Formal Modelling of a Petri Net in VDM++

Final Report

Universidade do Porto

Faculdade de Engenharia

FEUP

Master in Informatics and Computing Engineering

Formal Methods in Software Engineering

Group T03_3:
Duarte Nuno Pereira Duarte - ei11101
Ruben Fernando Pinto Cordeiro - ei11097

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

January 7, 2015

Contents

1 Summary

The goal of this project is to model the structure and behaviour of a Petri net in VDM++.

A Petri net is mathematical modelling language for the description of distributed systems. It is typically represented by a graphical notation for stepwise processes that include choice, iteration and cocurrent execution.

In order to build the executable formal model, the model-oriented specification language from the *Vienna Development Method* (VDM++) was used, as well as the Overture Tool[3] for development.

2 System Description

Petri nets are a graphical and mathematical modeling tool applicable to many systems. As a graphical tool, Petri nets can be used as a visual-communication aid similar to flow charts, block diagrams and netweorks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems.

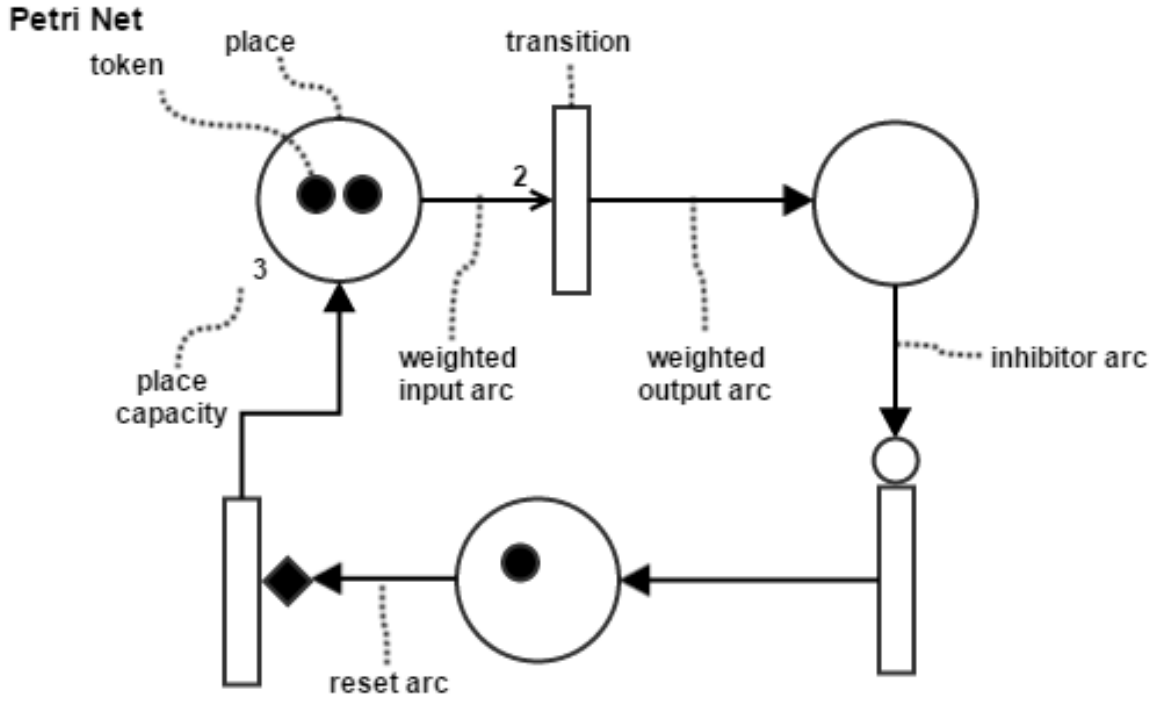


Figure 1: Petri net graphical structure

A Petri net is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, signified by bars) and places (i.e. conditions, signified by circles). The directed arcs describe which places are pre- and/or postconditions for which transitions (signified by arrows).

A Petri net can be described as a 5-tuple, $PN = (P, T, F, W, M_0)$, where:

P = $\{p_1, p_2, ..., p_m\}$ is a finite set of places.

T = $\{t_1, t_2, ..., t_n\}$ is a finite set of transitions.

F $\subseteq (P \times T) \bigcup (T \times P)$ is a set of arcs.

W : $F \to \{1, 2, 3, ...\}$ is a weight function.

M_0 : $P \to \{0, 1, 2, 3, ...\}$ is the initial marking.

Graphically, places in a Petri net may contain a discrete number of marks called tokens. Any distribution of tokens over the places will represent a configuration of the net called a marking. In an abstract sense relating to a Petri net diagram, a transition of a Petri net may fire if it is enabled, i.e. there are sufficient tokens in all of its input places; when the transition fires, it consumes the required input tokens, and creates tokens in its output places. A firing is atomic, i.e., a single non-interruptible step.

2.1 Arc types and firing rules

There are three types of arcs:

- Weighted arcs.
- Inhibitor arcs.
- Reset arcs.

The state or marking in a Petri net is changed according to a firing rule for a transition t.

A reset arc does not impose a precondition on firing: the transition is always enabled and the place is emptied when the transition fires.

An inhibitor arc imposes the precondition that the transition may only fire when the place is empty.

If the arc is weighted, the following firing rule applies:

1. A transition t is said to be enabled is each input place *p* of t is marked with at least w(p, t) tokens, where w(p, t) is the weight of the arc from p to t.
2. The firing of the enabled transition t removes w(p, t) tokens from each input place p of t, and adds w(t, p) tokens to each output place p of t, where w(t, p) is the weight of the arc from t to p.

2.2 Reachability

The reachability problem for Petri nets is to decide, given a Petri net N and a marking M, whether $M \in R(N)$. This is a matter of walking the reachability graph until either the requested marking is reached or there is a point where it can no longer be found.

3 Requirements

List of Requirements:

R1 The user can instantiate a Petri net given a set of places, transitions and arcs

R2 It should be possible to extend the Petri net with:

1. reset arcs
2. inhibitor arcs
3. adding capacities to places
4. weighted input and output arcs

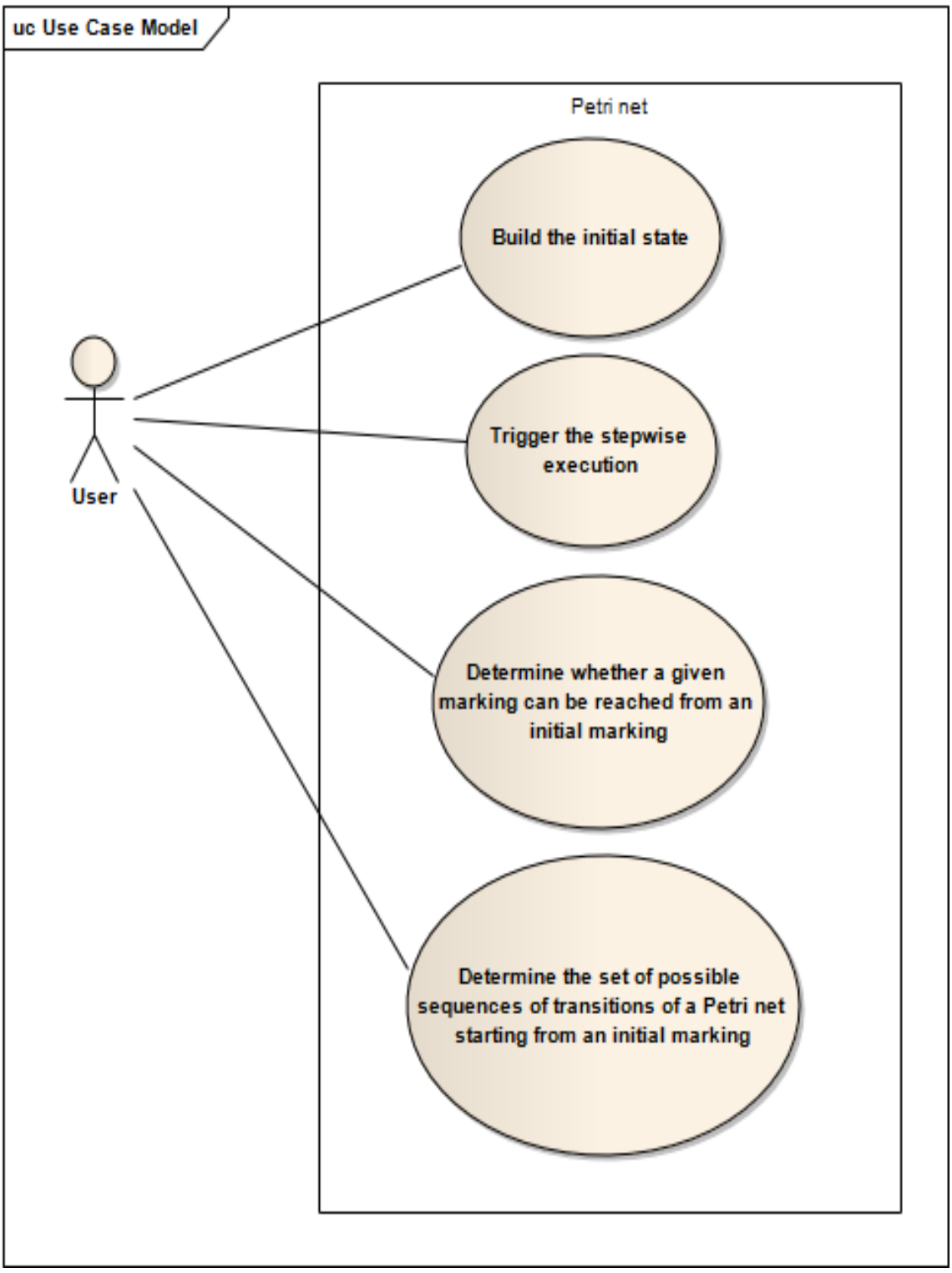
R3 The user can trigger the stepwise execution of a Petri net (by firing an enabled transition at a time).

R4 The user can verify if a given marking can be reached from an initial marking.

R5 The user can get a set of possible transitions between an initial marking and a target marking.

4 Visual UML Models

4.1 Use Case Models



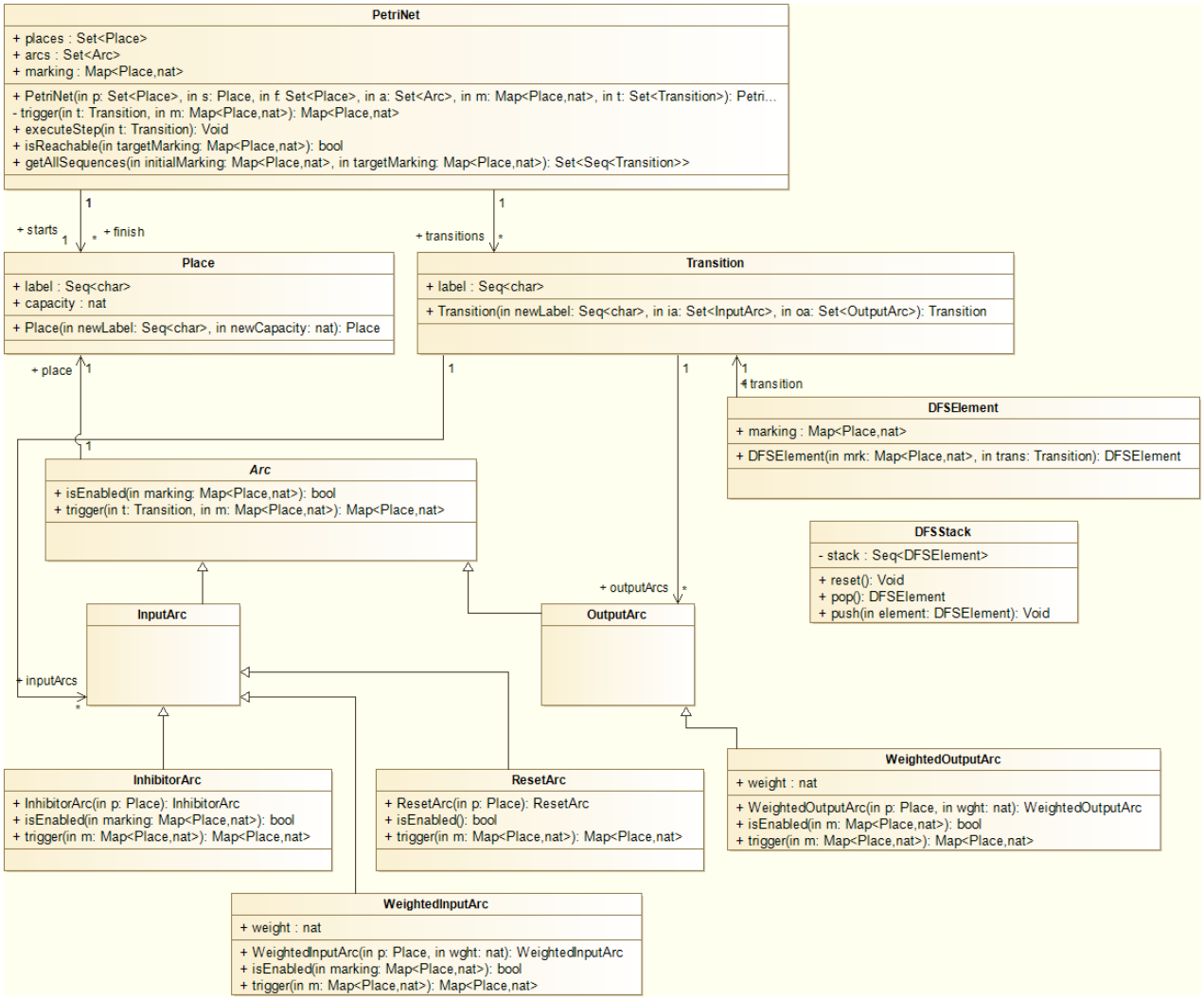
4.2 Major Use Cases Descriptions

Scenario	Trigger stepwise execution
Description	User triggers a transition of the Petri net
Pre-conditions	1. The specified transition exists in the Petri net. (*input*)
Post-conditions	1. The marking places domain remains unchanged. 2. The marking of the Petri net is changed according to the triggered transition. (*output*)
Steps	(Unspecified)
Exceptions	(Unspecified)

Scenario	Determine marking reachability
Description	Determine whether a given marking can be reached from an initial marking
Pre-conditions	1. The places of the given marking exist in the Petri net (*input*)
Post-conditions	1. The state of the Petri net remains unchanged (*final system state*)
Steps	(Unspecified)
Exceptions	Behaviour is undefined if the Petri net has cycles.

Scenario	Get all sequences of transitions
Description	Get all sequences of transitions from an initial and target marking
Pre-conditions	1. The places of the initial and target markings exist in the Petri net (*input*)
Post-conditions	1. The state of the Petri net remains unchanged (*final system state*)
Steps	(Unspecified)
Exceptions	Behaviour is undefined if the Petri net has cycles.

5 Visual UML model



6 Formal VDM++ model

6.1 Arc

```
/*
 * Abstract class to represent arcs between transitions and places
 */
class Arc
  instance variables
    /**
     * Place origin or destination of this arc
     */
    public place: Place
  operations
    /**
     * Returns true if the arc meets all requirements for an 'event' to happen
     *
     * @param marking mapping of places to number of tokens
     * @return boolean
     */

    public isEnabled: map Place to nat ==> bool
    isEnabled(marking) == is subclass responsibility;

    /**
     * Modifies the marking at the place of the arc
     *
     * @remark Should only be called if arc is enabled
     * @see isEnabled(marking)
     *
     * @param t transition to be triggered
     * @param m initial marking
     * @return new marking
     */

    public trigger: Transition * map Place to nat ==> map Place to nat
    trigger(t, m) == is subclass responsibility;
    -- pre isEnabled(m);

end Arc

/**
 * Abstract class for all input arcs: arcs from a place to a transition
 */
class InputArc is subclass of Arc

end InputArc

/**
 * Input arc that consumes all tokens at its place
 */
class ResetArc is subclass of InputArc
  operations
    /**
     * ResetArc constructor.
     *
     * @param place place connected to this arc
     */

    public ResetArc: Place ==> ResetArc
    ResetArc(p) == (
      place := p;
      return self;
```

```
    );

    /**
     * Reset arcs are always enabled
     *
     * @param marking mapping of places to number of tokens
     * @return boolean
     */

    public isEnabled: map Place to nat ==> bool
    isEnabled(marking) == (
      return true;
    )
    pre place in set dom marking;

    /**
     * Reset arcs consume all the tokens at the place
     *
     * @remark Should only be called if arc is enabled
     * @see isEnabled(marking)
     *
     * @param t transition to be triggered
     * @param m initial marking
     * @return new marking
     */

    public trigger: Transition * map Place to nat ==> map Place to nat
    trigger(-, m) == (
      dcl newMarking: map Place to nat := m;
      newMarking(place) := 0;
      return newMarking;
    )
    -- pre isEnabled(m)
    pre place in set dom m;

end ResetArc

/**
 * Input arc that is enabled if its place has no tokens
 */
class InhibitorArc is subclass of InputArc
  operations
    /**
     * InhibitorArc constructor.
     *
     * @param place place connected to this arc
     */

    public InhibitorArc: Place ==> InhibitorArc
    InhibitorArc(p) == (
      place := p;
      return self;
    );

    /**
     * Inhibitor arcs are enabled if their place is empty
     *
     * @param marking mapping of places to number of tokens
     * @return boolean
     */

    public isEnabled: map Place to nat ==> bool
    isEnabled(marking) == (
      if marking(place) = 0 then
        return true
```

```
      else
        return false;
    )
    pre place in set dom marking;

    /**
     * Inhibitor arcs do not change marking
     *
     * @remark Should only be called if arc is enabled
     * @see isEnabled(marking)
     *
     * @param t transition to be triggered
     * @param m initial marking
     * @return new marking
     */

    public trigger: Transition * map Place to nat ==> map Place to nat
    trigger(-, m) == (
      return m;
    )
    -- pre isEnabled(m)
    pre place in set dom m;

end InhibitorArc

/**
 * Input arc with weight
 */
class WeightedInputArc is subclass of InputArc
  instance variables
    /**
     * Amount of tokens that can 'travel' this arc
     */
    public weight: nat;
  operations
    /**
     * WeightedInputArc constructor.
     *
     * @param place place connected to this arc
     * @param weight capacity of this arc
     */

    public WeightedInputArc: Place * nat ==> WeightedInputArc
    WeightedInputArc(p, wght) == (
      place := p;
      weight := wght;
      return self
    );

    /**
     * Weighted input arcs are enabled if the weight
     * is less than the number of tokens of their place
     *
     * @param marking mapping of places to number of tokens
     * @return boolean
     */

    public isEnabled: map Place to nat ==> bool
    isEnabled(marking) == (
      return marking(place) >= weight;
    )
    pre place in set dom marking;

    /**
     * Weighted input arcs remove <weight> tokens from their place
```

```
     *
     * @remark Should only be called if arc is enabled
     * @see isEnabled(marking)
     *
     * @param t transition to be triggered
     * @param m initial marking
     * @return new marking
     */

    public trigger: Transition * map Place to nat ==> map Place to nat
    trigger(-, m) == (
      dcl newMarking: map Place to nat := m;
      newMarking(place) := newMarking(place) - weight;
      return newMarking;
    )
    -- pre isEnabled(m)
    pre place in set dom m;

end WeightedInputArc

/**
 * Abstract class for all output arcs: arcs from a transtion to a place
 */
class OutputArc is subclass of Arc

end OutputArc

/**
 * Output arc with weight
 */
class WeightedOutputArc is subclass of OutputArc
  instance variables
   /**
    * Amount of tokens that can 'travel' this arc
    */
    public weight: nat;
  operations
    /**
     * WeightedOutputArc constructor.
     *
     * @param place place connected to this arc
     * @param weight capacity of this arc
     */

    public WeightedOutputArc: Place * nat ==> WeightedOutputArc
    WeightedOutputArc(p, wght) == (
      place := p;
      weight := wght;
      return self
    );

    /**
     * Weighted output arcs are enabled if their place
     * capacity is not overrun after moving <weight> tokens
     *
     * @param marking mapping of places to number of tokens
     * @return boolean
     */

    public isEnabled: map Place to nat ==> bool
    isEnabled(m) == (
      return m(place) + weight <= place.capacity;
    )
    pre place in set dom m;
```

```
    /**
     * Weighted output arcs moves <weight> tokens to their place
     *
     * @remark Should only be called if arc is enabled
     * @see isEnabled(marking)
     *
     * @param t transition to be triggered
     * @param m initial marking
     * @return new marking
     */

    public trigger: Transition * map Place to nat ==> map Place to nat
    trigger(-, m) == (
      dcl newMarking: map Place to nat := m;
      newMarking(place) := newMarking(place) + weight;
      return newMarking;
    )
    -- pre isEnabled(m)
    pre place in set dom m;

end WeightedOutputArc
```

Function or operation	Line	Coverage	Calls
InhibitorArc	100	100.0%	2
ResetArc	53	100.0%	2
WeightedInputArc	154	100.0%	19
WeightedOutputArc	216	100.0%	24
isEnabled	112	100.0%	5
isEnabled	168	100.0%	144
isEnabled	230	100.0%	100
isEnabled	65	100.0%	4
isEnabled	17	100.0%	2
trigger	80	100.0%	4
trigger	183	100.0%	83
trigger	30	100.0%	2
trigger	245	100.0%	97
trigger	130	100.0%	2
Arc.vdmpp		100.0%	490

6.2 PetriNet

```
/**
 * This class represents Petri nets with places, transitions
 * and multiple types of arcs
 */
class PetriNet
  instance variables
    /**
     * set of places (redundant, places are referenced from arcs)
     */
    public places : set of Place := {};

    /**
     * set of arcs (redundant, arcs are referenced from transitions)
     */
    public arcs : set of Arc := {};

    /**
     * set of transitions
     */
    public transitions : set of Transition := {};

    /**
     * state of the Petri net
     * mapping of places to number of tokens
     */
    public marking: map Place to nat := {|->};

    /**
     * single starting place
     */
    public starts : Place;

    /**
     * multiple ending places
     */
    public finish : set of Place := {};

    inv starts in set places;
    inv finish subset places;
    inv forall arc in set arcs & arc.place in set places;
    inv dom marking subset places;
    inv forall transition in set transitions & (
     transition.inputArcs subset arcs and
     transition.outputArcs subset arcs
    );

  operations
    /**
     * PetriNet constructor.
     *
     * @param p set of places
     * @param s start place
     * @param f set of finish places
     * @param a set of arcs
     * @param m initial marking (mapping of places to number of tokens)
     * @param t set of transitions
     */

    public PetriNet: set of Place * Place * set of Place * set of Arc *
                     map Place to nat * set of Transition ==> PetriNet
    PetriNet(p, s, f, a, m, t) == (
      atomic(
```

```
    places := p;
    starts := s;
    finish := f;
    arcs := a;
    marking := m;
    transitions := t;
  )
  return self;
)
pre dom m = p and
  s in set p and
  f subset p and
  forall a1 in set a & a1.place in set p and
  forall t1 in set t & (
   forall a2 in set t1.inputArcs & a2 in set a and
   forall a3 in set t1.outputArcs & a3 in set a
  );

/**
 * Triggers given transition and returns the new state of the Petri net.
 *
 * @remark The internal state of the Petri net is not changed by this
 *         method. For that intention, use executeStep instead.
 *
 * @param t transition to be triggered
 * @param m marking (mapping of places to number of tokens)
 * @return  new marking representing the new state of the Petri net.
 *          It may or may not have been modified, depending if
 *          the transition was actually triggered (i.e all conditions
 *          at the arcs have been met).
 */

private trigger: Transition * map Place to nat ==> map Place to nat
trigger(t, m) == (
  dcl newMarking: map Place to nat := m;

  for all arc in set t.inputArcs do (
    if not arc.isEnabled(newMarking) then
      return m;
    newMarking := arc.trigger(t, newMarking);
  );

  for all arc in set t.outputArcs do (
    if not arc.isEnabled(newMarking) then
      return m;
    newMarking := arc.trigger(t, newMarking);
  );

  return newMarking;
)
pre t in set transitions and dom m subset places
post dom RESULT subset places;

/**
 * Triggers given transition and changes marking accordingly.
 *
 * @see trigger(t, m)
 *
 * @param t transition to be triggered
 * @return  nothing
 */

public executeStep: Transition ==> ()
executeStep(t) == (
  marking := trigger(t, marking);
```

```
)
pre t in set transitions;

/**
 * Verifies if a given marking is reachable from current state.
 *
 * Reachability definition (@wikipedia): "Given a computational
 * (potentially infinite state) system with a set of allowed rules
 * or transformations, decide whether a certain state of a system is
 * reachable from a given initial state of the system."
 *
 * @remark The behaviour is undefined if the Petri net is cyclic
 *
 * @param targetMarking marking (mapping of places to number of tokens)
 * @return boolean
 */

public isReachable: map Place to nat ==> bool
isReachable(targetMarking) == (
  dcl stack: DFSStack := new DFSStack();
  dcl currentInputMarking: map Place to nat := marking;
  dcl currentOutputMarking: map Place to nat := marking;
  dcl previousMarkings: set of map Place to nat := {};

  stack.push(new DFSElement(marking, new Transition("Start", { }, { })));

  while not stack.empty() do (

    currentInputMarking := stack.pop().marking;

    for all transition in set transitions do (

      currentOutputMarking := trigger(transition, currentInputMarking);

      /* if target marking is found */
      if (currentOutputMarking = targetMarking) then return true;

      /* if marking is not already explored */
      if (currentOutputMarking not in set previousMarkings) then (
        stack.push(new DFSElement(currentOutputMarking, transition));
        previousMarkings := previousMarkings union {currentOutputMarking};
      )
    );

  );

  return false;
)
pre dom targetMarking subset places;

/**
 * Get all sequences of transitions from an initial marking and
 * a target marking.
 *
 * @remark The behaviour is undefined if the Petri net is cyclic
 *
 * @param initialMarking initial marking (mapping of places to number of tokens)
 * @param targetMarking final marking (mapping of places to number of tokens)
 * @return set of seq of transitions
 */

public getAllSequences: map Place to nat * map Place to nat ==> set of seq of Transition
getAllSequences(initialMarking, targetMarking) == (
  dcl stack: DFSStack := new DFSStack();
  dcl currentDFSElement: DFSElement;
```

```
    dcl currentInputMarking: map Place to nat := initialMarking;
    dcl currentTransition: Transition;
    dcl currentOutputMarking: map Place to nat := initialMarking;
    dcl previousMarkings: set of map Place to nat := {};
    dcl transitionSeqs: set of seq of Transition := {};
    dcl sequencePath: seq of Transition := [];
    dcl newMarkingsGenerated: bool := false;
    dcl dummyTransition: Transition := new Transition("Start", { }, { });

    stack.push(new DFSElement(initialMarking, dummyTransition));

    while not stack.empty() do (

      currentDFSElement := stack.pop();
      currentTransition := currentDFSElement.transition;
      currentInputMarking := currentDFSElement.marking;

      newMarkingsGenerated := false;

      if (currentTransition <> dummyTransition) then
        sequencePath := [currentTransition] ^ sequencePath;

      for all transition in set transitions do (

        currentOutputMarking := trigger(transition, currentInputMarking);

        /* if target marking is found */
        if (currentOutputMarking = targetMarking and
         currentOutputMarking <> currentInputMarking and
         currentOutputMarking not in set previousMarkings) then (
          if (len sequencePath <> 0) then (
            transitionSeqs := transitionSeqs union {sequencePath ^ [transition]};
          );
        );

        /* if marking is not already explored */
        if (currentOutputMarking not in set previousMarkings) then (
          newMarkingsGenerated := true;
          stack.push(new DFSElement(currentOutputMarking, transition));
          previousMarkings := previousMarkings union {currentOutputMarking};
        );

        /* reached a dead end, remove sequence from path, a backtracking will occur */
        if (not newMarkingsGenerated) then (
          if (len sequencePath > 0) then
            sequencePath := tl sequencePath;
        );

      );

    );

    return transitionSeqs;
  )
  pre dom initialMarking subset places and dom targetMarking subset places;

end PetriNet
```

Function or operation	Line	Coverage	Calls
PetriNet	57	100.0%	9
executeStep	119	100.0%	20

getAllSequences	181	100.0%	3
isReachable	138	100.0%	18
trigger	90	100.0%	142
PetriNet.vdmpp		100.0%	192

6.3 Place

```
/**
 * Place of a Petri net with a maximum capacity
 */
class Place
  instance variables
    /**
     * Name of the place (debug/cosmetic purposes)
     */
    public label: seq of char;

    /**
     * Maximum number of tokens that this place can hold
     */
    public capacity: nat;

  operations
    /**
     * Place constructors.
     *
     * @param newLabel description of the place
     * @param newCapacity capacity of the place
     */

    public Place: seq of char * nat ==> Place
    Place(newLabel, newCapacity) == (
      label := newLabel;
      capacity := newCapacity;
      return self
    )
    pre newCapacity > 0;
end Place
```

Function or operation	Line	Coverage	Calls
Place	23	100.0%	27
Place.vdmpp		100.0%	27

6.4 Transition

```
/**
 * Transition of a Petri net
 */
class Transition
  instance variables
    /**
     * Name of the transition (debug/cosmetic purposes)
     */
    public label: seq of char;

    /**
     * Set of all input arcs to this transition
     */
    public inputArcs: set of InputArc;

    /**
     * Set of all output arcs from this transition
     */
    public outputArcs: set of OutputArc;

  operations
    /**
     * Transition constructors.
     *
     * @param newLabel description of the transition
     * @param ia set of connected input places
     * @param oa set of connected output places
     */

    public Transition: seq of char * set of InputArc * set of OutputArc ==> Transition
    Transition(newLabel, ia, oa) == (
      label := newLabel;
      inputArcs := ia;
      outputArcs := oa;
      return self;
    );
end Transition
```

Function or operation	Line	Coverage	Calls
Transition	29	100.0%	38
Transition.vdmpp		100.0%	38

6.5 DFSStack

```
class DFSStack
  instance variables
    stack: seq of DFSElement := [];
  operations

    public reset: () ==> ()
    reset() ==
      stack := [];

    public pop: () ==> DFSElement
    pop() ==
      def res = hd stack in
      (stack := tl stack;
      return res)
    pre stack <> [];


    public push: DFSElement ==> ()
    push(element) == (
      stack := [element] ^ stack;
    );


    public empty: () ==> bool
    empty() == (
      return len stack = 0
    );

end DFSStack

class DFSElement
  instance variables
    public marking: map Place to nat;
    public transition: Transition;
  operations

    public DFSElement: map Place to nat * Transition ==> DFSElement
    DFSElement(mrk, trans) == (
      marking := mrk;
      transition := trans;
      return self;
    );
end DFSElement
```

Function or operation	Line	Coverage	Calls
DFSElement	32	100.0%	55
empty	20	100.0%	59
pop	8	100.0%	53
push	15	100.0%	55
reset	5	0.0%	0
DFSStack.vdmpp		100.0%	22

7 Model validation

7.1 MyTestCase

```
/*
 * Superclass for test classes, simpler but more practical than VDMUnit`TestCase.
 * For proper use, you have to do: New -> Add VDM Library -> IO.
 * JPF, FEUP, MFES, 2014/15.
 */
class MyTestCase
operations

  /**
   * Simulates assertion checking by reducing it to pre-condition checking.
   * If 'arg' does not hold, a pre-condition violation will be signaled.
   */

  protected assertTrue: bool ==> ()
  assertTrue(arg) ==
    return
  pre arg;

  /**
   * Simulates assertion checking by reducing it to post-condition checking.
   * If values are not equal, prints a message in the console and generates
   * a post-conditions violation.
   */

  protected assertEqual: ? * ? ==> ()
  assertEqual(expected, actual) ==
    if expected <> actual then (
      IO`print("Actual value (");
      IO`print(actual);
      IO`print(") different from expected (");
      IO`print(expected);
      IO`println(")\n")
    )
  post expected = actual

end MyTestCase
```

Function or operation	Line	Coverage	Calls
assertEqual	23	38.8%	43
assertTrue	13	100.0%	18
MyTestCase.vdmpp		45.0%	61

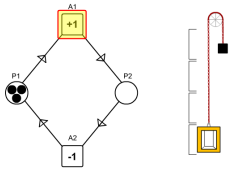
7.2 TestArc

```
class TestArc is subclass of MyTestCase

operations

  public testConstructors: () ==> ()
  testConstructors() == (
    IO`println("\t\t test constructor");
    let p = new Place(),
     wia = new WeightedInputArc(p, 1),
     ra = new ResetArc(p),
     ia = new InhibitorArc(p),
     woa = new WeightedOutputArc(p, 2) in (
      assertEqual(1, wia.weight);
      assertEqual(2, woa.weight);
      assertEqual(p, wia.place);
      assertEqual(p, woa.place);
      assertEqual(p, ra.place);
      assertEqual(p, ia.place);
    );
  );

  /**
   * Entry point that runs all tests with valid inputs (no test should fail)
   */

  public testAll: () ==> ()
  testAll() == (
    IO`println("\t arc tests");
    testConstructors();
  );

end TestArc
```

Function or operation	Line	Coverage	Calls
testAll	24	100.0%	1
testConstructors	4	100.0%	1
TestArc.vdmpp		100.0%	2

7.3 TestPetriNet

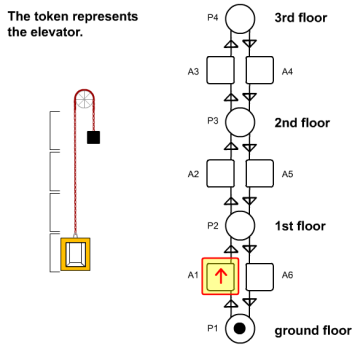
```
class TestPetriNet is subclass of MyTestCase

operations
  /**
   * Requirements: R1, R2.3, R2.4, R3, R4, R5
   */
```



The number of tokens represents the number of vertical movements an elevator can make upwards or downwards in a certain state of the system.

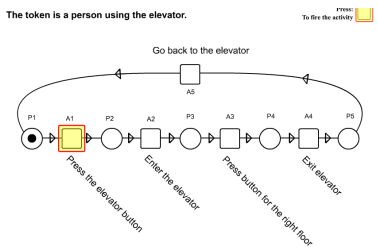
```
  public testElevator1Example: () ==> ()
  testElevator1Example() == (
    /* http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/aalst/elevator1.swf */
    IO`println("\t\t test elevator 1");
    let p1 = new Place("P1", 3),
     p2 = new Place("P2", 3),
     a1i = new WeightedInputArc(p1, 1),
     a1o = new WeightedOutputArc(p2, 1),
     a2i = new WeightedInputArc(p2, 1),
     a2o = new WeightedOutputArc(p1, 1),
     t1 = new Transition("+1", { a1i }, { a1o }),
     t2 = new Transition("-1", { a2i }, { a2o }),

     places = { p1, p2 },
     arcs = { a1i, a1o, a2i, a2o },
     marking = { p1 |-> 3, p2 |-> 0 },
     transitions = { t1, t2 },

     petriNet = new PetriNet(places, p1, {}, arcs, marking, transitions) in (

      assertEqual({ p1 |-> 3, p2 |-> 0 }, petriNet.marking);

      /* Test reachability of the markings before step execution */
      assertTrue(petriNet.isReachable({ p1 |-> 2, p2 |-> 1 }));
      assertTrue(petriNet.isReachable({ p1 |-> 1, p2 |-> 2 }));
      assertTrue(petriNet.isReachable({ p1 |-> 0, p2 |-> 3 }));
      assertTrue(not petriNet.isReachable({ p1 |-> 4, p2 |-> 5 }));

      assertEqual({}, petriNet.getAllSequences(
        { p1 |-> 3, p2 |-> 0 },{ p1 |-> 4, p2 |-> 5 })
      );
      assertEqual({[t1, t1]}, petriNet.getAllSequences(
        { p1 |-> 3, p2 |-> 0 }, { p1 |-> 1, p2 |-> 2 })
      );
      assertEqual({[t1, t1, t1]}, petriNet.getAllSequences(
        { p1 |-> 3, p2 |-> 0 }, { p1 |-> 0, p2 |-> 3 })
      );

      /* Test stepwise execution of the petri net */
      petriNet.executeStep(t1);
      assertEqual({ p1 |-> 2, p2 |-> 1 }, petriNet.marking);

      petriNet.executeStep(t1);
      assertEqual({ p1 |-> 1, p2 |-> 2 }, petriNet.marking);

      petriNet.executeStep(t1);
```

```
    assertEqual({ p1 |-> 0, p2 |-> 3 }, petriNet.marking);

    petriNet.executeStep(t2);
    assertEqual({ p1 |-> 1, p2 |-> 2 }, petriNet.marking);

    petriNet.executeStep(t1);
    assertEqual({ p1 |-> 0, p2 |-> 3 }, petriNet.marking);
  );
);

/**
 * Requirements: R1, R2.3, R2.4, R3, R4, R5
 */
```



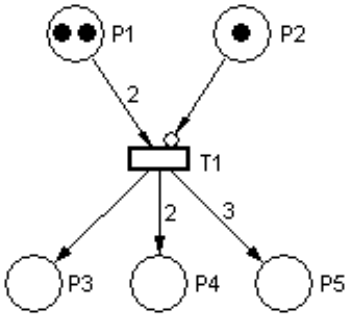

```
public testElevator2Example: () ==> ()
testElevator2Example() == (
  /* http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/aalst/elevator2.swf */
  IO`println("\t\t test elevator 2");
  let p1 = new Place("ground floor", 1),
   p2 = new Place("1st floor", 1),
   p3 = new Place("2nd floor", 1),
   p4 = new Place("3rd floor", 1),
   a1i = new WeightedInputArc(p1, 1),
   a1o = new WeightedOutputArc(p2, 1),
   a2i = new WeightedInputArc(p2, 1),
   a2o = new WeightedOutputArc( p3, 1),
   a3i = new WeightedInputArc(p3, 1),
   a3o = new WeightedOutputArc(p4, 1),
   a4i = new WeightedInputArc(p4, 1),
   a4o = new WeightedOutputArc(p3, 1),
   a5i = new WeightedInputArc(p3, 1),
   a5o = new WeightedOutputArc(p2, 1),
   a6i = new WeightedInputArc(p2, 1),
   a6o = new WeightedOutputArc(p1, 1),
   t1 = new Transition("A1", { a1i }, { a1o }),
   t2 = new Transition("A2", { a2i }, { a2o }),
   t3 = new Transition("A3", { a3i }, { a3o }),
   t4 = new Transition("A4", { a4i }, { a4o }),
   t5 = new Transition("A5", { a5i }, { a5o }),
   t6 = new Transition("A6", { a6i }, { a6o }),

   places = { p1, p2, p3, p4 },
   arcs = { a1i, a1o, a2i, a2o, a3i, a3o, a4i, a4o, a5i, a5o, a6i, a6o },
   marking = { p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0 },
   transitions = { t1, t2, t3, t4, t5, t6 },

   petriNet = new PetriNet(places, p1, {}, arcs, marking, transitions) in (

    /* Test reachability before stepwise execution */
    assertTrue(petriNet.isReachable({ p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0 }));
    assertTrue(petriNet.isReachable({ p1 |-> 0, p2 |-> 1, p3 |-> 0, p4 |-> 0 }));
```

```
    assertTrue(petriNet.isReachable({ p1 |-> 0, p2 |-> 1, p3 |-> 0, p4 |-> 0 }));

    assertEqual({ p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0 }, petriNet.marking);

    /* Test stepwise execution of the Petri net */
    petriNet.executeStep(t1); /* valid */
    assertEqual({ p1 |-> 0, p2 |-> 1, p3 |-> 0, p4 |-> 0 }, petriNet.marking);

    petriNet.executeStep(t2); /* valid */
    assertEqual({ p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 0 }, petriNet.marking);

    petriNet.executeStep(t5); /* valid */
    assertEqual({ p1 |-> 0, p2 |-> 1, p3 |-> 0, p4 |-> 0 }, petriNet.marking);

    let previousMarking = petriNet.marking in (
      petriNet.executeStep(t1); /* not valid, no marking change */
      assertEqual(previousMarking, petriNet.marking);
    )
  );
);

/**
 * Requirements: R1, R2.3, R2.4, R3, R4, R5
 */
```



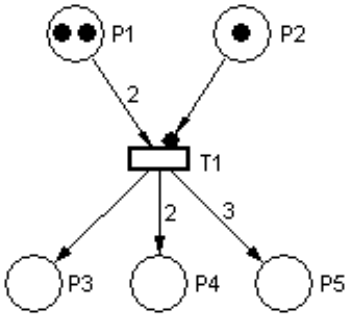

```
public testElevator3Example: () ==> ()
testElevator3Example() == (
  /* http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/aalst/elevator3.swf */
  IO`println("\t\t test elevator 3");
  let p1 = new Place("p1", 1),
   p2 = new Place("p2", 1),
   p3 = new Place("p3", 1),
   p4 = new Place("p4", 1),
   p5 = new Place("p5", 1),
   a1i = new WeightedInputArc(p1,  1),
   a1o = new WeightedOutputArc(p2, 1),
   a2i = new WeightedInputArc(p2, 1),
   a2o = new WeightedOutputArc(p3, 1),
   a3i = new WeightedInputArc(p3, 1),
   a3o = new WeightedOutputArc(p4, 1),
   a4i = new WeightedInputArc(p4, 1),
   a4o = new WeightedOutputArc(p5, 1),
   a5i = new WeightedInputArc(p5, 1),
   a5o = new WeightedOutputArc(p1, 1),
   t1 = new Transition("A1 - Press the elevator button", { a1i }, { a1o }),
   t2 = new Transition("A2 - Enter the elevator", { a2i }, { a2o }),
   t3 = new Transition("A3 - Press button for the right floor", { a3i }, { a3o }),
   t4 = new Transition("A4 - Exit elevator", { a4i }, { a4o }),
   t5 = new Transition("A5 - Go back to the elevator", { a5i }, { a5o }),

   places = { p1, p2, p3, p4, p5 },
   arcs = { a1i, a1o, a2i, a2o, a3i, a3o, a4i, a4o, a5i, a5o },
   marking = { p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 0 },
   transitions = { t1, t2, t3, t4, t5 },
```

```
    petriNet = new PetriNet(places, p1, {}, arcs, marking, transitions) in (

     /* Test reachability before stepwise execution */
     assertTrue(petriNet.isReachable({ p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 0 }));
     assertTrue(petriNet.isReachable({ p1 |-> 0, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 0 }));
     assertTrue(petriNet.isReachable({ p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 0, p5 |-> 0 }));
     assertTrue(petriNet.isReachable({ p1 |-> 0, p2 |-> 0, p3 |-> 0, p4 |-> 1, p5 |-> 0 }));
     assertTrue(petriNet.isReachable({ p1 |-> 0, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 1 }));

     assertEqual({ p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 0 }, petriNet.marking);

     /* Test stepwise execution of the petri net */
     petriNet.executeStep(t1); /* valid */
     assertEqual({ p1 |-> 0, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 0 }, petriNet.marking);

     petriNet.executeStep(t2); /* valid */
     assertEqual({ p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 0, p5 |-> 0 }, petriNet.marking);

     petriNet.executeStep(t3); /* valid */
     assertEqual({ p1 |-> 0, p2 |-> 0, p3 |-> 0, p4 |-> 1, p5 |-> 0 }, petriNet.marking);

     petriNet.executeStep(t4); /* valid */
     assertEqual({ p1 |-> 0, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 1 }, petriNet.marking);

     petriNet.executeStep(t5); /* valid */
     assertEqual({ p1 |-> 1, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 0 }, petriNet.marking);

  );
);

/**
 * Requirements: R1, R2.2, R2.3, R2.4, R3, R4
 */
```



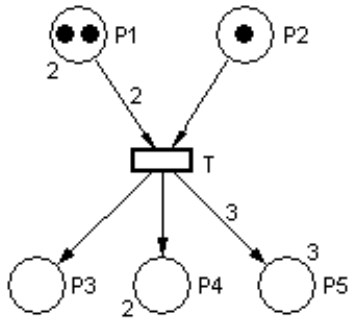

```
public testSimpleInhibitorArc: () ==> ()
testSimpleInhibitorArc() == (
  /* http://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq_files/image005.gif */
  IO`println("\t\t test simple inhibitor arc");
  let p1 = new Place("P1", 2),
   p2 = new Place("P2", 1),
   p3 = new Place("P3", 1),
   p4 = new Place("P4", 3),
   p5 = new Place("P5", 4),
   ia1 = new WeightedInputArc(p1,  2),
   ia2 = new InhibitorArc(p2),
   oa1 = new WeightedOutputArc(p3, 1),
   oa2 = new WeightedOutputArc(p4, 2),
   oa3 = new WeightedOutputArc(p5, 3),
   t1 = new Transition("T1", { ia1, ia2 }, { oa1, oa2, oa3 }),

   places = { p1, p2, p3, p4, p5 },
   arcs = { ia1, ia2, oa1, oa2, oa3 },
```

```
    marking1 = { p1 |-> 2, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 0 },
    marking2 = { p1 |-> 2, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 0 },
    transitions = { t1 },

    petriNet1 = new PetriNet(places, p1, {}, arcs, marking1, transitions),
    petriNet2 = new PetriNet(places, p1, {}, arcs, marking2, transitions) in (

     assertEqual({ p1 |-> 2, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 0 }, petriNet1.marking);
     assertEqual({ p1 |-> 2, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 0 }, petriNet2.marking);

     assertTrue(not petriNet1.isReachable(
       { p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 })
     );
     assertTrue( petriNet2.isReachable(
       { p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 })
     );

     petriNet1.executeStep(t1); /* should not trigger */
     petriNet2.executeStep(t1); /* should trigger */

     assertEqual({ p1 |-> 2, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 0 }, petriNet1.marking);
     assertEqual({ p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 }, petriNet2.marking);
  );
);

/**
 * Requirements: R1, R2.1, R2.3, R2.4, R3, R4
 */
```




```
public testSimpleResetArc: () ==> ()
testSimpleResetArc() == (
  /* http://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq_files/image003.gif (mod.) */
  IO`println("\t\t test simple reset arc");
  let p1 = new Place("P1", 2),
   p2 = new Place("P2", 1),
   p3 = new Place("P3", 1),
   p4 = new Place("P4", 3),
   p5 = new Place("P5", 4),
   ia1 = new WeightedInputArc(p1,  2),
   ia2 = new ResetArc(p2),
   oa1 = new WeightedOutputArc(p3, 1),
   oa2 = new WeightedOutputArc(p4, 2),
   oa3 = new WeightedOutputArc(p5, 3),
   t1 = new Transition("T1", { ia1, ia2 }, { oa1, oa2, oa3 }),

   places = { p1, p2, p3, p4, p5 },
   arcs = { ia1, ia2, oa1, oa2, oa3 },
   marking1 = { p1 |-> 2, p2 |-> 5, p3 |-> 0, p4 |-> 0, p5 |-> 0 },
   marking2 = { p1 |-> 2, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 0 },
   transitions = { t1 },

   petriNet1 = new PetriNet(places, p1, {}, arcs, marking1, transitions),
```

```
    petriNet2 = new PetriNet(places, p1, {}, arcs, marking2, transitions) in (

     assertEqual({ p1 |-> 2, p2 |-> 5, p3 |-> 0, p4 |-> 0, p5 |-> 0 }, petriNet1.marking);
     assertEqual({ p1 |-> 2, p2 |-> 0, p3 |-> 0, p4 |-> 0, p5 |-> 0 }, petriNet2.marking);

     assertTrue(petriNet1.isReachable({ p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 }));
     assertTrue(petriNet2.isReachable({ p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 }));

     petriNet1.executeStep(t1); /* should trigger */
     petriNet2.executeStep(t1); /* should trigger */

     assertEqual({ p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 }, petriNet1.marking);
     assertEqual({ p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 }, petriNet2.marking);
   );
);

/**
 * Requirements: R1, R2.3, R2.4, R3, R4
 */
```




```
public testSimpleCapacityArc: () ==> ()
testSimpleCapacityArc() == (
  /* http://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq_files/image003.gif (mod.) */
  IO`println("\t\t test simple capacity arc");
  let p1 = new Place("P1", 2),
   p2 = new Place("P2", 1),
   p3 = new Place("P3", 1),
   p4 = new Place("P4", 2),
   p5 = new Place("P5", 3),
   ia1 = new WeightedInputArc(p1,  2),
   ia2 = new WeightedInputArc(p2,  1),
   oa1 = new WeightedOutputArc(p3, 1),
   oa2 = new WeightedOutputArc(p4, 2),
   oa3 = new WeightedOutputArc(p5, 3),
   t1 = new Transition("T1", { ia1, ia2 }, { oa1, oa2, oa3 }),

   places = { p1, p2, p3, p4, p5 },
   arcs = { ia1, ia2, oa1, oa2, oa3 },
   marking1 = { p1 |-> 2, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 3 },
   marking2 = { p1 |-> 2, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 0 },
   transitions = { t1 },

   petriNet1 = new PetriNet(places, p1, {}, arcs, marking1, transitions),
   petriNet2 = new PetriNet(places, p1, {}, arcs, marking2, transitions) in (

    assertEqual({ p1 |-> 2, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 3 }, petriNet1.marking);
    assertEqual({ p1 |-> 2, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 0 }, petriNet2.marking);

    assertTrue(not petriNet1.isReachable(
      { p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 })
    );
    assertTrue(petriNet2.isReachable(
```

```
        { p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 })
      );

      petriNet1.executeStep(t1); /* should not trigger */
      petriNet2.executeStep(t1); /* should trigger */

      assertEqual({ p1 |-> 2, p2 |-> 1, p3 |-> 0, p4 |-> 0, p5 |-> 3 }, petriNet1.marking);
      assertEqual({ p1 |-> 0, p2 |-> 0, p3 |-> 1, p4 |-> 2, p5 |-> 3 }, petriNet2.marking);
    );
  );

  /* Entry point that runs all tests with valid inputs */

  public testAll: () ==> ()
  testAll() == (
    IO`println("\t petrinet tests");
    testElevator1Example();
    testElevator2Example();
    testElevator3Example();
    testSimpleInhibitorArc();
    testSimpleResetArc();
    testSimpleCapacityArc();
  );

end TestPetriNet
```

Function or operation	Line	Coverage	Calls
testAll	304	100.0%	1
testElevator1Example	4	100.0%	1
testElevator2Example	60	100.0%	1
testElevator3Example	118	100.0%	1
testSimpleCapacityArc	260	100.0%	1
testSimpleInhibitorArc	178	100.0%	1
testSimpleResetArc	221	100.0%	1
TestPetriNet.vdmpp		100.0%	7

7.4 TestPlace

```
class TestPlace is subclass of MyTestCase

operations

  public testConstructor: () ==> ()
  testConstructor() == (
    IO`println("\t\t test constructor");
    let p = new Place("test1", 1) in (
      assertEqual("test1", p.label);
      assertEqual(1,  p.capacity);
    );
  );

  /* Entry point that runs all tests with valid inputs */

  public testAll: () ==> ()
  testAll() == (
    IO`println("\t place tests");
    testConstructor();
  );

end TestPlace
```

Function or operation	Line	Coverage	Calls
testAll	14	100.0%	1
testConstructor	4	100.0%	1
TestPlace.vdmpp		100.0%	2

7.5 TestTransition

```
class TestTransition is subclass of MyTestCase

operations

  public testConstructor: () ==> ()
  testConstructor() == (
    IO`println("\t\t test constructor");
    let p = new Place(),
     wia = new WeightedInputArc(p, 1),
     woa = new WeightedOutputArc(p, 2),
     t = new Transition("test1", { wia }, { woa }) in (
      assertEqual("test1",  t.label);
      assertEqual({ wia }, t.inputArcs);
      assertEqual({ woa }, t.outputArcs);
    );
  );

  /* Entry point that runs all tests with valid inputs */

  public testAll: () ==> ()
  testAll() == (
    IO`println("\t transition tests");
    testConstructor();
  );

end TestTransition
```

Function or operation	Line	Coverage	Calls
testAll	18	100.0%	1
testConstructor	4	100.0%	1
TestTransition.vdmpp		100.0%	2

7.6 Tests

```
class Tests
  operations

    public static main: () ==> ()
    main() == (
      IO`println("Running tests");
      new TestPlace().testAll();
      new TestTransition().testAll();
      new TestArc().testAll();
      new TestPetriNet().testAll();
      IO`println("Tests finished");
    );
end Tests
```

Function or operation	Line	Coverage	Calls
main	3	100.0%	1
Tests.vdmpp		100.0%	1

Tests output:

```
** Overture Console
**
Running tests
     place tests
         test constructor
     transition tests
         test constructor
     arc tests
         test constructor
     petrinet tests
         test elevator 1
         test elevator 2
         test elevator 3
         test simple inhibitor arc
         test simple reset arc
         test simple capacity arc
Tests finished

Tests`main() = ()
```

8 Model verification

8.1 Example of Domain Verification

One of the proof obligation generated by Overture is:

No.	PO Name	Type
2	WeigthedInputArc‘isEnabled(map (Place) to (nat))	legal map application

The code under analysis (with the relevant map application underlined) is:

```
public isEnabled: map Place to nat ==> bool
isEnabled(marking) == (
  return marking(place) >= weight;
)
pre place in set dom marking;
```

The proof is trivial because the pre condition states that *place* must be in the domain of the map *marking*, therefore the map application must be valid when the operation is executed.

8.2 Example of Invariant Verification

One of the proof obligation generated by Overture is:

No.	PO Name	Type
7	PetriNet‘PetriNet(set of (Place), Place, set of (Place), set of (Arc), map (Place) to (nat), set of (Transition))	state invariant holds

The code under analysis is:

```
public PetriNet: set of Place * Place * set of Place * set of Arc *
                 map Place to nat * set of Transition ==> PetriNet
PetriNet(p, s, f, a, m, t) == (
  atomic(
    places := p; starts := s; finish := f;
    arcs := a; marking := m; transitions := t;
  );
  return self;
)
pre dom m = p and
  s in set p and
  f subset p and
  forall a1 in set a & a1.place in set p and
  forall t1 in set t & (
   forall a2 in set t1.inputArcs & a2 in set a and
   forall a3 in set t1.outputArcs & a3 in set a
  );
```

The relevant invariants under analysis are:

```
inv starts in set places; -- inv1
inv finish subset places; -- inv2
inv forall arc in set arcs & arc.place in set places; -- inv3
inv dom marking subset places; -- inv4
inv forall transition in set transitions & (
 transition.inputArcs subset arcs and
 transition.outputArcs subset arcs
); -- inv5
```

Invariants must hold true after and before the *atomic* block is executed.

Having the pre condition `s in set p` and the atomic block of code `places := p; starts := s;` implies that `starts in set places` which is directly translated into *inv1*.

Having the pre condition `f subset p` and the atomic block of code `places := p; finish := f;` implies that `finish subset places` which is directly translated into *inv2*.

The pre condition `forall a1 in set a & a1.place in set p` and the atomic block of code `places := p; arcs := a;` states that the place at each arc in the set of arcs must be in the set of place, that is `forall arc in set arcs & arc.place in set places` (*inv3*).

The 4th invariant says that `dom marking subset places`. Since `dom m = p` and `places := p; marking := m;` then we have `dom marking = places` which also means that `marking` is a subset of `places`.

inv5 verifies that all input and output arcs of transitions are a subset of arcs which can be verified because the precondition states that each input and output arc must be in the set of arcs, which obviously true.

9 Conclusions

9.1 Results

The developed model satisfies all the necessary requirements. All the Petri net extensions were implemented, such as inhibitor arcs and reset arcs.

9.2 Improvements

The reachability analysis and determination of all sequences of transitions do not detect cycles in the Petri net, however, since that problem is EXPSPACE-hard[1], it is not easy to implement.

Other mathematical properties of Petri nets could be implemented, such as the liveness and boundedness analysis.

Another improvement would be to be able to fire multiple transitions simultaneously as well as supporting multiple types of concurrency like interleaving or independence models [2].

9.3 Work division

All members of the group worked equally in the development of the project.

[1] decision problems solvable by a deterministic Turing machine in $O(2^{p(n)})$

References

[1] Murata, Tadao *Petri Nets: Properties, Analysis and Applications*, May 1988.

[2] Hayman, Jonathan M. *Petri net semantics*, University of Cambridge, Computer Laboratory, 2010.

[3] Overture, Overture Tool website, `http://overturetool.org/`. accessed on 22-12-2014.

[4] Peter G. Larsen and Kenneth Lausdahl and Nick Battle and John Fitzgerald and Sune Wolff. VDM-10 Language Manual. Overture. February 2011.