. In our approach, the cost can be set to an arbitrary linear function.

The feasibility of the state equation is a necessary, but not sufficient condition for reachability, therefore if no solution exists, the target marking is not reachable. Otherwise, the obtained solution must be checked whether it is realisable in the original model (i.e., in the Petri net  $PN$ ).

**Examining the solution.** The solution of the state equation is a vector  $x \in \mathbb{N}^{|T|}$ , where  $x(t)$  denotes the number of times a transition  $t \in T$  has to fire in order to reach  $m'$  from  $m_0$ . However,  $x$  does not include any information about the order of the transition firings and whether they are enabled. Thus, the algorithm must explore the state space of the Petri net with the limitation that each transition  $t$  can fire at most  $x(t)$  times. If the target marking  $m'$  can be reached with this limit (i.e.,  $x$  is realisable), it is a sufficient proof for reachability. Otherwise,  $x$  is a counterexample and the abstraction has to be refined.

### Refining the abstraction.

If a solution  $x$  is not realisable, the ILP solver has to be forced to generate a different solution. This can be done by adding additional *constraints* (i.e., linear inequalities over transitions) to the state equation. Each solution  $x$  of the state equation  $m + Cx = m'$ , can be written as the sum of a *base vector* and the linear combination of T-invariants [11]. The following two types of constraints were defined in [11].

- *Jump constraints* have the form  $|t_i| < n$ , where  $n \in \mathbb{N}$ ,  $t_i \in T$  and  $|t_i|$  represents the firing count of the transition  $t_i$ . Jump constraints can be used to obtain different base vectors, exploiting their pairwise incomparability.
- *Increment constraints* have the form  $\sum_{i=1}^k n_i |t_i| \geq n$ , where  $n_i \in \mathbb{Z}$ ,  $n \in \mathbb{N}$ , and  $t_i \in T$ . Increment constraints can be used to reach non-base solutions. This means that a new solution  $x + y$  is obtained, where  $y$  is a T-invariant.

After adding the new constraint, the state equation may become infeasible, or a new solution is obtained. Figure 5.2 presents the solution space. The bottom dots represent base solutions, while the cones represent the linear space formed by the T-invariants. The upper dots correspond to non-base solutions. Jumps are denoted by dashed arrows and increments by continuous arrows.

At each non-realizable solution multiple jump and/or increment constraints can be applied. The three algorithms in PetriDotNet traverse the solution space using depth-first search (DFS), breadth-first search (BFS) and a hybrid strategy [6] until a realizable solution is found, or the state equation becomes infeasible and there are no more solutions to backtrack to.

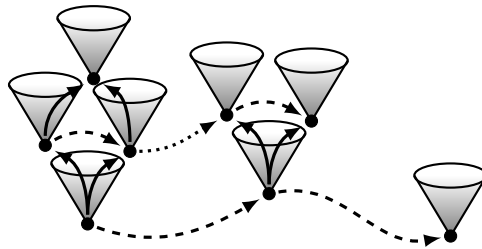


Figure 5.2: Solution space of the state equation

**Extensions.** In our previous work [5] we proved by a counterexample that the original algorithm [11] is incorrect and we suggested a solution to overcome the problem. We also presented several examples where the algorithm could not decide reachability [5, 6]. We extended the set of decidable problems, but the algorithm still lacks completeness [6]. Furthermore, we extended the algorithm to be able to handle inhibitor arcs and submarking coverability problems, and we also introduced new optimization methods in order to improve efficiency [5].

## 5.2 Usage

All three algorithms have the same GUI and capabilities, only the underlying search strategies are different [6]. The algorithms can be started from the “CEGAR-Based Reachability” section of `Add-in / NET ANALYSIS`.

### 5.2.1 Overview of the GUI

The main window of the plug-in can be seen in Figure 5.3. The window is divided into the following three sections:

- information about the net,
- parameters of the reachability problem,
- result of the analysis.

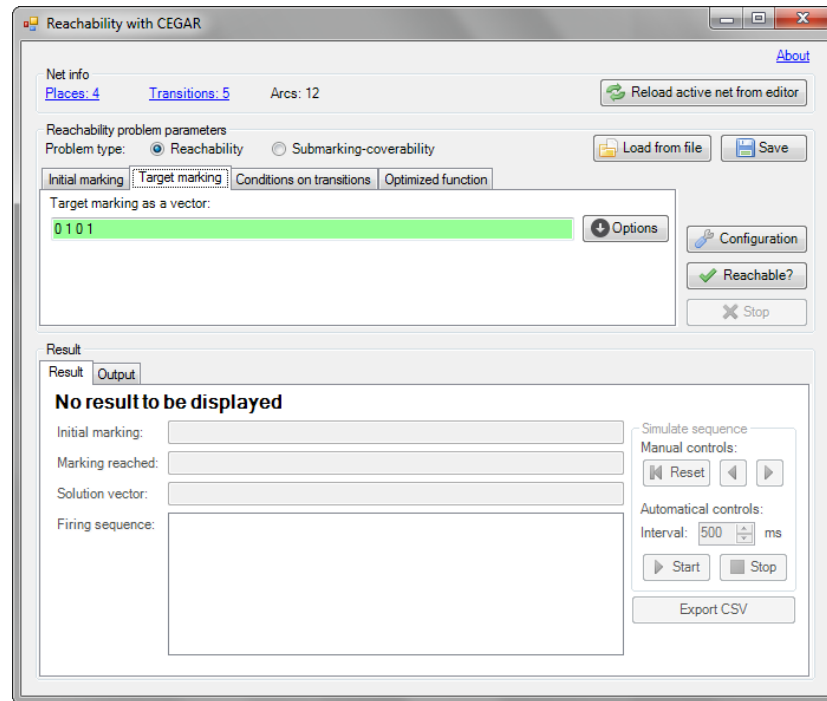


Figure 5.3: Main window of the CEGAR plug-in.

### 5.2.2 Information about the net

The top section displays information about the currently loaded Petri net. If the net is changed in the editor, it must be reloaded manually with the [Reload active net from editor] button. The number of places, transitions and edges is also displayed. If the [Places] or [Transitions] label is clicked, the plug-in displays the ID and name of each place or transition in a pop-up dialogue (Figure 5.4).

ID	Name
0	p0
1	p1
2	p2
3	p3

Figure 5.4: Place names with IDs.

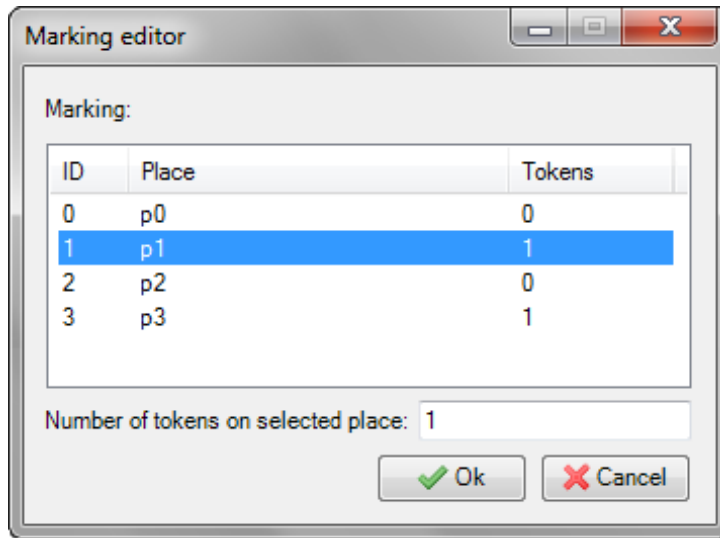


Figure 5.5: Marking editor dialogue.

### 5.2.3 Parameters of the reachability problem

The type of the problem (reachability or submarking coverability) can be set at the top of the section “Reachability problem parameters” with the radio buttons. Each parameter (initial marking, target marking or predicates, conditions on transitions<sup>1</sup>, optimized function) can be edited on a separate tab.

There are several options to enter the initial and target markings:

- They can be entered in the text boxes as a vector of integers separated with spaces. The  $i$ th element of the vector corresponds to the place with ID  $i$ .
- Using the [Options] button they can be edited in a separate window (Figure 5.5) where the ID, name and token count of each place is displayed.
- The actual token distribution of the net can be loaded from or into the PetriDotNet framework with the [Load from PDN] and [Show in PDN] buttons.

If the type of the problem is submarking coverability, the tab for predicates is visible (Figure 5.6) instead of the tab for the target marking. A new predicate can be added with the [Add predicate] button. The dialogue seen in Figure 5.7 helps entering a predicate. The coefficient of each place can be set by selecting the name of the place in the combo box and entering the coefficient in the text box below. The type of the predicate (“ $\geq$ ”, “=”, “ $\leq$ ”) can be set with the combo box in the top right corner. The right-hand side value can be entered in the text box below the type. The selected predicates can be removed with the [Remove] button. The list can be cleared by the [Remove all] button under [Options]. The option [Load tokens from PDN] under [Options] does the

<sup>1</sup>The conditions on transitions are added directly to the ILP problem as a row.

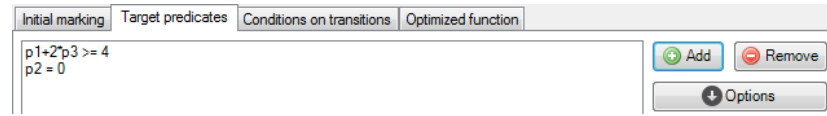


Figure 5.6: Predicates tab.

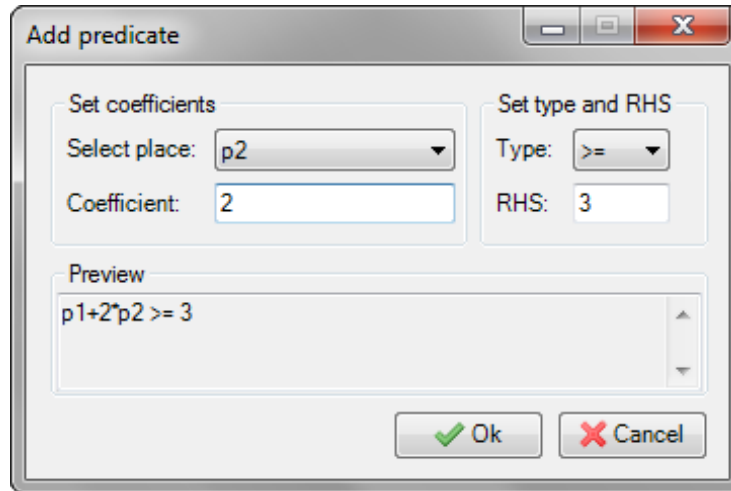


Figure 5.7: Predicate editor.

following: for each place  $p_i$  of the Petri net with  $m(p_i) > 0$  a predicate of the form  $m'(p_i) = m(p_i)$  is created, i.e., places with no tokens are ignored.

The conditions on transitions can be added similarly to predicates. The only difference is that these conditions correspond to transitions instead of places.

The optimized function of the ILP solver can be edited as a vector or with a helper dialogue similar to the marking editor (Figure 5.5).

All parameters can be written into an `.xml` file with the [Save] button and loaded from an `.xml` file with the [Load from file] button.

#### 5.2.4 Configuration of the algorithm

The optimizations and the logging level of the algorithm can be configured by clicking the [Configuration] button. The following options are available (Figure 5.8):

- Level of logging: Sets the detailedness of logging. At level 0, only the solution is displayed, while at level 4, each detail is logged into the “Output” tab of the “Result” section.
- Generate state space: Sets whether the state space should be generated in the file “statespace.dot” in GraphViz format.<sup>2</sup>
- Use stubborn sets: Enables the stubborn set [8] optimization, which reduces the number of potential solutions by investigating dependencies and conflicts between transitions.

<sup>2</sup><http://www.graphviz.org>



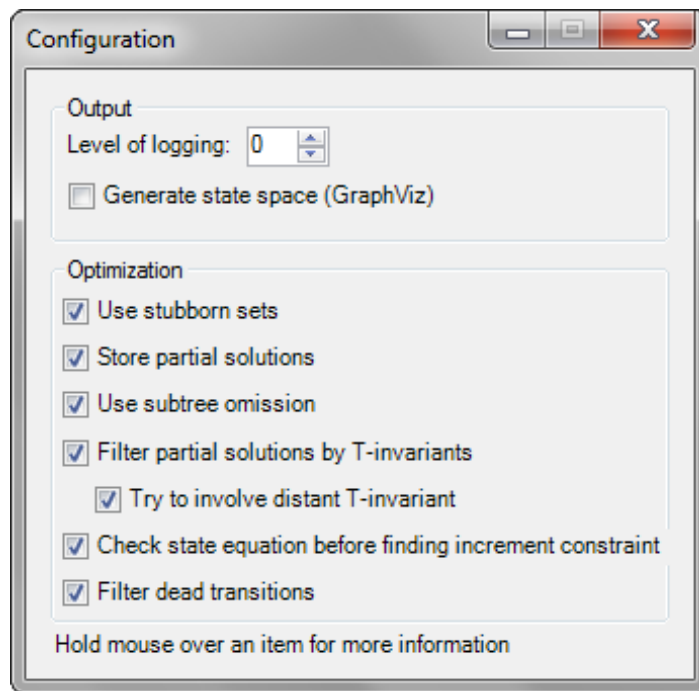


Figure 5.8: Configuration dialogue.

- Store partial solutions: Enables the partial solution storing [11] optimization, avoiding a potential solution to be processed multiple times.
- Use subtree omission: Enables the subtree omission [5] optimization, which reduces the number of potential solutions by ignoring the different order of transitions.
- Filter partial solutions by T-invariants: Enables the T-invariant filtering [5] optimization, which can avoid non-termination by detecting infinite loops in the abstraction refinement.
- Try to involve distant T-invariants: Enables the new extension [6] that tries to involve distant invariants when a potential solution is skipped by the filtering optimization.
- Check state equation before finding increment constraints: Enables filtering based on an extra check of the state equation [5].
- Filter dead transitions: Enables filtering transitions that can never fire at the beginning of the algorithm.<sup>3</sup>

The reachability analysis can be started with the `[Reachable?]` button. It runs on a background thread and it can also be interrupted with the `[Stop]` button.

<sup>3</sup>Developed by Pál András Papp.

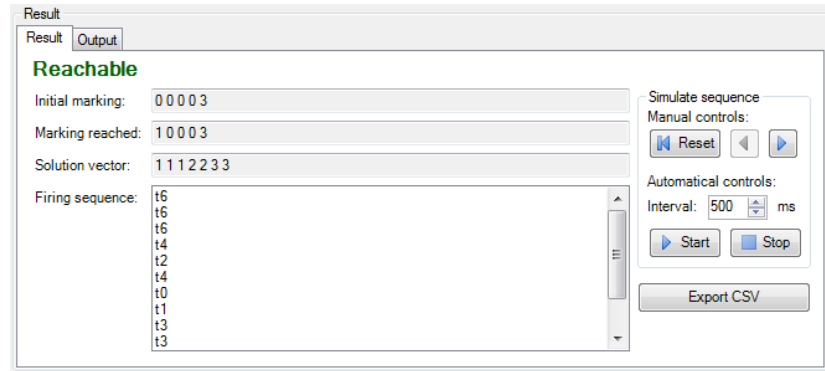


Figure 5.9: Result of a problem, where the target is reachable.

### 5.2.5 Examination of the result of the algorithm

The algorithm prints information depending on the level of logging in the “Output” tab of the “Result” section. When the reachability analysis finished, detailed information can be seen in the “Result” tab. If a realisable solution is found, the plug-in displays the following items (Figure 5.9):

- the initial marking,
- the marking reached by the solution,
- the solution vector,
- and the firing sequence realising the solution.

The firing sequence can be simulated automatically or manually with the playback controls in the “Simulate sequence” group. The result can also be exported into a .csv file with the [Export CSV] button. Clicking the button, a place selector dialogue (Figure 5.10) appears. Each step of the firing sequence is written in the .csv file with the actual marking of the selected places.

If no solution was found, the following cases are possible:

1. If some solutions were skipped by the T-invariant filtering optimization, the result is “Not decidable”.
2. If over-estimation occurred, the result is also “Not decidable”.
3. Otherwise, the result is “Not reachable”.

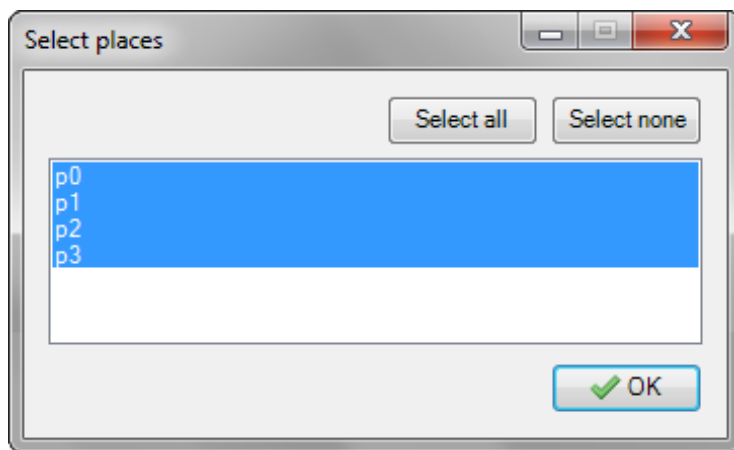


Figure 5.10: Place selector dialogue.



## Chapter 6

# Quick Introduction to Plug-in Development

The functionality of PetriDotNet is extensible with the help of plug-ins. Plug-ins can perform simulation tasks, provide analysis features (e.g. model checking) or export/import capabilities. Each plug-in can access the Petri net data models, use the graphical user interface, add new menu items, and perform PetriDotNet commands. The development of plug-ins is really simple, in order to help the users to focus on functionality instead of technology.

Each plug-in is a .NET class library (.dll) in which at least one class implements the `IPDNPlugIn` interface. The class should have some annotations attached to it:

- `AddinAuthor` that specifies the author of the plug-in,
- `ToolVersion` that specifies the minimum PetriDotNet version needed to execute the plug-in,
- `IncludeInPublicRelease` that specified that the plug-in should be visible in the public releases. Without this annotation the plug-in is loaded only in diagnostic releases and in Debug mode.

The plug-in should hold a reference to the `PetriNetBase.dll` that specifies the base Petri net data model and to the `PetriDotNet.exe` that defines the plug-in interface.

The compiled plug-in .dlls should be placed into the `add-in` folder of PetriDotNet. When PetriDotNet starts it loads all plug-ins from this folder.

**IPDNPlugIn Interface.** The `IPDNPlugIn` interface defines the methods that should be implemented by each plug-in. To help the developers, the defined interface is really simple. It defines one single method: `Initialize(PDNAppDescriptor appDesc)` that is called exactly once after PetriDotNet started.

**Initialization of a Plug-in.** The plug-in should initialize itself in the `Initialize` method. The typical tasks in this method are the following:

- Store the given `appDesc`. This descriptor will provide the mean to interact with the PetriDotNet framework.

- Register the menu items. This can be done by invoking the `appDesc.AddPluginMenuItem` method. This method has two parameters: the first `string` parameter defines the label of the menu item to be placed in the Add-in menu (e.g. "DummyAddin"), optionally together with some submenus (e.g. "DummyGroup1\\DummyGroup2\\DummyAddin"). The second parameter is an event handler delegate of type `EventHandler` that will be invoked when the menu item is selected by the user. The plug-in may register several menu items.

Later, when the user selected the plug-in and the given event handler method is invoked, the previously stored application descriptor (`appDesc`) allows to interact with the framework.

- The `appDesc.CurrentPetriNet` property returns the currently active Petri net.
- Through the `InvokeCommand` method a command (e.g. open, save) can be invoked.

**Example 6.1** See Listing 6.1 for an example plug-in skeleton written in C#.

Listing 6.1: Example PetriDotNet plug-in skeleton

```
using PetriNetBase;
using PetriTool;
using System.Windows.Forms;

[AddinAuthor("John.Doe"), ToolVersion("1.5"),
 IncludeInPublicRelease]
public class DummyAddin : IPDNPlugin
{
    private PDNAppDescriptor appDesc = null;

    public void Initialize(PDNAppDescriptor appDesc)
    {
        this.appDesc = appDesc;
        appDesc.AddPluginMenuItem(
            "DummyGroup\\DummyAddin", Foo);
    }

    private void Foo(object sender, EventArgs e)
    {
        PetriNet pn = appDesc.CurrentPetriNet;
        //...
    }
}
```

# Bibliography

- [1] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. Ten years of saturation: A Petri net perspective. In Kurt Jensen, Susanna Donatelli, and Jetty Kleijn, editors, *Transactions on Petri Nets and Other Models of Concurrency V*, volume 6900, pages 51–95. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29071-8. doi: 10.1007/978-3-642-29072-5\_3. URL [http://dx.doi.org/10.1007/978-3-642-29072-5\\_3](http://dx.doi.org/10.1007/978-3-642-29072-5_3). doi: 10.1007/978-3-642-29072-5\_3.
- [2] Dániel Darvas, András Vörös, and Tamás Bartha. Improving saturation-based bounded model checking. *Acta Cybernetica*, 2014. ISSN 0324-721X. Accepted, in press.
- [3] Ákos Hajdu. Extensions to the CEGAR approach on Petri nets. Bachelor’s thesis, Budapest University of Technology and Economics, 2013. URL <https://diplomaterv.vik.bme.hu/en/Theses/Petrihalok-CEGAR-alapu-vizsgalatanak>.
- [4] Ákos Hajdu, Róbert Németh, Szilvia Varró-Gyapay, and András Vörös. Petri net based trajectory optimization. In *ASCONIKK 2014: Extended Abstracts. Future Internet Services*, pages 11–19. University of Pannonia, 2014.
- [5] Ákos Hajdu, András Vörös, Tamás Bartha, and Zoltán Mártonka. Extensions to the CEGAR approach on Petri nets. *Acta Cybernetica*, 21(3): 401–417, 2014.
- [6] Ákos Hajdu, András Vörös, and Tamás Bartha. New search strategies for the Petri net CEGAR approach. In Raymond Devillers and Antti Valmari, editors, *Application and Theory of Petri Nets and Concurrency*, volume 9115 of *Lecture Notes in Computer Science*, pages 309–328. Springer, 2015.
- [7] Attila Klenik and Kristóf Marussy. Configurable stochastic analysis framework for asynchronous systems, 2015. URL <https://tdk.bme.hu/VIK/DownloadPaper/Aszinkron-rendszerek-konfigurarhato>. 1st prize.
- [8] Karsteb Schmidt. Stubborn sets for standard properties. In *Application and Theory of Petri Nets*, volume 1639 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 1999.
- [9] András Vörös, Dániel Darvas, and Tamás Bartha. Bounded saturation-based CTL model checking. *Proceedings of the Estonian Academy of Sciences*, 62(1):59–70, 2013. ISSN 1736-6046. doi: 10.3176/proc.2013.1.07.

- [10] András Vörös, Dániel Darvas, Attila Jámbor, and Tamás Bartha. Advanced saturation-based model checking of well-formed coloured Petri nets. *Periodica Polytechnica, Electrical Engineering and Computer Science*, 58(1): 3–13, 2014. ISSN 2064-5279. doi: 10.3311/PPee.2080.
- [11] Harro Wimmel and Karsten Wolf. Applying CEGAR to the Petri net state equation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2011.