

Contents

1	Introduction to Timed-Arc Petri Nets	2
1.1	Petri Net	2
1.2	Timed Tokens and Timed Arcs	6
1.3	Invariants on Places	8
1.4	Transport Arcs	10
1.5	Queries about TAPN Models	10
2	TAPAAL Manual	13
2.1	User Interface	13
2.1.1	Menu bar	14
2.1.2	Tool bar	14
2.1.3	Drawing area	15
2.2	Drawing a TAPN	16
2.3	Verifying the Model	17

Chapter 1

Introduction to Timed-Arc Petri Nets

In this section Petri Nets (PN) and Timed-Arc Petri Nets (TAPN) will be described, starting with the basic components and features of the simple Petri nets. One by one we introduce the more advanced features, which include *invariants* and *transport arcs*, that are extensions of TAPN.

If you are already familiar with PN, you might want to jump directly to Section 1.2, where the TAPN is introduced.

1.1 Petri Net

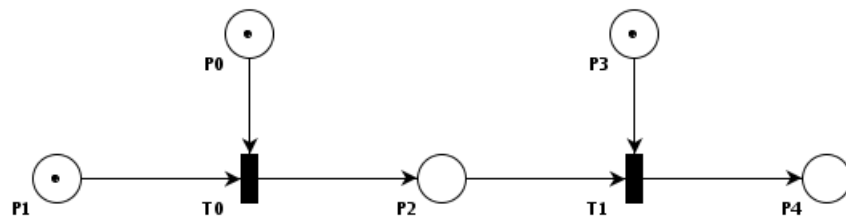
A Petri Net is a directed bipartite graph of *places* (circles), *transitions* (squares) and *arcs* (arrows) as illustrated in Figure 1.1(a) and (b).

In Figure 1.1(a) small dots are placed within the places **P0**, **P1** and **P3**. These dots are called *tokens*. The behaviour of a given PN is defined by the way that tokens are placed.

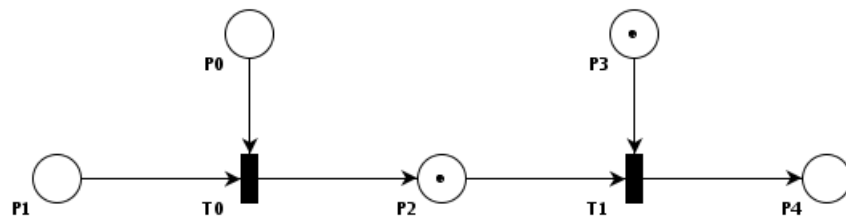
A configuration of a given PN is a distribution of tokens on places and is referred to as a *marking* of a given PN.

We say that a given transition is *enabled* if all the places that have arcs going to the transition, referred to as a transition's *preset*, has at least one token. When a transition is enabled it can be *fired*. This means that **T0** in Figure 1.1(a) can be fired. In Figure 1.1(b) **T0** has been fired. A firing of a given transition will consume a token from each place in its preset and produce a token in each place of its *postset*. The postset of a transition is the places that have an arc going from the transition to it self. We see that the places **P0** and **P1** lose one token and **P2** receive one.

Example 1.1. Say we want to model a production line with producers and



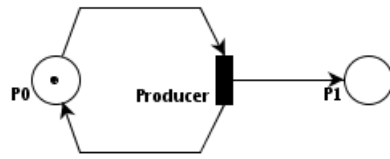
(a) A PN with a token in places P0, P1 and P3.



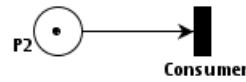
(b) The PN from Figure (a), after transition T0 has been fired.

Figure 1.1: Firing of a transition in a PN.

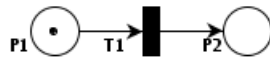
consumers using a Petri Net. Figure 1.2(a) illustrates how to make a producer. This part is able to produce an arbitrary number of tokens. A PN part that consumes tokens is illustrated in Figure 1.2(b). It is simply a transition with no outgoing arcs. In Figure 1.2(c) we have a PN that "moves" a token through, what could be, a production line. In Figure 1.2(d) we have com-



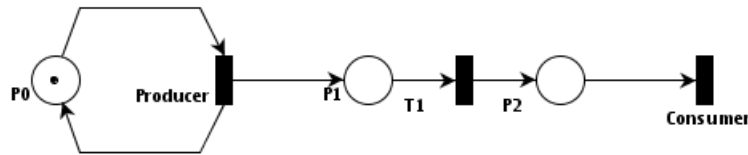
(a) A token producing PN.



(b) A token consuming PN.



(c) A token transporting PN.



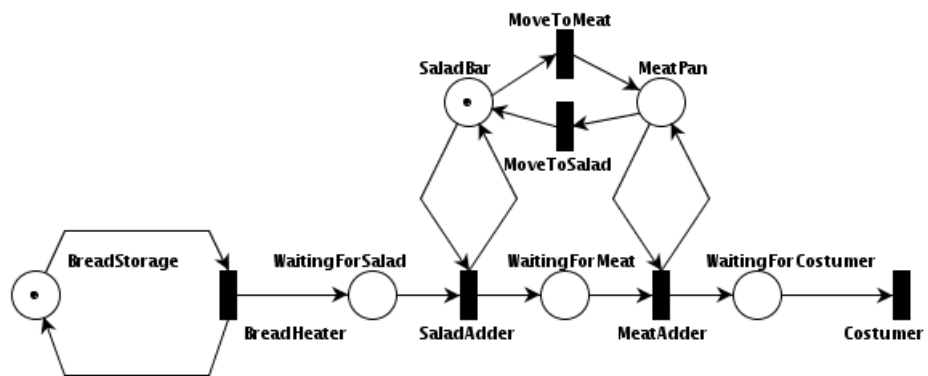
(d) Combined producer consumer with a transporting part in between.

Figure 1.2: Combining different PN modules.

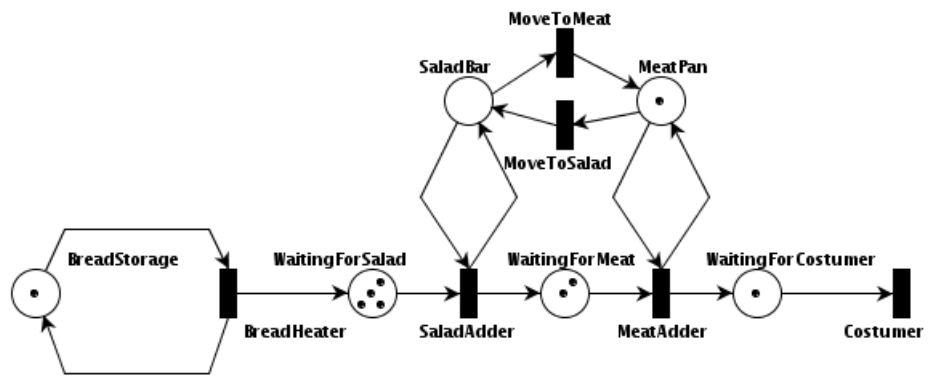
bined the three small nets into a simple producing and consuming production line.

This example could be the production line of a fast food restaurant. The producer would be heating bread, and the line would be treatments of the bread. Meat will be added and salad could be added. We could also model an assembly worker as illustrated Figure 1.3(a). Meat and salad are added at different places, but by the same assembly worker, who has to move from place to place to assemble a burger from the heated bread.

This model can behave differently from what we would expect from a production line. This is because we can choose to fire arbitrarily amongst



(a) A PN modelling a fast food production line.



(b) Behaviour of a modelled fast food production line.

Figure 1.3: Possible behaviour of a modelled fast food production line.

the enabled transitions — e.g. bread could stack up in different places of the production line as illustrated in Figure 1.3(b).

In some fast food restaurants burgers are produced in advance, so that costumers can be served as fast as possible. However these restaurants have to guarantee that the food that the consumers get is not too old. To model this we have to introduce some notion of age to the tokens in the net, so that we can check the age of the produced tokens and ensure that only fresh burgers get consumed.

By introducing time we can also prevent the stack up of bread, by describing that bread with a certain age cannot be added meat or salad, but should maybe be reheated. \triangle

The PN we have studied so far have introduced unbound behaviour. This is because every place can stack up any number of tokens. In Figure 1.4 we illustrate how one can restrict the number of tokens at a single place. By adding a new place, often called the capacity place, and adding arcs from the capacity place to the transitions producing tokens in the specified place and adding arcs from the transitions consuming from the place, it is ensured that the number of tokens in the given place cannot exceed the number of tokens originally placed in a token and its capacity place.

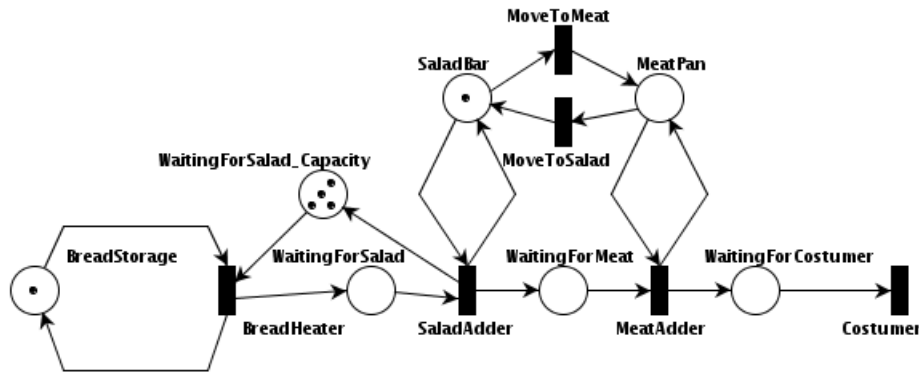
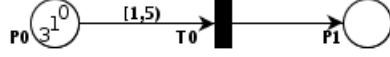


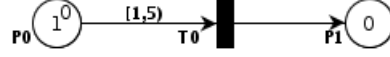
Figure 1.4: An example of a PN, where P1 is restricted to a capacity of four tokens.

1.2 Timed Tokens and Timed Arcs

In Figure 1.5 a small TAPN is illustrated. Here we see that arcs going from places to transitions carry guards. Tokens each have their own clock or age,



(a) A small TAPN.



(b) The net from (a), where $T0$ has been fired.

Figure 1.5: A small TAPN illustrating age of tokens and resets when transitions are fired.

which must conform with the guard before it can be consumed — being enabled is no longer enough for a transition to consume tokens. When time passes in the net all tokens are aged. In Figure 1.5(a) the arc going from place $P0$ to $T0$ has a guard specifying that only tokens of age 1 to 5 exclusive are allowed to be consumed. A consumed token will have its age reset to 0, which is what Figure 1.5(b) illustrates; here the token of age 3 has been consumed and its age reset to 0.

This is the components that Timed-Arc Petri nets (TAPN) consist of. Extensions that follow are further extensions to the TAPN.

Example 1.2. Now let us add time to our fast food production line. In Figure 1.6 we have illustrated a more advanced fast food production line, which model that the assembly worker will use time assembling different parts of the burgers. Heating a piece of bread takes more than 1 time unit, hence the guard $(1, \text{inf})$, which specifies that the age of the token in **BreadStorage** should be more than 1 time unit. To add salad to the heated bread the assembly worker has to move to the **Saladbar** place to add salad to the bread. If the bread waits for more than 2 time units it is too cold to be added salad, and is instead reheated. Adding salad takes at least 1 time unit. After salad have been added the burger needs heated meat. It takes more than 1 time unit to heat the meat meanwhile the burger is only allowed to cool down for about 2 time units — also the assembly worker will have to go to the **MeatPan** to assemble this part of the burger. If the burger waits for 2 or more time units it is too old and meat cannot be added and so it can only be thrown out. It should not be reheated since there have been added salad. After meat has been added the assembly worker will add cheese in the same matter as meat and salad was added. Now the burger is finished and is ready to be consumed by a costumer. If the burger is ready for 4 or more time units cannot be sold, and so it will eventually be thrown out.

From this example we have an intuition that it is possible to make burgers, but we would like to be sure i.e. is it possible to make infinitely many burgers?

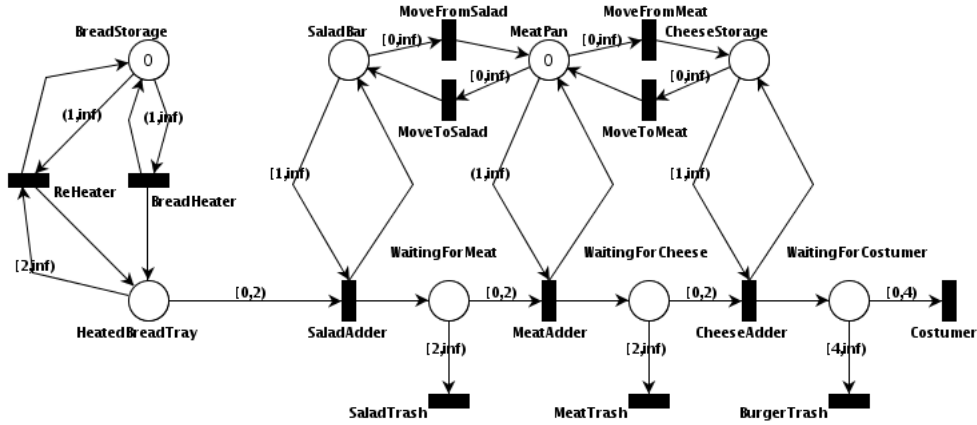


Figure 1.6: A modelled fast food production line with a single bread heater and a single assembly worker, that each uses time to do their tasks.

It would also be nice to know how fast it is possible to make a burger? Or, how many burgers are consumed before a burger is thrown out?. To keep track of that we will have to add places that collect the sold burgers and the thrown out burgers.

If we would model that e.g. bread cannot wait more than a fixed number of time units (else it starts growing mold), we would have to introduce some new construct dictating that tokens of a certain age have to move away from a given place. \triangle

1.3 Invariants on Places

Invariants are restrictions to the age of tokens in places. Invariants make sure that tokens are consumed if possible, when the tokens reach a certain age. If a token reaches the bound of an invariant, time cannot pass any more. The only actions possible are firing of transitions. In Figure 1.7 P_0 has an invariant that ensures that the token is consumed before it is at age four. The guard on the arc make sure that the arc is not consumed before it has age one. Even though the arc allows the token to get age five, the only behaviour possible for this net is to fire T_0 when the age of the token is between 1 and 4 (exclusive).

Example 1.3. Returning to the fast food production line — in Figure 1.8, we have added invariants on `WaitingForMeat`, `WaitingForCheese` and `WaitingFor-`

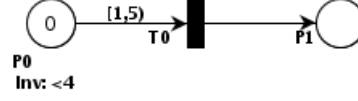


Figure 1.7: A Petri net with a place **P0** with an invariant that ensures that **T0** is fired before the token reaches age four.

Costumer so that we ensure that if bread is to old to go further into the production line it will eventually get thrown out.

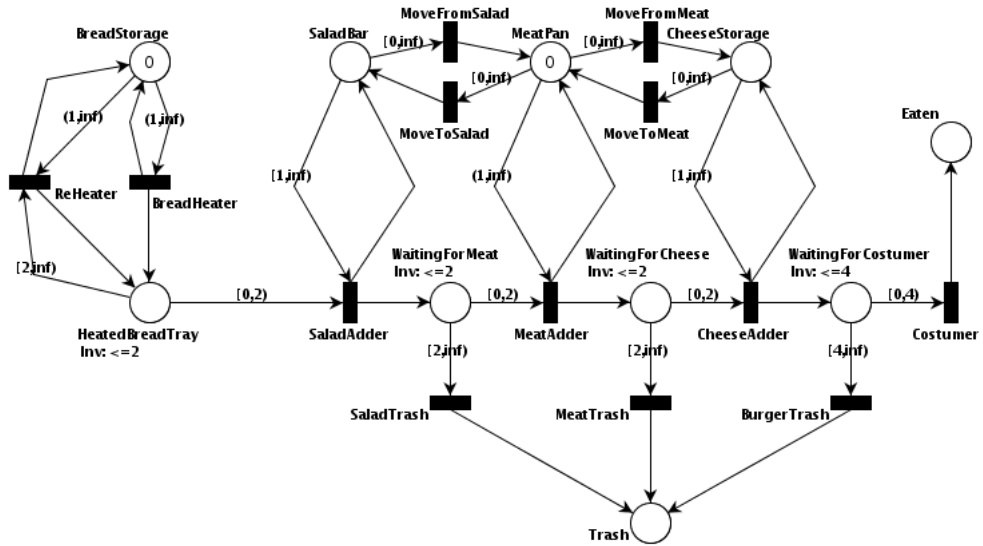
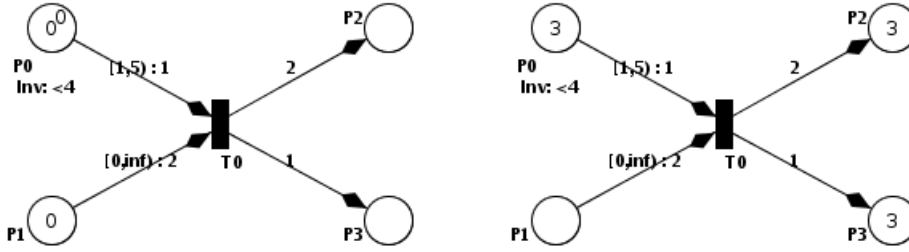


Figure 1.8: A Petri net model of a fast food production line that is hygienic. Invariants ensure that old food is thrown out.

Now we have added **Trash** and **Eaten**, which makes it possible to check how many burgers are thrown out and how many are eaten. However it would be nice to know how old the burgers are before they are eventually thrown out or sold. To do this we will have to bring the notion of transporting the tokens and not consuming and producing tokens when firing a transition. \triangle

1.4 Transport Arcs

Transport arcs are arcs that give the notion of transporting tokens and so preserve their age. Transport arcs have guards just like timed arcs, but instead of consuming a token and producing a new one with age zero, transport arcs ensure that the produced token has the same age as the consumed token. Transport arcs are paired so that the token consumed will be produced in the correct place with the correct age. In Figure 1.9(a) one transport arc pair is marked by 1 and another is marked by 2. In Figure 1.9(b) time has elapsed 3 time units and T_0 has been fired.



(a) A small Petri net with transport arcs at age zero.

(b) A small Petri net with transport arcs at age three and transport arc pair 1 has fired.

Figure 1.9: A small Petri net with transport arcs.

We end this introduction with an example of a full blown Extended TAPN modelling a fast food production line.

Example 1.4. In this example of the fast food production line we are able to keep track of the age of the burger (since the bread was heated) thanks to the transport arcs as illustrated in Figure 1.10. Now to check if burgers of certain ages are thrown out or sold, we need a way to express questions about the behaviour of the model.

△

1.5 Queries about TAPN Models

To verify the existence of some special behaviour in a TAPN, we need a way to formulate these questions. We do this by using queries. In TAPN we can ask atomic questions of the type: Does some number compare to the number

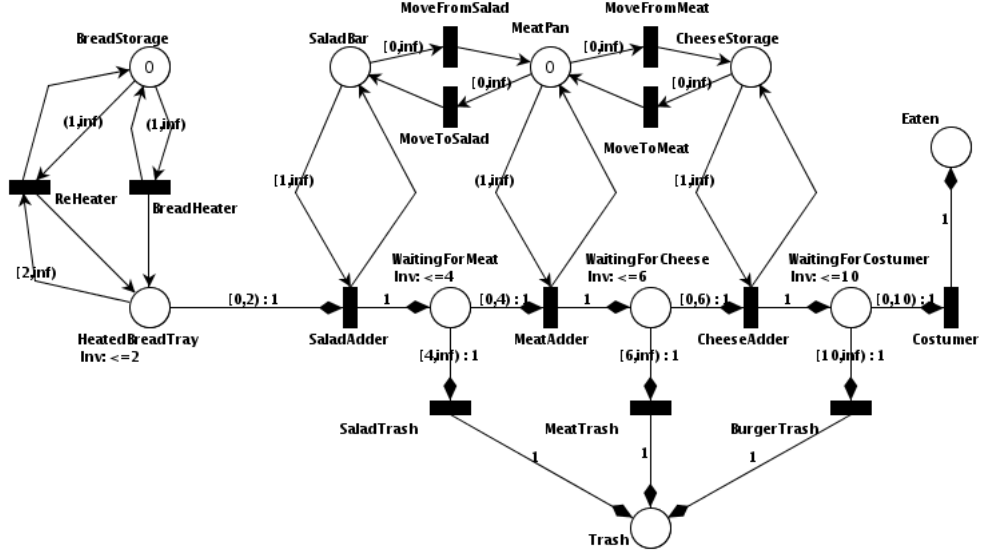


Figure 1.10: An Extended TAPN model of a fast food production line that is hygienic and traceable.

of tokens in some place? e.g. are there more than 5 tokens in P_0 . These ‘frases’ can be combined using logical *and* and *or*.

To ask questions of the behaviour of the model, we introduce 4 special, questions (inspired by the tool Uppaal):

- To ask the question if something is possible, we use a query $\exists \Diamond \Phi$. This query asks if there are some computation s.t. the atomic question Φ is true at some marking of the computation,
- To ask the question if some property potentially always hold, we use query, $\exists \Box \Phi$, if there is a computation s.t. for all steps in this computation the property Φ is satisfied.
- To ask questions of the type, will something eventually happen, we use the query, $\forall \Diamond \Phi$, meaning that for all possible different computations, the property Φ must at some point be satisfied.
- The last form of question is invariantly true. We use the query $\forall \Box \Phi$, meaning that for all possible different computations each reachable marking has to satisfy Φ .

We are now able to ask questions about the behaviour of a TAPN.

For example we might want to know if we can make burgers with the production line modelled in Figure 1.10. So we would pose the query: $\exists \Diamond \text{Eaten} \geq 1$? Or we would like to know if we can produce a specific number of burgers, e.g. 5 burgers, without throwing any out: $\exists \Diamond \text{Eaten} = 5 \text{ and } \text{Trash} = 0$?

Chapter 2

TAPAAL Manual

This is the introduction to the TAPAAL.NET, it includes an introduction to the interface and how to export a model, such that it can be verified using Uppaal.

This it meant to be a manual that can be disturbed together with the tool when it is released.

If you work or study at Aalborg University you can can download a pre-relace of TAPAAL from URL www.cs.aau.dk/~kyrke/tapas.

2.1 User Interface

TAPAAL.NET has a simple, easy to overview user interface. The user interface is presented in Figure 2.1.

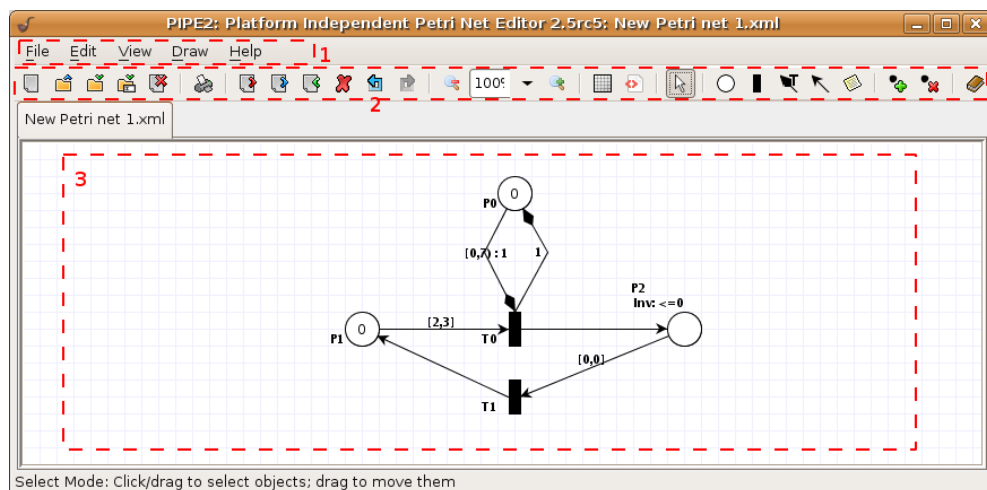


Figure 2.1: The TAPAAL.NET interface

The user interface consist of tree main areas, marked **1.** the menu bar, **2.** a tool bar, and **3.** the drawing area.

2.1.1 Menu bar

The menu bar contains all the functions that a offered by TAPAAL.NET. The *file menu* is mainly used for save and loading TAPAAL.NET files. The file menu also contains a submenu “*Export*” for Exporting the modelled TAPN:

- *PNG*
Export the TAPN as a PNG image file.
- *PostScript*
Export the TAPN as a Post Script file.
- *Export to Uppaal*
Export the TAPN as a Uppaal model, using the naive reduction.
- *Export to Uppaal, Advanced*
Export the TAPN to Uppaal model, select the reduction method advanced (Mainly a developer feature).
- *Export to Uppaal, Symetric*
Export the TAPN as a Uppaal model, using the naive reduction with symmetric reduction.
- Export a degree-2 net
Export a net into degree-2, and have it open in TAPAAL.NET for visual inspection (Mainly a developer feature).

The menu “*Edit*” contains function used when editing the model, this include undo/redo and copy/paste. (Notice that at the moment only copy/paste of Places works).

The “*View*” menu is used for manipulating the appearance of the drawing area, by enabling grid drawing and zooming.

The “*Draw*” menu us used for drawing the TAPN. Most of the functions found here are also accessible in the tool bar. The meaning of the different buttons are explained in the following sections.

2.1.2 Tool bar

In Figure 2.1 the tool bar is presented. The tool bar is divided into **1.** a file bar, **2.** an edit bar, **3.** a zoom bar and **4.** a draw bar:



Selection 1. shows the file part of the bar. It contains the usual features as new Petri net, open Petri net, save Petri net, save Petri net as and close Petri net tablet.

Selection 2. shows the edit part of the bar and contains buttons: cut selection, copy selection, paste selection, delete selection and undo/redo buttons. (Notice that for the moment only copy/past of Places works).

Selection 3. shows the zoom part of the bar, it is possible to zoom in and out, change grid size and move grid.

Selection 4. shows the drawing part of the bar. It includes buttons to draw, transitions, places and arcs. The last two buttons are used for placing and removing token. The meaning of the different symbols are explained in the following sections.

Selection 5. shows the Uppaal export button on the tool bar used for generating a Uppaal model based on the modelled net.

2.1.3 Drawing area

The drawing area is the place where the TAPN can be drawn. In Figure 2.2 we see a part of the drawing area.

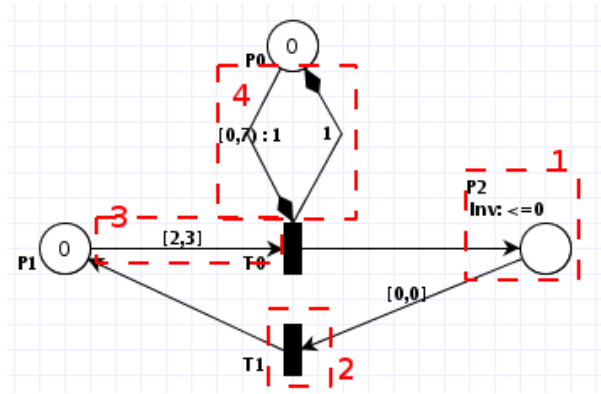


Figure 2.2: The drawing area of TAPAAL.NET.

Selection 1. shows a *Place* with its name and its invariant displayed. The invariant is only displayed if it differs from the default invariant

$< \infty$. A number drawn in the place, marks that there is a token of that age in the place. By holding the mouse over the place, you will get a list of all tokens and there age.


Selection 2. shows a *transition*, and its name.

Selection 3. shows an arc from a place to a transition. The value “[2, 3]” on the arc, is the guard.

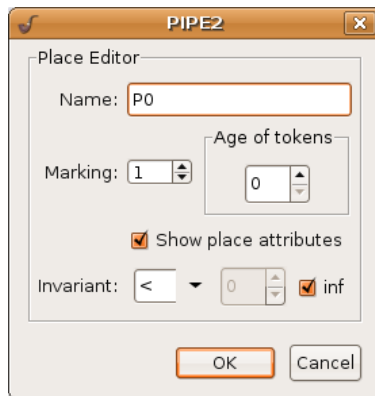
Selection 4. shows a transport arc. The value 1 on the arcs indicates that these two transport arcs are connected.

2.2 Drawing a TAPN


To draw a TAPN use the tools in the drawing tool bar.


Places To draw a place use the  button, and click in the drawing area where you wish to place the place.

By double clicking on the place you will get the menu:

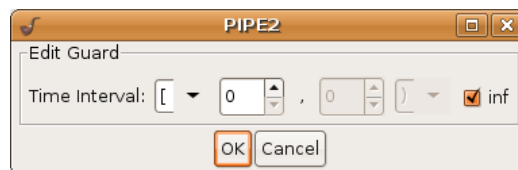


where the name, number of tokens, the age of tokens and the invariant of the place can be changed. By scrolling on the place, tokens can be added/removed.


Transitions To draw a transition use the  button and click on the drawing area where you wish to place the transition. To change its name double click it, and change the name. By scrolling on the transition the angle of the transition is changed.

Arc To draw an arc use the  button and click on the place/transition where the arc should start from and click again on the place/transition where the arc should point to. You are of coarse not allowed to connect two places or transition with one arc.

To edit the guard right click on the desired arc and choose “Edit Time Interval” and the following window will pop up:




When the “inf” check box is checked there is no upper bound to the time interval. Otherwise the square parenthesis means including, and the soft parenthesis means not including.

Transport Arc To draw a transport arc use the  button and click on a place from where the arc starts, now click on a transition followed by a place.

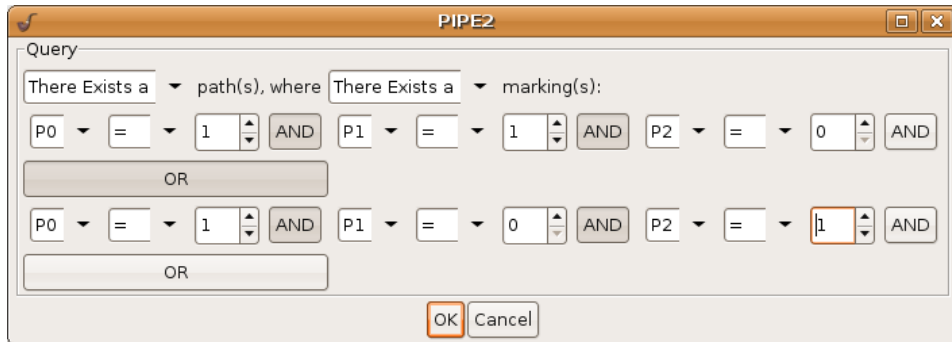
You can change the guard of a transport arc in the same way as on a normal arc.

2.3 Verifying the Model

Verification of the model at the moment consist of two steps: creating the Uppaal model and verifying it using Uppaal.

To create the Uppaal model press the  export to Uppaal button in the menu bar. (To use one of the implemented optimisation export by selecting “File->Export->the-export-variant-you-want”.)

Enter the number of extra tokens and the query in the popup window press enter and enter your query into the graphical query designer:



By pressing an “AND” button the window will expand sideways, and likewise if an “OR” button is pressed the window will expand vertically and a new row of “AND”’s is displayed. Press ok.

Now select a filename for the Uppaal model and press save. TAPN now generates the Uppaal xml and q files.

The model has been saved as a Uppaal NTA. Open the generated .xml file in Uppaal, run the verification and wait for an answer.