NPNtool: Modelling and Analysis Toolset for Nested Petri Nets

Leonid Dworzanski
Department of Software Engineering
National Research University Higher School of Economics
Moscow, Russia
leo@mathtech.ru

Daniil Frumin
Department of Software Engineering
National Research University Higher School of Economics
Moscow, Russia
difrumin@edu.hse.ru

***Abstract*—Nested Petri nets is an extension of Petri net formalism with net tokens for modelling multi-agent distributed systems with complex structure. While having a number of interesting properties, NP-nets have been lacking tool support. In this paper we present the NPNtool toolset for NP-nets which can be used to edit NP-nets models and check liveness in a compositional way. An algorithm to check m-bisimiliarity needed for compositional checking of liveness has been developed. Experimental results of the toolset usage for modelling and checking liveness of classical dinning philosophers problem are provided.**

***Index Terms*—Petri nets, nested Petri nets, multi-agent systems, compositionality, liveness**

I. INTRODUCTION

In our world distributed, multi-agent and concurrent systems are used everyday to the point that we don't even notice them working for us. Not only civilian and military air and water carriers are equipped with hi-tech electronics and software, but even laundry machines, microwave ovens, refrigerators, air-condition systems and other implements are controlled by distributed software.

In the great amount of research on defining parallel and concurrent systems, in recent years a range of formalisms have been introduced, modified or extended to cover agent systems. One of such approaches, which gained widespread usage, is Petri nets. One downside of the classical Petri nets formalism is its flat structure, while multi-agent systems commonly have complex nested apparatus. This prevents us from easily specifying models of multi-agent systems in a natural way. The solution to this problem was found by R. Valk [12], who originated the net-within-nets paradigm. According to the nets-within-net paradigm [11], the tokens in a Petri net can be nets themselves. Usually, there is some sort of hierarchy among the networks: there is a *system net*, the top level network, and all other nets are assigned each to their *initial place*, providing us with the hierarchy of the nets in one big *higher-order net*.

One of the non-flat Petri net model is Nested Petri nets [9], [10], [7]. In nested Petri nets (NP-nets), there is a *system net*, in some places of which *element nets* resign, in the form of *net tokens*. NP-nets have internal means of *synchronization* between element nets and the system net.

But the application and evolution of the formalism is hampered by the lack of tool support So far, there are no instruments (simulators, model checker software) which provide any kind of support for the nested Petri nets formalism. In this paper we present our newly developed project *NPNtool*

The paper is organized as follows. To start with, we give some necessary foundations of Petri nets and nested Petri nets. After that we describe our toolset (both frontend and backend). We describe a simple experiment we've conducted and conclude the paper with the directions of future research.

II. PETRI NETS

In literature, there is a variety of definitions for Petri nets, a common one would be the following.

Definition 1. *A* Petri net (P/T-net) *is a 4-tuple* (P, T, F, W) *where*

- P *and* T *are disjoint finite sets of* places *and* transitions*, respectively;*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of* arcs*;*
- $W:\ F \to \mathbb{N} \setminus 0$ – *an* arc multiplicity function*, that is, a function which assigns every arc a positive integer called an* arc multiplicity.

A *marking* of a Petri net (P, T, F, W) is a multiset over P, i.e. a mapping $M : P \to \mathbb{N}$. By $\mathfrak{M}(N)$ we denote a set of all markings of a P/T-net N.

We say that transition t in P/T-net $N = (P, T, F, W)$ is *active* in marking M iff for every $p \in \{p \mid (p, t) \in F\}$: $M(p) \geq W(p, t)$. An active transition may *fire*, resulting in a marking M', such as for all $p \in P$: $M'(p) = M(p) - W(p, t)$ if $p \in \{p \mid (p, t) \in F\}$, $M'(p) = M(p) - W(p, t) + W(t, p)$ if $p \in \{p \mid (t, p) \in F\}$ and $M'(p) = M(p)$ otherwise.

However, for our purpose we use a definition in algebraic representation. Firstly, we define a low-level abstract net

Definition 2. *A* Low-level Abstract Petri Net *is a 4-tuple* $(P, T, pre, post)$ *where*

- P *and* T *are disjoint finite sets of* places *and* transitions*, respectively;*
- $pre : T \to N(P)$ *is a precondition function;*
- $post : T \to N(P)$ *is a postcondition function;*

The research is partially supported by the Russian Fund for Basic Research (project 11-01-00737-a).

Here, $N : Set \to Set$ is a functor, defined by $N = G \circ F$, where F is a functor from the category of sets to the category of some structures $Struct$ and G is a forgetful functor from $Struct$ to Set.

Using this concept we can define P/T net as a low-level abstract Petri net where *Struct* is the category of commutative monoids and F maps each set x to a free monoid $F(x)$ over x [1].

This definition suggests for a straightforward embedding in Haskell:

```
data Net p t n m = Net
     { places :: Set p
     , trans :: Set t
     , pre :: t -> n p
     , post :: t -> n p
     , initial :: m p
     }
type PTNet = Net PTPlace Trans MultiSet MultiSet
type PTMark  = MultiSet PTPlace
type PTTrans = Int
type PTPlace = Int
```

III. NESTED PETRI NETS

In this section we define *nested Petri nets* (*NP-nets*) [9]. For simplicity we consider here only two-level NP-nets, where net tokens are usual Petri nets.

Definition 3. *A* nested Petri net *is a tuple* $(Atom, Expr, Lab, SN, (EN_1, \ldots, EN_k))$ *where*

- $Atom = Var \cup Con$ *– a set of atoms;*
- *Lab is a set of transition labels;*
- $(EN_1, \ldots, EN_k)$*, where* $k \geq 1$ *– a finite collection of P/T-nets, called* element nets*;*
- $SN = (P_{SN}, T_{SN}, F_{SN}, \upsilon, W, \Lambda)$ *is a high-level Petri net where*
  - P_{SN} *and* T_{SN} *are disjoint finite sets of* system places *and* system transitions *respectively;*
  - $F_{SN} \subseteq (P_{SN} \times T_{SN}) \cup (T_{SN} \times P_{SN})$ *is a set of* system arcs*;*
  - $\upsilon : P_{SN} \to \{EN_1, \ldots, EN_k\} \cup \{\bullet\}$ *is a* place typing *function;*
  - $W : F_{SN} \to Expr$ *is an* arc labelling *function, where Expr is the* arc expression language*;*
  - $\Lambda : T_{SN} \to Lab \cup \{\tau\}$ *is a* transition labelling *function, τ is the special "silent" label;*

The arc expression language *Expr* is defined as follows.

- *Con* is a set of *constants* interpreted over $A = A_{net} \cup \{\bullet\}$ and $A_{net} = \{(EN, m) \mid \exists i = 1, \ldots, k : EN = EN_i, m \in \mathfrak{M}(EN_i)\}$, i.e. A_{net} is a set of marked element nets, A is a set of element nets with markings and a regular black token $\bullet$ familiar to us from flat Petri nets (see section above);
- *Var* is a set of *variables*, we use variables x, y, z to range over *Var*.

Definition 4. *Expr is a language consisting of multisets over Con $\cup$ Var.*

The arc labeling function W is restricted in such way that constants or multiple instances of the same variable are not allowed in input arc expressions of the transition, constants and variables in the output arc expressions should correspond to the types of output places, and each variable in an output arc expression of the transition should occur in one of the input arc expressions of the transition.

We use notation like $x + 2y + 3$ to denote multiset $\{x, y, y, \bullet, \bullet, \bullet\}$.

A marking M in an NP-net *NPN* is a function mapping each $p \in P_{SN}$ to some (possibly empty) multiset $M(p)$ over A.

Let $Vars(e)$ denote a set of variables in an expression $e \in Expr$. For each $t \in T_{SN}$ we define $W(t) = \{W(x, y) \mid (x, y) \in F_{SN} \wedge (x = t \vee y = t)\}$ – all expressions labelling arcs incident to t.

Definition 5. *A* binding *b of a transition t is a function $b : Vars(W(t)) \to A$, mapping every variable in the t-incident arc expression to some value.*

We say that a transition t is *active* w.r.t. a binding b iff

$$\forall p \in \{p \mid (p, t) \in F_{SN}\} : b(W(p, t)) \subseteq M(p)$$

An active transition may *fire* (denoted $M \xrightarrow{t[b]} M'$) yielding a new marking $M'(p) = M(p) - b(W(p, t)) + b(W(t, p))$ for each $p \in P_{SN}$.

A behavior of an NP-net consists of three kinds of steps: system-autonomous step, element-autonomous step and synchronization step.

- An *element-autonomous step* is a firing of a transition in one of the element nets, which abides standart firing rules for P/T-nets.
- A *system-autonomous step* is a firing of a transition, labeled with τ, in the system net.
- A *(vertical) synchronization step* is a simultaneous firing of a transition, labeled with some $\lambda \in Lab$, in a system net together with firings of transitions, also labeled with λ, in all net tokens involved in (i.e. consumed by) this system net transition firing.

IV. USER INTERFACE

The modelling tool of the toolset consists of the meta-model of NP-nets and the tree-based editor which supports editing of NP-nets models. This tool is implemented via well-known modelling framework and code generation facility EMF (Eclipse Modeling Framework). The core of any EMF-based application is the EMF Ecore metamodel which describes domain-specific models. The crucial part of the developed NP-nets metamodel is depicted in fig. 1. The root element of the model is the instance of PetriNetNestedMarked class which represents marked NP-nets. TokenTypeElementNet class represents element nets. NetConstant class represents net constants which bound constants with marked element nets at the

---

[1] Since there is no commutative monoid datatype in Haskell, we use (isomorphic) representation via multisets.

time of NP-net model construction. We omit here the technical details of the remaining part of the metamodel. The metamodel resembles the formal definition of NP-nets given in section III.

The Tree-based editor for the developed metamodel is generated from the Ecore metamodel via EMF codegenerators and modified for the model specific needs. The editor takes care of standard model editing procedures like move, copy, delete, or create fragments of a model and provides undo/redo and serialization/deserialization support.

A NP-net model can be serialazed into XMI (XML Metadata Interchange) representation via the standard serialization mechanism of EMF. Serialized XMI documents are exported to the Haskell backend which carries out analysis procedures.

V. BACKEND

The backend for the tool is written in Haskell [5] and consists of the following parts:

- A library for constructing flat Petri nets;
- A library for constructing nested Petri net;
- Algorithms for checking compositional liveness of nested Petri nets [3];
- A CTL model checker for classical Petri nets;
- Communication layer.

We also make use of a number of GHC extensions which enrich the Haskell's type system.

A. Import

There are two ways to load models into the library: to load the XML file generated by the frontend or to construct the model using specialised library (see section V-B).

For parsing input we use the HXT [6] library based on Arrows [8]. We process the definitions into a `NPNConstr` code which is later converted to NP-net.

B. Dynamic construction

Libraries for dynamic construction of Petri nets are used in all the other modules of the system. To understand why they are useful, let's take a look at the straightforward definition of a Petri net using the datatype described in the section II.

```
pn1 :: PTNet
pn1 = Net { places = Set.fromList [1,2,3,4]
          , trans  = Set.fromList [t1,t2]
          , pre    = \(Trans x) -> case x of
               "t1" -> MSet.fromList [1,2]
               "t2"  -> MSet.fromList [1]
          , post   = \(Trans x) -> case x of
               "t1" -> MSet.fromList [3,4]
               "t2"  -> MSet.fromList [2]
          , initial = MSet.fromList [1,1,2,2]
          }
  where t1 = Trans "t1"
        t2 = Trans "t2"
```

However, it does get tedious after a while to write out all the nets this way. In addition, such approach is not modular or compositional. We've included a library with simple monadic interface for constructing P/T-nets.

The module `PTConstr` includes a monad `PTConstrM l` which is used for constructing P/T-nets, which transitions might be labelled with `l`. Among others it also includes the following functions:

```
mkPlace :: PTConstrM l PTPlace
mkTrans :: PTConstrM l PTTrans
label :: PTTrans -> l -> PTConstrM l ()
```

used for creating places and labelling transitions. In order to have more slick API we use Type Families [1] for providing the interface for arc construction:

```
class Arc k where
  type Co k :: *
  arc :: k -> Co k -> PTConstrM l ()
instance Arc Trans where
  type Co Trans = PTPlace
  arc = ...
instance Arc PTPlace where
  type Co PTPlace = Trans
  arc = ...
```

This allows us to uniformly use `arc` for constructing arcs both from transitions to places and from places to transitions, as shown in the example:

```
pn3 :: PTNet
pn3 = run \$ do
  [t1,t2] <- replicateM 2 mkTrans
  [p1,p2] <- replicateM 2 mkPlace
  label t1 "L1"
  arc p1 t1
  arc p1 t2
  arc t1 p2
  arc t2 p2
```

Furthermore, this allows us to take advantage of type polymorphism and define functions such as

```
arcn :: Arc k => k -> Co k -> Int -> PTConstrM l ()
arcn a b n = replicateM_ n \$ arc a b
```

Similar library for constructing nested Petri nets – `NPNConstr` – also has facilities for lifting `PTConstrM` code into `NPNConstrM` monad, which allows for better code reuse.

C. Algorithms

Algorithmically we have implemented a CTL model checker (as shown in [2]) with memoization, algorithm for determining the existence of m-bisimilarity (the algorithm is shown Appendix A) and liveness algorithms (as shown in [4]) which are used for checking liveness in a compositional way.

Definition 6 (Liveness)**.** *A net N is called* live *if every transition t in its system net is live, eg:* $\forall m \in \mathfrak{M}(N).\exists\sigma \in T^*.m \xrightarrow{\sigma} m' \wedge m' \xrightarrow{s} m'' \wedge t \in s$

Theorem 1. *Let NPN be a marked NP-net with a system net SN and initial marking m_0. Let also NPN satisfy the following conditions:*

1) *$(SN, m_0\,|_{SN})$ is live (if considered as a separate component);*
2) *all net tokens in m_0 and all net constants in every arc expression in NPN are live (if considered as separate components);*
3) *for each net token α in m_0, residing in a place p, α (if considered as a separate component) is m-bisimilar to the α-trail net of p.*

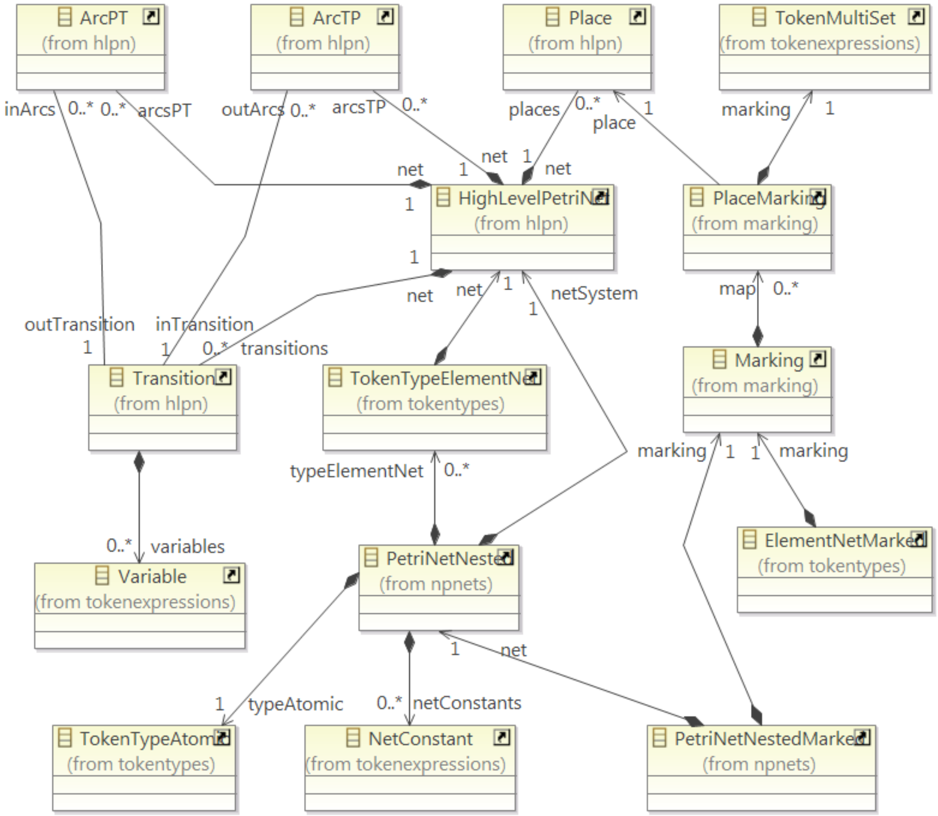

Fig. 1. The EMF Ecore metamodel of NP-nets

Then (NPN, m_0) *is is live.*

For proof of this theorem, definition of α-trail net and algorithm for its construction see [3]. In out project we've implemented the α-trail net construction algorithm and developed the m-bisimilarity checking algorithm (see section A).

VI. EXPERIMENT

For our experiment we decided to check liveness in a compositional way [3] on the following examples: the example net from [3] was checked instantly, due to it's facile structure.

We've decided to test our tool on the classical problem of dining philosophers extended with the ability of philosophers to walk: walking philosophers. In our modification philosophers are modeled as separate agents who may exist in different states. Thinking is an important philosophical activity, but who would turn down an opportunity to have a nice walk after a pleasant meal? Therefore philosophers can be either thinking, walking or eating.

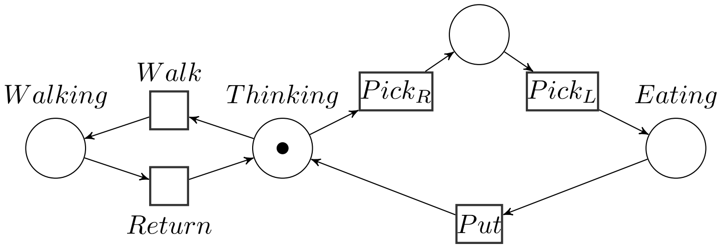

Fig. 2. $philAgent$ - A net token representing a single philosopher

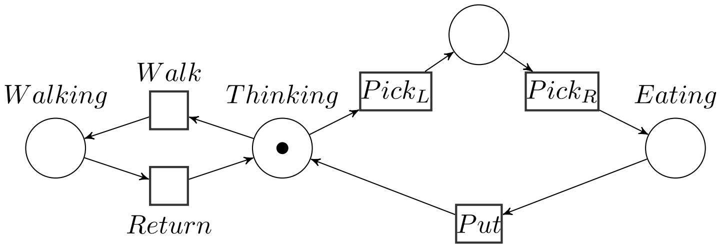

Fig. 3. $lastPhilAgent$ - A net token representing the last philosopher

Given a table with n philosophers and n forks, a net, modeling the first $n-1$ philosophers is shown in figure 2. However, the n-the philosopher is left-handed, and his net is a little bit different (see Fig. 3).

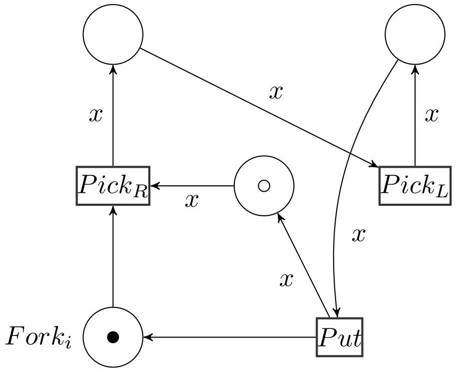

Fig. 4. *phil* - A portion of net representing a philosopher and his right fork

The system net consists of a number of repeated pieces. First $n-1$ pieces are shown in Fig. 4 and connected in the following way: for each i there is an arc from $Fork_{i+1}$ to $Pick_{R_i}$ and an arc from Put_i to $Fork_{i+1}$. The last piece looks somewhat differently (see Fig. 5) and have arcs from $Fork_1$ to $Pick_{L_n}$ and from Put_n to $Fork_1$.

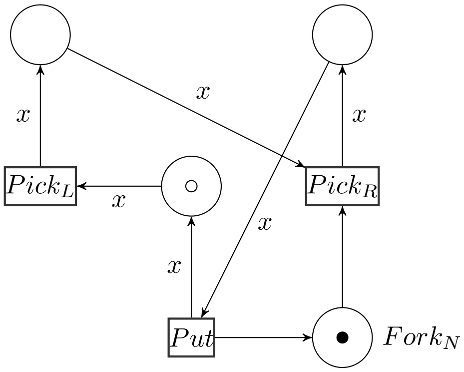

Fig. 5. *lastPhil* - A portion of net representing the last philosopher and his right fork

This system modeled via both interfaces. Firstly the system of 5 philosophers modelled via the frontend modeling tool. We also use API of the backend to automatically generate several system instances with different amount of philosophers and check their liveness.

Due to the modular nature of this task, it was easy to encode it using the construction library from the previous section. The code for the problem is shown in the appendix.

We've verified the compositional liveness of the system for $n = 3, 5, 7, 11$ and got the following results:

Number of philosophers	3	5	7	11
Mean execution time	8.23ms	144.9ms	2.17s	415.5s

The tests were performed using the `criterion` library on the 1.66GHz machine with 993mb RAM running Linux 3.5.0. The data was collected from 20 samples for each test.

VII. CONCLUSIONS AND FURTHER WORK

In this paper we have presented *NPNtool* – a support program for nested Petri nets formalism, capable of modeling NP-nets, checking them for liveness in a compositional way, model checking separate components for CTL specifications. We have also developed an algorithm for checking m-bisimilarity needed for liveness. The toolset can be used in both ways - to create and check models with the usage of the NP-nets editor and with the usage of the Haskell-based backend API.

The case study was presented in which we showed how to model NP-nets in a modular way, by modeling a "walking philosophers" problem and testing our tool against it.

Our future works directions includes: implementing a nCTL model checker, implementing a remote simulator. Tree based editor is pretty convenient to create or modify a model, however it is not very helpful to get quick overview of the model or its fragment. So the next step is to implement graphical editor of NP-nets diagrams.

We also intend this tool to be used as a framework for implementing algorithms on nested Petri nets.

APPENDIX A
ALGORITHM FOR CHECKING M-BISIMILARITY

Algorithm 1: $mBisim$ – checking for existence of a m-bisimilarity relation

```
Data: Two nets $pt_1$, $pt_2$ with their labelling functions
      $l_1$,$l_2$ and initial markings $m_1$,$m_2$. $R$ of type
      $\mathfrak{M}(pt_1) \times \mathfrak{M}(pt_2)$ is a relation we are building
      (initially empty).
Result: $True$ if nets are m-bisimilar, $False$ otherwise
begin
    if $(m_1, m_2) \in R$ then
        return $True$
    $Ts_1 \leftarrow \{t \mid t \in trans(pt1) \wedge enabled(pt_1, m_1, t)\}$
    $Ts_2 \leftarrow \{t \mid t \in trans(pt2) \wedge enabled(pt_2, m_2, t)\}$
    insert $(m1, m2)$ in $R$
    for $t \in Ts_1$ do
        $l \leftarrow l_1(t)$
        $m_1' \leftarrow fire(pt_1, m_1, t)$
        $nodes \leftarrow \{n \mid n \in \mathfrak{M}(pt_2) \wedge m_2 \overset{l}{\Rightarrow} n\}$
        if $null(nodes)$ then
            return $False$
        return $\bigwedge\{mBisim(pt1, pt2, l1, l2, m_1', m_2', R) \mid$
        $m_2' \in nodes\}$
    for $t \in Ts_2$ do
        $l \leftarrow l_2(t)$
        $m_2' \leftarrow fire(pt_2, m_2, t)$
        $nodes \leftarrow \{n \mid n \in \mathfrak{M}(pt_1) \wedge m_1 \overset{l}{\Rightarrow} n\}$
        if $null(nodes)$ then
            return $False$
        return $\bigwedge\{mBisim(pt1, pt2, l1, l2, m_1', m_2', R) \mid$
        $m_2' \in nodes\}$
```

The algorithm is implemented using the `StateT (Set (PTMark,PTMark)) Maybe` monad which allows for a more or less direct translation of the above code.

APPENDIX B
WALKING PHILOSOPHERS

```
import NPNTool.PTConstr
import NPNTool.NPNConstr
  (arcExpr, liftPTC, liftElemNet
  , addElemNet, NPNConstrM)
import qualified NPNTool.NPNConstr as NPC

-- Labels
data ForkLabel = PickR | PickL | Put
                deriving (Show,Eq,Ord)
-- Variables
data V = X -- we only need one
      deriving (Show,Eq,Ord)

-- Code for a single philosopher-agent
philAgent :: PTConstrM ForkLabel ()
philAgent = do
  ...

-- Code for the n-th philosopher
lastPhilAgent :: PTConstrM ForkLabel ()
lastPhilAgent = do
  ...

-- returns (Fork_i,PickL_i,Put_i)
phil :: NPNConstrM
        ForkLabel V (PTPlace,Trans,Trans)
phil = do
  [fork,p1,p2,p3] <- replicateM 4 NPC.mkPlace
  [pickL,pickR,put] <- replicateM 3 NPC.mkTrans
  -- get the philAgent token
  agent <- liftElemNet philAgent
  let x = Var X
  NPC.label pickL PickL
  NPC.label pickR PickR
  NPC.label put Put
  -- mark the Fork position with a single token
  NPC.mark fork (Left 1)
  -- mark the philosopher position with an agent
  NPC.mark p1 (Right agent)


  NPC.arc fork pickR
  NPC.arcExpr p1 x pickR
  NPC.arcExpr pickR x p2
  NPC.arcExpr p2 x pickL
  NPC.arcExpr pickL x p3
  NPC.arcExpr p3 x put
  NPC.arcExpr put x p1
  NPC.arc put fork

  return (fork,pickL,put)

lastPhil :: NPNConstrM ForkLabel V (PTPlace,Trans,Trans)
lastPhil = do
  ...

cyclePhils :: Int -> NPNConstrM ForkLabel V ()
cyclePhils n = do
  (fork1,pickL1,put1) <- phil
  (pickL,put) <- midPhils (n-2) (pickL1,put1)
  (forkLast,pickLLast,putLast) <- lastPhil
  lift $ do
    NPC.arc put forkLast
    NPC.arc forkLast pickL
    NPC.arc fork1 pickLLast
    NPC.arc putLast fork1

midPhils :: Int -> (Trans,Trans)
            -> NPNConstrM ForkLabel V (Trans,Trans)
midPhils n interf | n == 0     = return interf
                  | otherwise = do
  (pl,put) <- midPhils (n-1) interf
  (f',pl',put') <- phil
 NPC.arc put f' >> NPC.arc f' pl
  return (pl',put')


diningPhils :: Int -> NPNet ForkLabel V Int
diningPhils n = NPC.run (cyclePhils n) NPC.new
```

REFERENCES

[1] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, Simon Marlow, *Associated Types with Class*. – Proceedings of The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05), ACM Press, 2005.

[2] Edmund M. Clarke, Orna Grumberg, Doron Peled. *Model Checking*, MIT Press, 2001.

[3] L. W. Dworzaski, I. A. Lomazova, *On Compositionality of Boundedness and Liveness for Nested Petri Nets*. – Fundamenta Informaticae, Vol. 120. No. 3-4. pp. 275-293, 2012.

[4] Serge Haddad, Francois Vernadat, *Analysis Methods for Petri Nets*. – In *Petri Nets: Fundamental Models, Verification and Application*, edited by Michel Diaz, Wiley-ISTE, 656 p., 2009.

[5] Haskell Programming Language, http://haskell.org

[6] Haskell XML Toolkit, http://www.fh-wedel.de/~si/HXmlToolbox/index.html

[7] K. M. van Hee, I. A. Lomazova, O. Oanea, A. Serebrenik, N. Sidorova, M. Voorhoeve, *Nested nets for adaptive systems*. – Proceedings of the 27th international conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN'06), pp. 241-260, Springer, 2006.

[8] John Hughes, *Generalising Monads to Arrows*. – Science of Computer Programming 37, 2000.

[9] I. A. Lomazova, *Nested Petri nets: Modeling and analysis of distributed systems with object structure*. – Moscow:Scientific World, 208 p., 2004.

[10] I. A. Lomazova, *Nested Petri Nets for Adaptive Process Modeling*. – Lecture Notes in Computer Science, volume 4800. pp. 460-474, Springer, 2008.

[11] M. Mascheroni, F. Farina, *Nets-Within-Nets paradigm and grid computing*. – Transactions on Petri Nets and Other Models of Concurrency V, pp. 201-220. Springer, Heidelberg, 2012.

[12] R. Valk, *Petri Nets as Token Objects: An Introduction to Elementary Object Nets*. –ICATPN 1998. LNCS, vol. 1420, pp. 1-25. Springer, Heidelberg, 1998.