Distributed Object Enhanced Real Time Petri Nets (OER-TPNs) Models

1. Laboratory Objectives

* Acquiring the concepts
  + Understanding of OER-TPN theory and framework,
  + implementing OER-TPN task migration,

1. Theory

*A. Object Enhanced Real Time Petri Nets*

The Object Enhanced Real-Time Petri Net (OER-TPN) that is a modification of the previous OER-TPN (Object Enhanced Time Petri Net) [1]. The modification allows the adding of new features for:

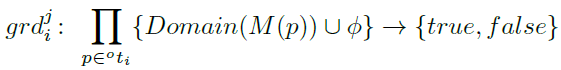
* describing moving objects that carry temporal information related to time such as speed, acceleration or deceleration,
* modeling the changing of state involved by dynamically created temporal information and static temporal information,
* describing of activities having associated deadlines and \_ detecting the missing deadline.



**Figure 1 Example of an OER\_TPN model**

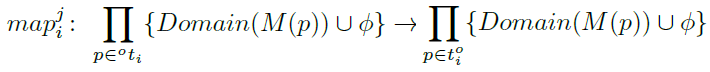
The OER-TPN models are obtained from the Object Enhanced Time Petri NETs (see [1]) endowing them with variable time intervals for the transition executions. The earliest and latest execution time of transitions can be calculated from information stored in the tokens of the transition activated input places. This information related to timed behavior (such as speed, acceleration, deceleration, etc.) allow the specification and verification of R-T requirements or properties. An example of an OER–TPN model is presented in Figure 1. The following notations are used:

* *N* = (*P*, *T*, *F*) is a net with two kinds of disjoint node sets
* a finite place set *P* = {*p1*, *p2*, …, *pm*}, (*m* ≥ 0),
* a finite transition set *T* = {*t1*, *t2*, …, *tn*}, (*n* ≥ 0),
* *F* ⊆ *P* × *T* *T* *P* flow relation,
* *ot* = {*p* ∈ *P*|(*p*, *t*) ∈ *F*} the transition t input place set
* *to* = {*p* ∈ *P*|(*t*, *p*) ∈ *F*} the transition t output place set
* *Inp = {pi1,…}* ⊆P, are external inputs that inserts tokens into the net,
* *Out = {po1,…}* ⊆P, are output channels that extracts tokens from the net sending them to channet destination tokens,
* *Types* = {*Class1*, *Class2*, …} represents the set of software classes of the net,
* *type*: *P* → *Types* is a mapping that assigns to each place a class (i.e. type), (the mapping is of *type(p)* is a class*)*
* a guard of a transition *ti* denoted by is a mapping:

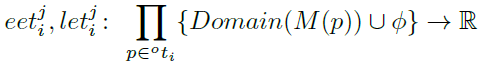
****

where true and false are the values of the boolean set. ϕ denotes the empty set. The set of all guards assigned to a transition *ti* is denoted by *grdi*, and **Grd** denotes the set of all the guards of the net.

* A transition *ti* mapping denoted is :



* *Domain*(*M*(*p*)) defines for the token that can be set in the place *p* the set (possible very complex and infinite) of the values assignable to the object’s attributes. All the mappings of a transition *ti* are included in the set *mapi* and **Map** represents the set of all the mappings of the net.
* a transition *ti* earliest execution time (delay) latest execution time (delay) are mappings ,



\mathbb{R}means the set of real numbers considering the time as being dense. Often the set \mathbb{N} of the natural numbers is used when the time is accepted as being discrete and it is measured with the clock accuracy. The set of all earliest execution time mappings assigned to a transition *ti* is denoted by *eeti*, and **Eet** denotes the set of all the delay mapping sets of the net. The set of all latest execution time mappings assigned to a transition *ti* is denoted by *leti*, and **Let** denotes the set of all the delay mapping sets of the net.

The definition of an OER-TPN is:

OER – TPN = (P, T, pre, post, Inp, Out, D, *δ, Types, type,*

***Grd****,* ***Map****, Λ, M, init, end)* (1)

Where

* Λ = {*λ*t|*t* ∈ *T*, *λt*⊆ *grdt*× *mapt* × *eett* × *lett*} is a set of relations *λt*= {(, , *eeti, leti*)} that assigns to each transition *t* tupples composed of a guard, a mapping, an *eet* and *let* from the sets *grdt*, *mapt*, *eeti,* and *leti.*
* **M** is the net vector marking with *M*(*p*) the marking of the place p that identifies the object token of the type (p).
* *init* is a function that initializes the net state (i.e. the markings) with values from the creators domain and
* *end* is a function that terminates the OER-TPN model execution and provides the creator processed information.

*b. The Framework of Object Enhanced Real Time Petri Net*

Each OER-TPN is executed by a separate thread called *executor*. When a parent creates a child, the offspring is executed by another concurrent executor until it is an active object. A distributed platform implements distributed executors that can communicate through the network channels. A platform has specified a set of channels linked to the specified input or output places. Some channels can link outside the computer devices such as printer, screen etc.

When a model external event occurs, the model executor is immediately signaled. When the model executor sets a token into an output place, this is immediately signaled (or sent) to destination (i. e. another OER-TPN or device).

The executor algorithm of OER-TPN is run with the period of 1 time unit (t.u.) or when an external event is signaled as loading an input place with a token. The algorithm updates the places of the input set and determines the transitions that are enabled taking into account the markings of the transition’s input place set. If a transition is chosen to be executed, the tokens from its input place set are removed and set into a temporal marking vector *Mt*. A time counter *Delay[ti]* is loaded with the transition assigned delay. If the variable value is missing (i.e. ) or is negative the delay is set to 0.

If the time counter is zero or it reaches the value 0 (after decreasing), the execution of the transition is finished. If a transition sets tokens to the output set *Out*, its execution is signaled to a driver that sends the information to the linked destination. The counters of all the started transitions are decreased after each sample period.

*OER-TPN executor algorithm:*

*Input:* ***Pre****,* ***Post****, M0, P, T, D, Grd&Map,* Out, Inp;

*Initialization: M = M0; execList = empty;*

**repeat**

*wait(event);*

**if** event is *tic* **then**

\* decrease the Delays of the transitions in *execList*;

**else**

*receive(Inp);*

\* update **M**;

**end if;**

**repeat**

**for** all *ti 2 T* do

**if** there is met at least one *grd* in the *ti Grdi* list

**for** M(p), *p* ∈ *oti*, **then**

\* move the tokens of *oti* from **M** to *Mt*;

\* add *ti* to *execList*;

*Delay[ti] = δ (ti);*

**end if**;

**end for**;

**for** all *ti* in execList do

**if** (*Delay[ti]* is 0) **then**

\* remove *ti* from *execList*;

\* calculate the tokens for **M** in ;

\* remove the tokens from *Mt* for all *oti*;

\* set the tokens in and start the active tokens;

**end if**

**if** ti ∈ *Out* **then**

*send(Out);*

**end if**

**end for**

**until** there is no transition that can be executed;

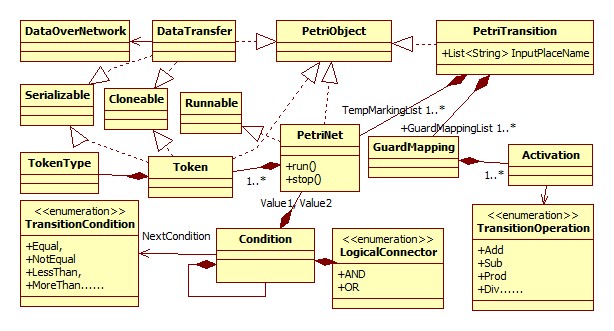
**until** the time horizon;

**END** algorithm;

Each OER-TPN is an object that contains a list of places and a list of transitions. The transition must have a list of GuardMapping containing the guards which is an instance of the Condition class that is connected to the next condition with the LogicalConnector (AND, OR), the Condition class compares between two values (tokens). The mapping is activated when the transition’s condition is true, it is an instance of the Activation class that contains the TransitionOperations that are performed on the tokens to calculated the output places of the transition. The marking is a list of places, the place type is either passive (objects), active (sub OER-TPN s), or output channels of type DataTrasnfer. The requirements for creating a new passive token type are, adding transition conditions and transition operations to implement the guards and mappings that are specific for each token type. The newly defined token types must implement the PetriObject interface so they can be recognized as Petri Net objects and treated as such.

The OER-TPN that has sub OER-TPNs is called a parent and the sub OER-TPNs are called children, the parent and the child are executed concurrently on different threads. The parent and the child can communicate through the marking of the token that is set in the parent place where the child is created. The parent creates the structure, the guards and the mapping of the child, when the child execution ends, its marking is accessible to the parent.

Every OER-TPN must have a unique name, a network port and an IP address so OER-TPNs can communicate with each other. The DataOverNetwork represents the message form between OER-TPNs. NetworkListener class is responsible for starting a thread that listens to the network for the input data from other OER-TPNs. Fig. 2 shows the class diagram for the OER-TPN system.



**Fig. 2 The class diagram for the OER-TPN system**

The executer algorithm is in the OER-TPN class run method, it runs with the period of 1time unit or when data is received in the input channels, when a 1 time unit elapses, the delay is decreased by one in every transition in the execution list, and when it reaches 0, the transition is executed.

*C. The transition, guard and map synthesis*

Each OETPN has a list of guards, each guard has a list of subguards, each subguard has a list of predicates and a list of mapping. If all the predicates inside one subguard is true then the transition is enabled, and the mapping of the respective subguard is executed. See figure 3. For example, if we want to represent the following guard: (M(p1)!=null) AND (M(p2)<1); map1: M(p3)=M(p2); the first predicate (M(p1)!=null) is the first item of the subguard1.predicate list, and the second predicate (M(p2)<1) is the second item in the subguard1. predicate list. Map1 is the first item of subguard1.maplist. If both of the predicates are true, then subguard1.maplist is executed, this represents the (AND) logical operation between two predicates in the same subgurad. But, if we have the following guard: (M(p3)>0) OR (M(p4)!=null); map1: M(p5)=M(p4); the first predicate (M(p3)>0) is the first item of the subguard1.predicate list with map1 inside subguard1.maplist, and the second predicate (M(p4)!=null) is the first item of the subguard2.predicate list with map1 inside subguard2.maplist. That means map1 is stored in two different map lists. So, if one of these predicates is true, then map1 is executed anyway, and thus how the (OR) logical operation is implemented.

The mapping list consists of a list of objects (vector objects, Boolean objects, and/or OETPNs) the number of these objects is equal to the number of output palaces of the transition. The Mapping class contains the functions which are used to validate predicates and loop over the subguards and set the mapping in the output places of the transitions as well as creating a new subOETPN.

Also, a DataOverNetwork instance has to be initiated to send the data over network to another OETPN if the output place of the transition is an output channel. If more than one sub guard is true, the first one found is taken into consideration, and if there are conflict transitions, the one with the lowest index wins the lottery.

*Transition algorithm:*

Input: oti, toi, gi, Pre, Post, M0, P, T, D, Grd&Map, network\_flag, Out, Inp;

Initialization: M = M0, execList = empty;

**for** all gki ∈ gi do

**for** all πli ∈ gki

**if** !πli **then**

**exit** //covers (and) operation

**end** **if**

**end** **for**

**for** all p ∈ to // the count of mapping list is equal the Post

**if** map = object **then //**a voctor of float or a Boolean object

\*Set object to Mi

**if** network\_flag = true **then**

Send object over network

**end if**

**else**

**if** map = active OETPN **then**

\*Start new thread with sub OETPN

**else**

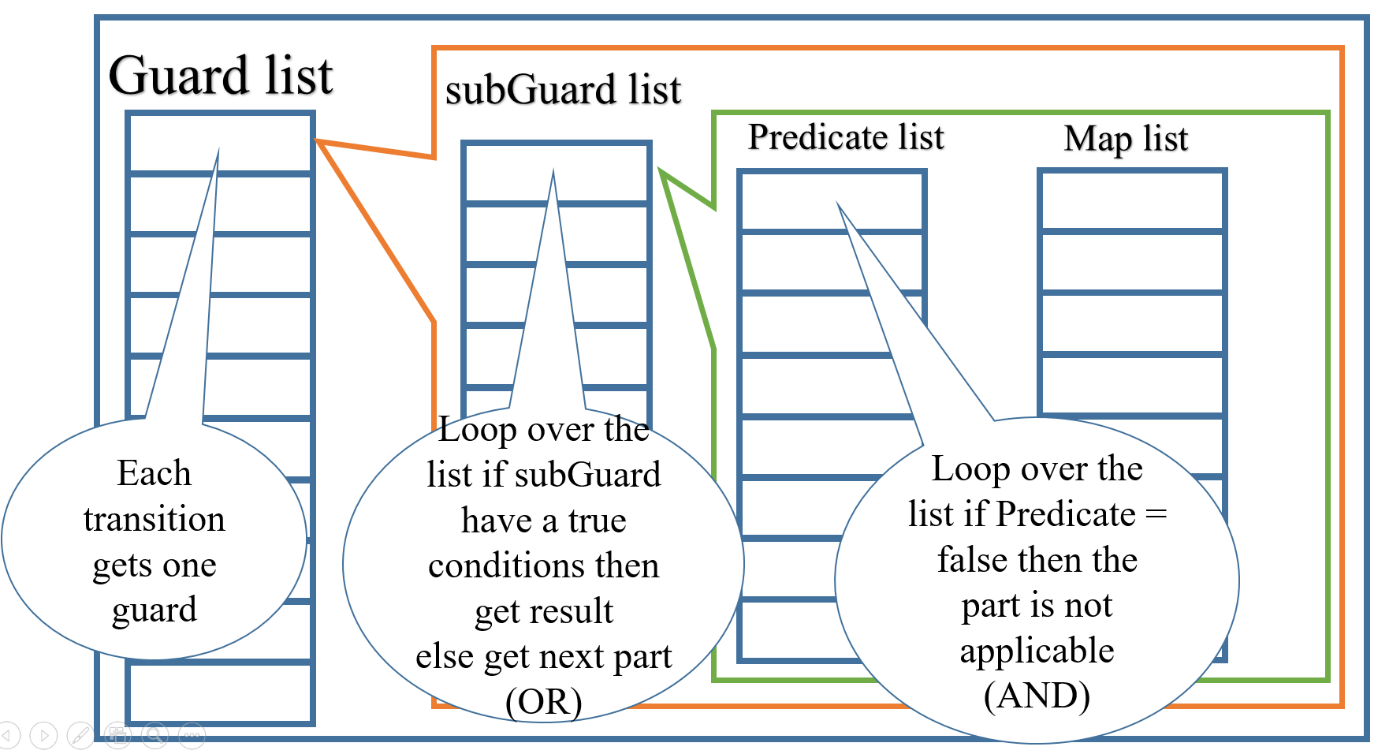
\*Set passive OETPN marking to Mi //this case of OETPN is passive

**end if**

**end if**

**end for**

**End for**



**Fig.3 Guard and mapping diagram**

*D. Task Migration implementation*

OER-TPNs can send and receive passive tokens ( (sub OER-TPNs) and objects ) through their input and output channels respectively. The passive token contains the complete data of the sub OER-TPN hence it is an instance of the PetriData class, it has its own structure (pre and post) and the behavior (guards conditions and mappings). The output channel of the sender OER-TPN is of type DataOverNetwork which is created while defining the OER-TPN’s places types, this definition contains the input channel of the recipient OER-TPN, its Network Port number, and its IP address. The mapping of the sender OER-TPN serializes the passive sub OER-TPN and sends it to the recipient OER-TPN through the network.

The recipient OER-TPN receives the token in its specified input channel, if the input channel doesn’t contain a token i.e. null, then the token is injected there directly, if not, it replaces the old token with the one received from the network. If the received token is a passive sub OER-TPN, it is executed when it passes one of the recipient OER-TPN transitions that has the mapping to do so. This ensures that the recipient OER-TPN continues working and there is no need to shut it down where only the sub OER-TPN is turned to passive and replaced by the new one.

4. Distributed execution

*A. Specifications*

Two OER-TPNs PN1 and PN2 are distributed implemented and can communicate using a TCP/IP protocol modeled by the transitions *tc1*and *tc2* as can be seen in Fig. 4. PN1 has the place *p2* linked to the keyboard and the place *p7* linked to the screen performing the print method. A task modeled by PN3 is built according to the information set by a user in the place p2 by the mapping of the transition *t1*. This new task is set in *p3* as a passive object.



**Fig. 4. Example of a distributed OETPN.**

The mapping of the transition *tc1* sends the object from *p3* to *p22*. Similarly, the mappings *map21, map22* move the object to *p24*. Unlike the previous mappings, *map22* sets it as an active object, links the task’s place p35 to the communication channel implemented by *tc2*, links the place *p36* to the environment screen print method and starts the task execution.

PN1 receives the information sent by PN3 through the place *p6* (linked to the communication channel) and the mapping *map2* sets it in the place p7. The PN1 executor prints the received information on the screen.

**PN\_1:**

Places types:

*type(p3) = OER-TPN*

*type(p1) = type(p2) = type(p4) = type(p5) = type(p6) = type(p7) =float*

t1: grd1 (p1 ≠ null) And (p2 ≤1)

map1 p4 = p1, p3 = PN\_3 (Create the passive OER-TPN)

tc\_1: grdc1 (p3 ≠ null)

mapc1 sendOverNetwork (PN\_3) (passive OER-TPN)

**PN\_2:**

Places types:

*type(p22) = type(p23) = type(p24) = OER-TPN*

*type(p21) = type(p25) = float*

t21: grd21 (p21 ≠ null) And (p22 ≠ null)

map21 p23 = p22 (passive OER-TPN)

t22: grd22 (p23 ≠ null)

map22 p25 = 1, p24 = p24, p24.start(); (Active OER-TPN)

t23: grd23 (p25 ≠ null)

map23 p21 = p25

**PN\_3:**

All types are float

t31: grd31 (p31 ≠ null)

map31 p32 = p31 + 0.1, p36 = p31 + 0.1

t32: grd32 (p32 ≠ null)

map32 p33 = p32, p34 = p32

t33: grd33 (p34 ≠ null)

map33 p33.sendOverNetwork to(p34) //sends the token to PN\_1 input channel p6

p31 = p34

t34: grd34 (p33 ≠ null) And (P33 ≥ 3)

map34 StopPetri();

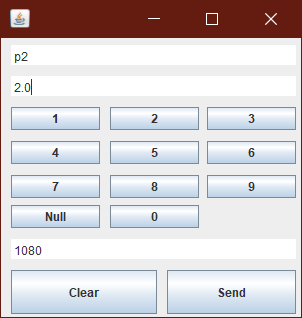
*A. Implementation*

The implementation is given with the (DCS OER-TPN Lab / New OETPN) eclipse workspace, in the *Distributed\_OER\_TPN\_Lab* package.

Note: Switch your eclipse directly to the give workspace location.

When testing, run the *GUI package/ InputFloat* class so we can send input through the keyboard to the input channel (p4) of the controller as shown in fig. 5.

In the first field you have to input the name of the input channel, in the second field is the current height of the lake, and the last field represents the port number of the OER-TPN which is 1080 where in this case is the port number for the PN1.



**Fig. 5 The GUI/ Input Float**

**Code explanation:**

In code sequence 1, a *PetriNet* object is created and given a name (Controller 1) so it can be shown on the *GUI* of the controller.

PetriNet pn = **new** PetriNet();

pn.PetriNetName = "PN1";

pn.NetworkPort = 1080;

The places are created according to their data types, for example p1 is of type *DataFloat*, the place name is considered as the place ID to access the data, The initial value can be set using the *setValue*(*value*) method, then each place must be added to the *PlaceList* of the *PetriNet* object pn.

DataFloat p1 = **new** DataFloat();

p1.SetName("p1");

p1.SetValue(2.0f);

pn.PlaceList.add(p1);

The Output channels are places with the type *DataTransfer*, in the place value we must introduce the IP address where the targeted OER-TPN is, its port number, and the place ID of its input channel. In our case, the PN2 OER-TPN is running on the same computer so the IP is set to the *locahost*, the port number of PN2 is 1090, and its input channel ID is p22.

DataTransfer p3Send = **new** DataTransfer();

p3Send.SetName("p3Send");

p3Send.Value = **new** TransferOperation("localhost", "1090", "p22");

pn.PlaceList.add(p3Send);

If we want to compare values in the *Condition* or set values in the mappings of any transition, we need to have a place to compare with or to get the value form. So extra places are defined to store the values and they are added to ConstantPlaceList that is not a part of the Petri Net’s structure. In this example, the constant value is used in PN3

DataFloat constantVal1 = **new** DataFloat();

constantVal1.SetName("constantVal1");

constantVal1.SetValue(0.1f);

mpn.ConstantPlaceList.add(constantVal1);

The transitions are defined as *PetriTransition*, the *TransitionName* is the transition ID. The input places for the transition must be added. The variables that are used for comparison or for mapping are added as well.

PetriTransition t1 = **new** PetriTransition(pn);

t1.TransitionName = "t1";

t1.InputPlaceName.add("p1");

t1.InputPlaceName.add("p2");

To define a guard for the transition, Condition objects must be created containing the transition ID, the input place, and the condition operation. The *Condition* objects must be linked with *LogicConnector* (AND, OR).

**The implementation for sub guard 1 for t34 in PN3:** grd34 (p33 != null) And (p33 ≥ 3)

The t34Ct1 condition is connected to t34Ct2, just like a *LinkList*. The logical connector between the condition parts is AND.

Condition t34Ct1 = **new** Condition(t34, "p33", TransitionCondition.***NotNull***);

Condition t34Ct2 = **new** Condition(t34, "p33", TransitionCondition.***MoreThanOrEqual***, "constantVal2");

t34Ct1.SetNextCondition(LogicConnector.***AND***, t34Ct2);

GuardMapping grdT34 = **new** GuardMapping();

grdT34.condition = t34Ct1;

Also, a *GuardMapping* list must be created to add the first condition part t34Ct1 hence it is already connected to the second one, and to add the *Activations* (mappings). The Activation object must contain the transition ID, the input place or the variables (that are used to set the values to the output places), the transition operation (*Copy, Move, SendOverNetwork, Add, Sub, Prod, Div*), and the output place of the transitoin. The *Activations* objects must be added to the *GuardMapping* list as well. And when it is finished, the entire list should be added to the transition t1 to store the subguard.

grdT1.Activations.add(**new** Activation(t1, "PN3", TransitionOperation.***Copy***, "p3"));

grdT1.Activations.add(**new** Activation(t1, "p1", TransitionOperation.***Move***, "p4"));

grdT1.Activations.add(**new** Activation(t1, "p2", TransitionOperation.***Move***, "p3-p31"));

The first mapping here is to copy the constant value of the child OER-TPN (PN3) that we have created earlier to place p2, the second mapping is to move the token form p1 to p4, the third mapping is to set the initial marking for PN3 where the token is moved from p2 of PN1 to p3 of PN1 and then to p31 of PN3

When all the gaurds are defined and added, the transition delay must be defined. Then the transition is added to the Petri net *Transitions* list.

t1.GuardMappingList.add(grdT1);

t1.Delay = 0;

pn.Transitions.add(t1);

When the output channel is connected to a transition, The Activation object must contain the transition ID, the *transitionOperation* (*SendOverNetwork*), and the output channel ID of the same OER-TPN. p3Send is the ouput channel of PN1 that is connected to tc\_1

grdT1Send.Activations.add(**new** Activation(t1Send, "p3", TransitionOperation.***SendPetriNetOverNetwork***, "p3Send"));

To run any Petri net and view the *GUI* for it as shown in fig. 6

System.***out***.println("Exp part 1 started \n ------------------------------");

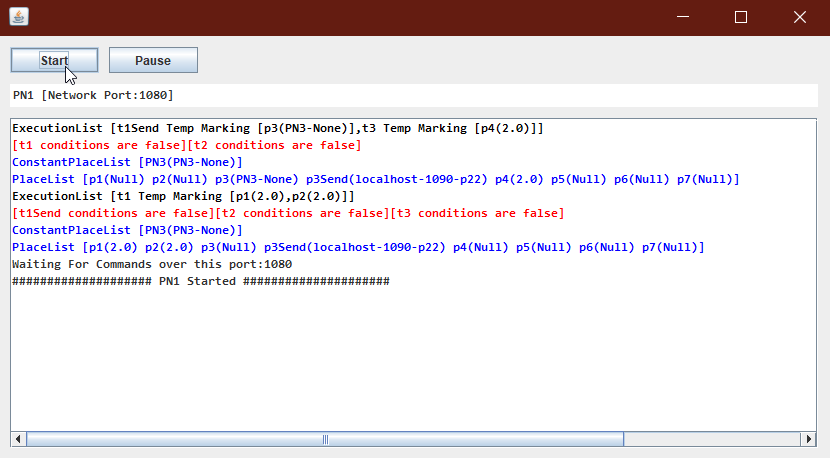
pn.Delay = 5000;

PetriNetWindow frame = **new** PetriNetWindow(**false**);

frame.petriNet = pn;

frame.setVisible(**true**);

}

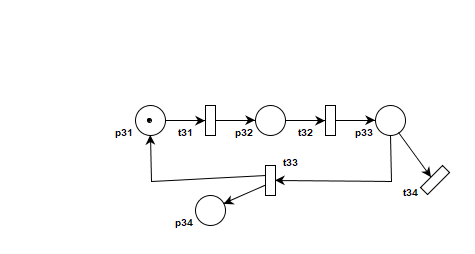


**Fig. 6 The controller GUI**

**Exercise:**

Modify the existing subguard of t1 in PN1 to be (p1 !=null) And (p2<2).

Add another subguard to t1 in PN1 (p1!= null) And (p2>=2), the following OER-TPN is created where t31 adds 0.2 to p31 and move the result to p32, t33 checks if the token is less than 4, then it sends the token to PN1 through the output channel p34, and t34 checks if p33 is more than 4 to stop the petri net.



Reference:

[1] Tiberiu S. Letia, Dahlia Al-Janabi, “Object Enhanced Time Petri Nets Models”, IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), IEEEXplore, DOI: [10.1109/AQTR.2018.8402743](https://doi.org/10.1109/AQTR.2018.8402743), 2018.