



Universidade do Porto  
Faculdade de Engenharia  
**FEUP**

# Petri Nets

*Final Report*

Formal Methods in Software Engineering  
Master in Informatics and Computing Engineering

Group Members::

Daniel Borges Pereira – 201109110 – ei11132@fe.up.pt

João Pedro Domingues da Rocha Marinho – 201101774 – ei11129@fe.up.pt

Fernando Martins – 200604165 – ei06098@fe.up.pt

17 of December 2014

# Table of Contents

Informal System Description .....	Page 3
List of Requirements .....	Page 4
Visual UML Model .....	Page 5
Formal VDM++ Model .....	Page 8
Model Validation .....	Page 17
Model Verification .....	Page 23
Conclusion .....	Page 25
References .....	Page 26

# Informal System Description

This project was focused on developing a functioning system of Petri nets, one of several mathematical programming languages used for the description of distributed systems. A Petri net consists of a directed bipartite graph, where the nodes represent *transitions* and *places*. Transitions can be seen as events that can happen, and are represented by black bars, while places are conditions, and are represented by circles.

Arcs, represented as directed arrows, connect places and transitions, and run exclusively from a place to a transition or vice-versa. *Inputs* are places an arc runs from to a transition, whereas *outputs* are arcs that run from transitions to places. Arcs can also be of the *reset* or the *inhibitor* variety. Reset arcs do not impose a precondition on firing and also empty the place when fired. Inhibitor arcs dictate that a transition only fires if the place is empty.

Places in a Petri net contain a discrete number of *tokens*, and can have a certain *capacity*, meaning the maximum amount of tokens they may contain at any given time. Any distribution of these tokens across the net is designated as a *marking*. A transition may only fire if it is *enabled*, that is, if there are enough tokens in all of its input places. When the transition fires, it consumes all of the required tokens, thus creating tokens in its output places. Firing is a non-interruptible step.

By default, a transition firing is a nondeterministic event, as without an execution policy, any enabled transition in a Petri net may fire. This makes Petri nets suitable for modelling the concurrent behavior of distributed systems.

# List of Requirements

Petri nets created in this project should allow for the execution of operations such as, but not exclusively:

- **R01 - Mandatory** - Definition of a standard Petri net comprising places, transitions and weighed arcs;
- **R02 - Mandatory** - Definition of an extended Petri net containing reset and inhibitor arcs as well as place capacities;
- **R03 - Mandatory** - Stepwise execution of a Petri net;
- **R04 - Mandatory** - Stepwise execution of an extended Petri net;
- **R05 - Mandatory** - Consumption and creation of tokens from a transition's inputs and in a transition's outputs, respectively;
- **R06 - Mandatory** - The ability to reset a Petri Net to its initial state;

Other important requirements that the project fulfills are:

- **R07 - Mandatory** - Determining whether a certain marking is reachable from an initial marking;
- **R08 - Mandatory** - Determining the set of possible sequences of transitions of the net, starting from an initial marking;

# Visual UML Model

## Use Case Model

Scenario	Setup Petri net
Description	Initial setup of a Petri net, where the transitions, places and their capacities, arcs and an initial marking is defined.
Preconditions	1. No Petri net must be defined prior.
Postconditions	1. A new Petri net must be defined; 2. All transitions must have inputs and outputs.
Steps	1. Setup places, transitions and arcs; 2. Setup initial marking.
Exceptions	None specified.

Scenario	Trigger Transition
Description	Triggers a transition if there are enough tokens in its inputs, thus removing said tokens and creating them in its outputs.
Preconditions	1. Enough tokens in its inputs. 2. Enough capacity for the output places to receive the new tokens.
Postconditions	1. Equal amount of tokens taken from its inputs in its outputs; 2. Places with a reset arcs are empty of tokens.
Steps	1. Fire the transition.
Exceptions	None specified.

<b>Scenario</b>	<b>Reset net</b>
<b>Description</b>	Resets the net to its initial marking.
<b>Preconditions</b>	1. A Petri net must exist.
<b>Postconditions</b>	1. The net's current marking must be the same as its initial marking.
<b>Steps</b>	1. Reset the net's current marking.
<b>Exceptions</b>	None specified.

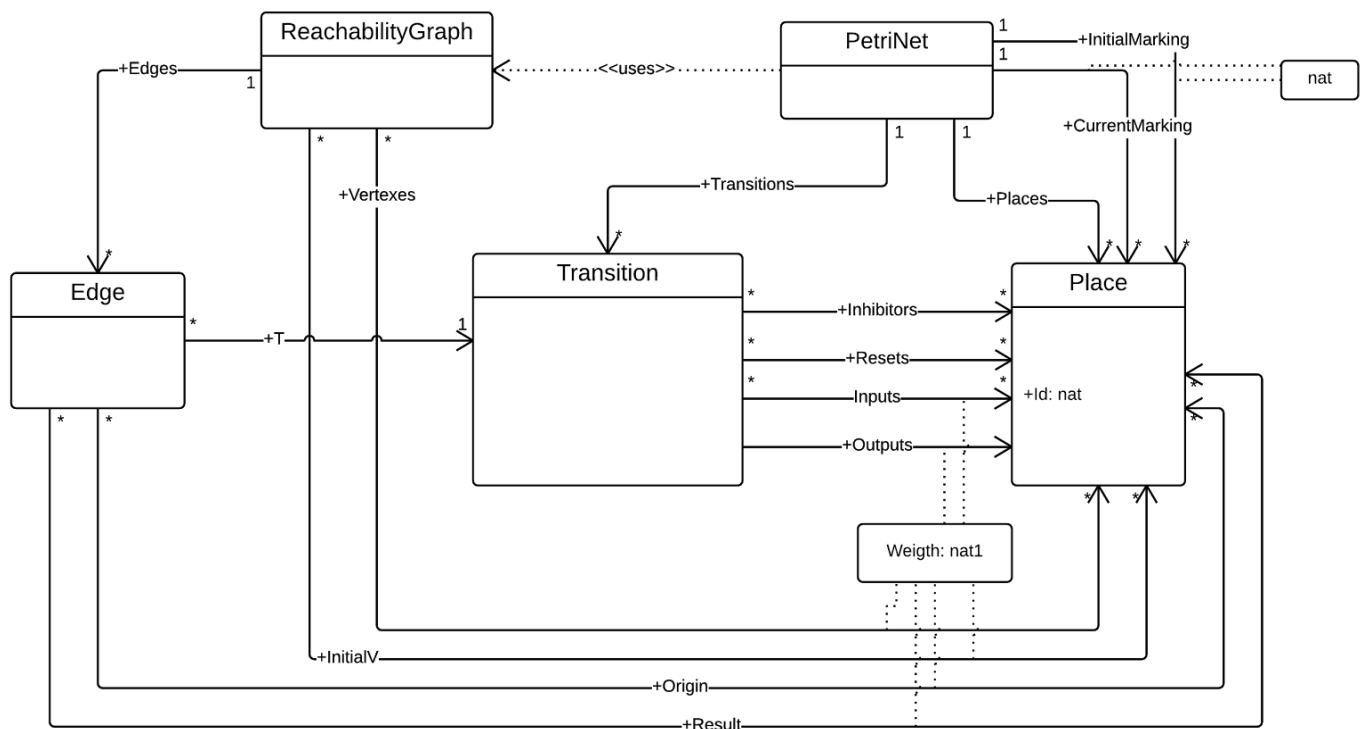
<b>Scenario</b>	<b>Calculate Reachability Graph</b>
<b>Description</b>	Calculates the reachability graph of a Petri net.
<b>Preconditions</b>	1. A Petri net must exist.
<b>Postconditions</b>	1. A graph must be generated. 2. All vertices in the graph must belong to the set of Places in the net.
<b>Steps</b>	1. Calculate a reachability graph.
<b>Exceptions</b>	None specified.

<b>Scenario</b>	<b>Check a Marking's Reachability</b>
<b>Description</b>	Checks if a given marking is reachable.
<b>Preconditions</b>	1. A Petri net must exist. 2. The given marking must be valid within that Petri net (i.e. its places must all belong to that Petri net).
<b>Postconditions</b>	1. A boolean value must be returned.
<b>Steps</b>	1. Test if a marking is reachable.
<b>Exceptions</b>	None specified.

<b>Scenario</b>	<b>Get Reachable Markings</b>
<b>Description</b>	Analyzes the net and returns the set of reachable markings within it.
<b>Preconditions</b>	1. A Petri net must exist.
<b>Postconditions</b>	1. A set of reachable markings must be returned.
<b>Steps</b>	1. Calculate the set of reachable markings.
<b>Exceptions</b>	None specified.

## Class Model

The visual UML class model for this project is represented below:



# Formal VDM++ Model

## Class PetriNet

**class PetriNet**

**types**

public Marking = map Place to nat;

**instance variables**

public Places : set of Place := {};

public Transitions : set of Transition := {};

public InitialMarking : Marking := {}->;

public CurrentMarking : Marking := {}->;

-- Checks if all inputs, outputs, resets and inhibitors in every transition belong to the set of places in this Petri net.

inv forall t in set Transitions & dom t.Inputs subset Places and  
dom t.Outputs subset Places and  
t.Resets subset Places and  
t.Inhibitors subset Places;

-- Checks if all transitions have at least one input arc and one output arc.

inv forall t in set Transitions & (dom t.Inputs <> {} or t.Resets <> {} or t.Inhibitors <> {}) and dom t.Outputs <> {};

-- No places with tokens over their capacity.

inv forall p in set dom CurrentMarking & p.LimitedCapacity = true => CurrentMarking(p) <= p.Capacity;



## operations

```
-- Constructor for the Petri Net.
-- @param places: the set of places in the net
-- @param transitions: the set of transitions in the net
-- @returns the petri net
public PetriNet(places: set of Place, transitions: set of Transition) result : PetriNet == (
    Places := places;
    Transitions := transitions;
    InitialMarking := {}->{};
    CurrentMarking := InitialMarking;

    return self;
)
pre (forall t in set Transitions & dom t.Inputs subset Places and
    dom t.Outputs subset Places and t.Resets subset Places and
    t.Inhibitors subset Places) and
-- All inputs, outputs, resets and inhibitors are in the set of Places of this Petri Nets
    (forall p in set dom InitialMarking & p in set Places) and
    (forall p in set dom CurrentMarking & p in set Places)
-- All places in the initial and current markings must be in the set of Places of this Petri Net.
post result.Places = places and result.Transitions = transitions and
    result.InitialMarking = {}->{} and result.CurrentMarking = {}->{};

-- Initializes this Petri Net with the provided marking.
-- @param m: the initial marking of the Petri Net.
public initializeMarking(m : Marking) == (
    InitialMarking := m;
    CurrentMarking := InitialMarking;
)
pre dom m = Places;
```

```

-- Checks if a transition is enabled at the time.
-- @param t: the transition to check
-- @returns true if the transition is enabled, false otherwise
public transitionEnabled(t : Transition) result : bool == (
    decl enabled: bool := true;

    for all i in set dom t.Inputs do (
        if CurrentMarking(i) < t.Inputs(i) then enabled := false;
    );

    for all i in set t.Inhibitors do (
        if CurrentMarking(i) > 0 then enabled := false;
    );

    for all o in set dom t.Outputs do (
        if o.LimitedCapacity and CurrentMarking(o) + t.Outputs(o) > o.Capacity
            then enabled := false;
    );

    return enabled;
)
pre t in set Transitions and dom CurrentMarking = Places
post result = true => ((forall i in set dom t.Inputs & CurrentMarking(i) >= t.Inputs(i)) and
    (forall i in set t.Inhibitors & CurrentMarking(i) = 0));
-- If true the transition must be enabled.

```

-- Gets all enabled transitions in the net.

-- @returns a set containing all enabled transitions at the time

```
public getEnabled() result : set of Transition == (
```

```
    dcl enabledTrans: set of Transition := {};
```

```
    for all t in set Transitions do
```

```
        if transitionEnabled(t) then enabledTrans := enabledTrans union {t};
```

```
    return enabledTrans;
```

```
)
```

```
post result subset Transitions and
```

```
    (forall t in set result & forall i in set dom t.Inputs & CurrentMarking(i) >= t.Inputs(i)) and
```

```
    (forall t in set result & forall i in set t.Inhibitors & CurrentMarking(i) = 0);
```

-- All returned transitions must be enabled.

-- Resets the petri net marking to its initial marking.

```
public reset() == (
```

```
    CurrentMarking := InitialMarking;
```

```
)
```

```
pre dom InitialMarking = Places
```

```
post CurrentMarking = InitialMarking;
```

```

-- Triggers the selected transition if it's enabled.
-- @param t: The transition to trigger.
public fireTransition(t : Transition) == (
    if(transitionEnabled(t)) then (
        for all i in set dom t.Inputs do
            CurrentMarking := CurrentMarking ++ {i |-> CurrentMarking(i) - t.Inputs(i)};

        for all o in set dom t.Outputs do
            CurrentMarking := CurrentMarking ++ {o |-> CurrentMarking(o) + t.Outputs(o)};

        for all r in set t.Resets do
            CurrentMarking := CurrentMarking ++ {r |-> 0};
    );
)
pre t in set Transitions and
    (forall i in set dom t.Inputs & CurrentMarking(i) >= t.Inputs(i)) and
    (forall i in set t.Inhibitors & CurrentMarking(i) = 0)
-- The transition must be enabled.
post (forall p in set dom t.Inputs & CurrentMarking(p) = CurrentMarking~(p) - t.Inputs(p)) and
    (forall p in set dom t.Outputs & CurrentMarking(p) = CurrentMarking~(p) + t.Outputs(p));
-- The tokens in the new marking must be set correctly.

```

```

-- Calculates the reachability graph of this Petri net.
-- @returns the reachability graph of this petry net.
public calculateReachabilityGraph() result : ReachabilityGraph == (
    dcl Graph : ReachabilityGraph := new ReachabilityGraph();
    dcl Work : seq of Marking := [InitialMarking];
    dcl M : Marking;
    dcl newM : Marking;
    dcl edge : Edge;
    dcl pn : PetriNet := new PetriNet(Places,Transitions);
    pn.InitialMarking := InitialMarking;

    Graph.Vertexes := {InitialMarking};
    Graph.Edges := {};
    Graph.InitialV := InitialMarking;

    while Work <> [] do (
        M := hd Work;
        Work := tl Work;

        pn.CurrentMarking := M;
        for all t in set pn.getEnabled() do (
            pn.fireTransition(t);
            newM := pn.CurrentMarking;
            if newM not in set Graph.Vertexes then (
                Graph.Vertexes := Graph.Vertexes union {newM};
                Work := Work ^ [newM];
            );

            edge := new Edge();
            edge.Origin := M;
            edge.Result := newM;
            edge.T := t;

            Graph.Edges := Graph.Edges union {edge};
        );
    );

```

```

    );

    return Graph;
)
post (forall m in set result.Vertexes & dom m subset Places) and
-- All vertexes must belong to the set of Places of this Petri Net
    result.InitialV = InitialMarking and
    (forall e in set result.Edges &
        dom e.Origin subset Places and dom e.Result subset Places and e.T in set Transitions);
-- All of the edges must connect two Places in this Petri Net.

-- Calculates if a given marking is reachable from the initial marking on this Petri net.
-- @param m: the marking to calculate the reachability for.
-- @returns true if m is reachable, false otherwise.
public isMarkingReachable(m : Marking) result : bool == (
    dcl Graph : ReachabilityGraph := calculateReachabilityGraph();
    if m in set Graph.Vertexes then return true;

    return false;
)
pre dom m subset Places;
-- All places in the provided marking must belong to the set of Places of this Petri Net.

-- Calculates the set of reachable markings from the initial marking in this Petri net.
-- @returns the set of reachable markings.
public getReachableMarkings() result : set of Marking == (
    dcl Graph : ReachabilityGraph := calculateReachabilityGraph();
    return Graph.Vertexes;
)
post forall m in set result & dom m subset Places;
-- All places in every marking returned must belong to the set of Places in this Petri Net.

end PetriNet

```

## Class Place

**class Place**

**instance variables**

```
public Id : nat := 0;
public LimitedCapacity : bool := false;
public Capacity : nat := 1;
```

**end Place**

## Class Transition

**class Transition**

**types**

```
public Weight = nat1;
```

**instance variables**

```
public Resets : set of Place := {};
public Inhibitors : set of Place := {};
public Inputs : map Place to Weight := {|->};
public Outputs : map Place to Weight := {|->};
```

**operations**

```
-- Constructor for the Transition.
-- @param inputs: the set of places that are inputs to this transition
-- @param outputs: the set of places that are outputs of this transition
-- @param resets: the set of places that are reseted by this transition
-- @param inhibitors: the set of places that inhibit this transition
-- @returns the transition
public Transition(inputs : map Place to Weight, outputs : map Place to Weight,
    resets : set of Place, inhibitors : set of Place) result: Transition == (
    Inputs := inputs;
    Outputs := outputs;
```

```

        Resets := resets;
        Inhibitors := inhibitors;
        return self;
    )
    pre (inputs <> {}|->} or resets <> {} or inhibitors <> {}) and outputs <> {}|->}
    post result.Inputs = inputs and result.Outputs = outputs and result.Resets = resets and
    result.Inhibitors = inhibitors;

end Transition

```

## Class ReachabilityGraph

```

class ReachabilityGraph
    instance variables
        public Vertexes : set of PetriNet`Marking := {};
        public Edges : set of Edge := {};
        public InitialV : PetriNet`Marking := {}|->};
end ReachabilityGraph

```

## Class Edge

```

class Edge
    instance variables
        public Origin : PetriNet`Marking := {}|->};
        public Result : PetriNet`Marking := {}|->};
        public T : Transition;
end Edge

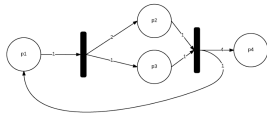
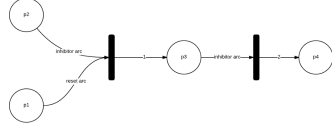
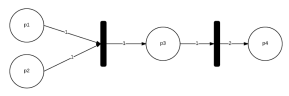
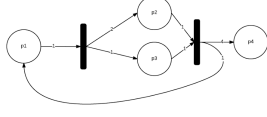
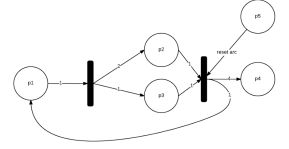
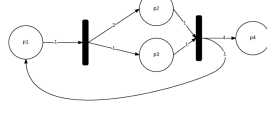
```



# Model Validation

## Coverage Tests

Below is a table of the tests performed upon the project:

Test Number	Requirement	Petri net
1	R01, R03 and R05	
2	R02, R04, R05 and R06	
3	R07 and R08	
4	R02, R04 and R05	
5	---	
6	---	

The code for coverage tests is presented below:

### **class TestPetriNet**

#### **operations**

```
private assertTrue: bool ==> ()
```

```
assertTrue(cond) == return
```

```
pre cond;
```

-- Requirement R01, R03 & R05 - Tests the definition and step-wise execution of a standard Petri Net.

```
public RunTest1() == (
```

```
    dcl p1 : Place := new Place();
```

```
    dcl p2 : Place := new Place();
```

```
    dcl p3 : Place := new Place();
```

```
    dcl p4 : Place := new Place();
```

```
    dcl t1 : Transition := new Transition({p1 |-> 1},{p2 |-> 2, p3 |-> 1},{},{});
```

```
    dcl t2 : Transition := new Transition({p2|->1,p3|->1}, {p1|->1,p4|->4},{},{});
```

```
    dcl pn1 : PetriNet := new PetriNet({p1,p2,p3,p4},{t1,t2});
```

```
    p3.Id := 3;
```

```
    p1.Id := 1;
```

```
    p2.Id := 2;
```

```
    p4.Id := 4;
```

```
    pn1.initializeMarking({p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0});
```

```
    assertTrue(pn1.transitionEnabled(t1) = true);
```

```
    assertTrue(pn1.transitionEnabled(t2) = false);
```

```
    pn1.fireTransition(t1);
```

```
    assertTrue(pn1.transitionEnabled(t1) = false);
```

```
    assertTrue(pn1.transitionEnabled(t2) = true);
```

```

pn1.fireTransition(t2);

assertTrue(pn1.transitionEnabled(t1) = true);
assertTrue(pn1.transitionEnabled(t2) = false);

assertTrue(pn1.CurrentMarking(p4) = 4);
assertTrue(pn1.CurrentMarking(p2) = 1);
assertTrue(pn1.CurrentMarking(p1) = 1);
);

-- Requirement 02, 04 & 05 - Tests the definition and stepwise execution of extended Petri Nets with
reset and inhibitor arcs.
-- Requirement 06 - Also tests the reset function of a Petri Net.
public RunTest2() == (
    dcl p1 : Place := new Place();
    dcl p2 : Place := new Place();
    dcl p3 : Place := new Place();
    dcl p4 : Place := new Place();

    dcl t1 : Transition := new Transition({|->},{p3 |-> 1},{p1},{p2});
    dcl t2 : Transition := new Transition({|->},{p4 |-> 2},{},{p3});

    dcl pn1 : PetriNet := new PetriNet({p1,p2,p3,p4},{t1,t2});

    p3.Id := 3;
    p1.Id := 1;
    p2.Id := 2;
    p4.Id := 4;

    pn1.initializeMarking({p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0});
    assertTrue(pn1.getEnabled() = {t1,t2});
    pn1.fireTransition(t1);
    assertTrue(pn1.CurrentMarking(p1) = 0);
    assertTrue(pn1.transitionEnabled(t2) = false);
    pn1.reset();
    assertTrue(pn1.CurrentMarking = {p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0});

```

```
);
```

-- Requirement 07 & 08 - Tests the definition and calculation of the reachability of a Petri Net.

```
public RunTest3() == (  
    dcl p1 : Place := new Place();  
    dcl p2 : Place := new Place();  
    dcl p3 : Place := new Place();  
    dcl p4 : Place := new Place();  
  
    dcl t1 : Transition := new Transition({p1 |-> 1, p2 |-> 1},{p3 |-> 1},{},{});  
    dcl t2 : Transition := new Transition({p3 |-> 1, {p4 |-> 2},{},{});  
    dcl pn1 : PetriNet := new PetriNet({p1,p2,p3,p4},{t1,t2});  
    dcl reachable : set of PetriNet`Marking;  
  
    p3.Id := 3;  
    p1.Id := 1;  
    p2.Id := 2;  
    p4.Id := 4;  
  
    pn1.initializeMarking({p1 |-> 1, p2 |-> 1, p3 |-> 0, p4 |-> 0});  
    reachable := pn1.getReachableMarkings();  
    assertTrue(reachable subset { {p1 |-> 1, p2 |-> 1, p3 |-> 0, p4 |-> 0} ,  
    {p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 0}, {p1 |-> 0, p2 |-> 0, p3 |-> 0, p4 |-> 2}});  
    assertTrue(pn1.isMarkingReachable({p1 |-> 0, p2 |-> 0, p3 |-> 0, p4 |-> 2}));  
    assertTrue(pn1.isMarkingReachable({p1 |-> 0, p2 |-> 0, p3 |-> 0, p4 |-> 4}) = false);  
);
```

-- Requirement 02, 04 & 05 - Tests the definition and step-wise execution of an extended Petri Net with Place capacities.

```
public RunTest4() == (  
    dcl p1 : Place := new Place();  
    dcl p2 : Place := new Place();  
    dcl p3 : Place := new Place();  
    dcl p4 : Place := new Place();  
  
    dcl t1 : Transition := new Transition({p1 |-> 1},{p2 |-> 2, p3 |-> 1},{},{});  
    dcl t2 : Transition := new Transition({p2 |-> 1, p3 |-> 1}, {p1 |-> 1, p4 |-> 4},{},{});  
    dcl pn1 : PetriNet := new PetriNet({p1,p2,p3,p4},{t1,t2});
```

```

    p3.Id := 3;
    p1.Id := 1;
    p2.Id := 2;
    p4.Id := 4;
    p2.LimitedCapacity := true;
    p2.Capacity := 2;

    pn1.initializeMarking({p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0});

    assertTrue(pn1.transitionEnabled(t1) = true);
    assertTrue(pn1.transitionEnabled(t2) = false);

    pn1.fireTransition(t1);
    assertTrue(pn1.transitionEnabled(t1) = false);
    assertTrue(pn1.transitionEnabled(t2) = true);

    pn1.fireTransition(t2);
    assertTrue(pn1.transitionEnabled(t1) = false);
    assertTrue(pn1.transitionEnabled(t2) = false);

    assertTrue(pn1.CurrentMarking(p4) = 4);
    assertTrue(pn1.CurrentMarking(p2) = 1);
    assertTrue(pn1.CurrentMarking(p1) = 1);
);

-- Tests the definition of an invalid Petri Net with a transition from a Place not in the Petri Net.
-- This test should fail.
public RunTest5() == (
    dcl p1 : Place := new Place();
    dcl p2 : Place := new Place();
    dcl p3 : Place := new Place();
    dcl p4 : Place := new Place();
    dcl p5 : Place := new Place();

    dcl t1 : Transition := new Transition({p1 |-> 1},{p2 |-> 2, p3 |-> 1},{},{});

```

```

    dcl t2 : Transition := new Transition({p2 |-> 1, p3 |-> 1}, {p1 |-> 1, p4 |-> 4},{p5},{});

    dcl pn1 : PetriNet := new PetriNet({p1,p2,p3,p4},{t1,t2});

    IO`println("This should fail.");
);

-- Tests the definition of an invalid Petri Net with the current marking containing a Place not in the Petri
Net.
-- This test should fail.
public RunTest6() == (
    dcl p1 : Place := new Place();
    dcl p2 : Place := new Place();
    dcl p3 : Place := new Place();
    dcl p4 : Place := new Place();
    dcl p5 : Place := new Place();

    dcl t1 : Transition := new Transition({p1 |-> 1},{p2 |-> 2, p3 |-> 1},{},{});
    dcl t2 : Transition := new Transition({p2 |-> 1, p3 |-> 1}, {p1 |-> 1, p4 |-> 4},{},{});

    dcl pn1 : PetriNet := new PetriNet({p1,p2,p3,p4},{t1,t2});

    pn1.initializeMarking({p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 1});
    IO`println("This should fail.");
);

-- Executes the test suite.
public static main: () ==> ()
main() == (
    new TestPetriNet().RunTest1();
    new TestPetriNet().RunTest2();
    new TestPetriNet().RunTest3();
    new TestPetriNet().RunTest4();
    new TestPetriNet().RunTest5(); -- Comment out this test in order to succeed.
    new TestPetriNet().RunTest6(); -- Comment out this test in order to succeed.
);

end TestPetriNet

```

# Model Verification

## Example of Domain Verification

One of the proof obligations generated by Overture is:

No	PO Name	Type
4	PetriNet`transitionEnabled(Transition)	legal map application

For which the code is:

$$((t \text{ in set Transitions}) \Rightarrow ((\text{result} = \text{true}) \Rightarrow (\text{forall } i \text{ in set } (\text{dom } (t.\text{Inputs})) \& \\ (i \text{ in set } (\text{dom CurrentMarking}))))))$$

The code related to this proof is as follows:

```
public transitionEnabled(t : Transition) result : bool == (
  dcl enabled: bool := true;
  for all i in set dom t.Inputs do (
    if CurrentMarking(i) < t.Inputs(i) then enabled := false;
  );
  for all i in set t.Inhibitors do (
    if CurrentMarking(i) > 0 then enabled := false;
  );
  for all o in set dom t.Outputs do (
    if o.LimitedCapacity and CurrentMarking(o) + t.Outputs(o) > o.Capacity
    then enabled := false;
  );
  return enabled;
)
pre t in set Transitions and dom CurrentMarking = Places
post result = true => ((forall i in set dom t.Inputs & CurrentMarking(i) >= t.Inputs(i)) and
  (forall i in set t.Inhibitors & CurrentMarking(i) = 0));
```

In order for the transition  $t$  to be enabled, all of its input places must have equal or greater number of tokens than the weight of their input arc. Therefore, if the result of the function `transitionEnabled` is true, since we make sure every place in the transition's inputs is in the Petri net's place set and that a net's current marking must map to each of its places, that transition's inputs must be in that Petri net's current marking. If they weren't, the number of tokens at that place would not be equal or larger than the weight of the arc, and the result wouldn't be true.

## Example of Invariant Verification

One of the proof obligations generated by Overture is:

No	PO Name	Type
19	PetriNet`reset()	state invariant holds

For which the code is:

```
(((forall t in set Transitions & (((dom (t.Inputs)) subset Places) and (((dom (t.Outputs)) subset Places) and (((t.Resets) subset Places) and ((t.Inhibitors) subset Places)))))) and (forall t in set Transitions & (((dom (t.Inputs)) <> {}) or (((t.Resets) <> {}) or ((t.Inhibitors) <> {}))) and ((dom (t.Outputs)) <> {})))) and (forall p in set (dom CurrentMarking) & ((p.LimitedCapacity) = true) => (CurrentMarking(p) <= (p.Capacity)))) => (((forall t in set Transitions & (((dom (t.Inputs)) subset Places) and (((dom (t.Outputs)) subset Places) and (((t.Resets) subset Places) and ((t.Inhibitors) subset Places)))))) and (forall t in set Transitions & (((dom (t.Inputs)) <> {}) or (((t.Resets) <> {}) or ((t.Inhibitors) <> {}))) and ((dom (t.Outputs)) <> {})))) and (forall p in set (dom CurrentMarking) & ((p.LimitedCapacity) = true) => (CurrentMarking(p) <= (p.Capacity))))))
```

The code related to this proof is as follows:

```
public reset() == (
    CurrentMarking := InitialMarking;
)
pre dom InitialMarking = Places
post CurrentMarking = InitialMarking;
```

Since we make sure, through pre-conditions and invariants that a Petri net's initial marking conforms to the model (i.e. it maps to all places, every transition in the net must have at least an input, reset or inhibitor as well as an output and they must refer to places in that Petri Net, a place's tokens must not



exceed its capacity, etc) and we are only assigning the current marking of the net to be the same as its initial marking, no invariants will be violated.

## Conclusion

As a result of developing this project, we have improved in our knowledge of the VDM++ modeling language and learned about Petri net formalism. We consider that the project achieved its proposed goals. It allows the definition and stepwise execution of standard and extended Petri Nets, as well as the execution of several calculations relating to reachability on them.

There were some difficulties with the implementation of the reachability calculations, but we were able to eventually deliver on all given objectives.

There are still some improvements that could be made to this project, with more time. One of the most obvious improvements that could be made is the implementation of a graphical interface that shows the location of tokens, as well as the network of places, arcs and transitions. Another good improvement would be to expand the capabilities of the model to allow the definition of other kinds of non standard Petri nets.

Regarding contribution per member, there was an equal amount of effort throughout the team. Each member gave its own contribution to the project's main functions, mainly following the fulfillment of each requirement as a guideline.

# References

For the development of this project, the following documents were consulted:

1. Petri net. Accessed on the 16th of December, 2014.  
[http://en.wikipedia.org/wiki/Petri\\_net](http://en.wikipedia.org/wiki/Petri_net).
2. Petri Nets. Accessed on the 16th of December, 2014.  
<http://www.labri.fr/perso/anca/FDS/Pn-ESTII.pdf>.
3. Petri Nets. Accessed on the 10th of December, 2014  
<http://www.informatik.uni-hamburg.de/TGI/PetriNets/>.
4. Petri Nets. Accessed on the 10th of December, 2014  
[http://www.scholarpedia.org/article/Petri\\_net](http://www.scholarpedia.org/article/Petri_net).
5. Faria, P., & Paiva, A. (2013). UML Checker: Formal Specification in VDM of the Petri Net - based Conformance Checking Engine. Retrieved December 18, 2014, from <https://blogs.fe.up.pt/sdbt/files/2013/04/TR-2013-04.pdf>