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

Danil 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 NPTool 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-bisimilarity needed for compositional checking of liveness has been developed. Experimental results of the toolset usage for modelling and checking liveness of classical dining 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.

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

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 NPTool.

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 \((P,T,\text{pre},\text{post})\)
where
- \(P\) and \(T\) are disjoint finite sets of places and transitions, respectively;
- \(\text{pre} : T \to \mathfrak{P}(P)\) is a precondition function;
- \(\text{post} : T \to \mathfrak{P}(P)\) is a postcondition function;
Here, $N : \text{Set} \to \text{Set}$ 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 $\text{Struct}$ and $G$ is a forgetful functor from $\text{Struct}$ to $\text{Set}$.

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

This definition suggests for a straightforward embedding in Haskell:

```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 PTPlace = Int
type PTTrans = Int
type PTMark = MultiSet PTPlace
type PTNet = Net PTPlace Trans MultiSet MultiSet
```

III. NESTED PETRI NETS

In this section we define nested Petri nets (NP-nets) \(^1\). For simplicity we consider here only two-level NP-nets, where net tokens are usual Petri nets.\(^2\)

Definition 3. A nested Petri net is a tuple $(\text{Atom}, \text{Expr}, \text{Lab}, SN, (EN_1, \ldots, EN_k))$ where
- $\text{Atom} = \text{Var} \cup \text{Con} \cup \{\cdot\} \cup \{\cdot\}$ is a set of atoms;
- $\text{Lab}$ is a set of transition labels;
- $(EN_1, \ldots, EN_k)$, where $k \geq 1$ – a finite collection of PT-nets, called element nets;
- $SN = (P_{SN}, T_{SN}, F_{SN}, \Lambda, W, \Lambda)$ 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;
  - $\Lambda : P_{SN} \to \{EN_1, \ldots, EN_k\} \cup \{\cdot\}$ is a place typing function;
  - $W : F_{SN} \to \text{Expr}$ is an arc labelling function, where $\text{Expr}$ is the arc expression language;
  - $\Lambda : T_{SN} \to \text{Lab} \cup \{\tau\}$ is a transition labelling function, $\tau$ is the special "delay" label;

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

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

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

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 $\{x, y, y, \cdot, \cdot, \cdot\}$.

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

Definition 5. A binding $b$ of a transition $t$ is a function $b : \text{Vars}(W(t)) \to A$, 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 $M \overset{t}{\longrightarrow} M'$) yielding a new marking $M' = M - b(W(p, t)) + b(W(t, p))$ 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 PT-nets.
- A system-autonomous step is a firing of a transition, labeled with $\tau$, in the system net.
- A (vertical) synchronization step is a simultaneous firing of a transition, labeled with some $\lambda \in \text{Lab}$, in a system net together with firings of transitions, also labeled with $\lambda$, 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 metamodel 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. [I]. 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...
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 serialized into XMI (XML Meta-
data 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 [11]

\[ \text{pn1} :: \text{PTNet} \]

\[
\text{pn1} = \text{Net} \\{ \text{places} = \text{Set.fromList [1,2,3,4]} , \text{trans} = \text{Set.fromList [t1,t2]} , \\
\text{pre} = \lambda (\text{Trans x}) \to \text{case x of} \\
\text{"t1"} \to \text{MSet.fromList [1,2]} \\
\text{"t2"} \to \text{MSet.fromList [3]} , \\
\text{post} = \lambda (\text{Trans x}) \to \text{case x of} \\
\text{"t1"} \to \text{MSet.fromList [1,2]} \\
\text{"t2"} \to \text{MSet.fromList [2]} , \\
\text{initial} = \text{MSet.fromList [1,1,2,2]} \\}
\]

\[
\text{where} \quad \text{t1} = \text{Trans "t1"} \\
\text{t2} = \text{Trans "t2"} 
\]

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 \( \mathfrak{I} \)
which is used for constructing P/T-nets, which transitions
might be labelled with 1. Among others it also includes the
following functions:

\[
\begin{align*}
\text{mkPlace} :: & \text{PTConstrM l PTPlace} \\
\text{mkTrans} :: & \text{PTConstrM l PTTrans} \\
\text{label} :: & \text{PTTrans \to l \to PTConstrM l ()}
\end{align*}
\]

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:

\[
\begin{align*}
\text{class Arc} & \text{ k where} \\
\text{type Co} & \text{ k :: \ast} \\
\text{arc ::} & \text{ k \to Co k \to PTConstrM l ()}
\end{align*}
\]

instance Arc \text{Trans} where

\[
\begin{align*}
\text{type Co} & \text{ Trans = PTPlace} \\
\text{arc} & \text{=} \ldots
\end{align*}
\]

instance Arc \text{PTPlace} where

\[
\begin{align*}
\text{type Co} & \text{PTPlace = Trans} \\
\text{arc} & \text{=} \ldots
\end{align*}
\]

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

\[
\begin{align*}
\text{pn3} :: & \text{PTNet} \\
\text{pn3} & \text{=} \text{run} \{\} \text{do} \\
& [t1,t2] \leftarrow \text{replicateM 2 mkTrans} \\
& [p1,p2] \leftarrow \text{replicateM 2 mkPlace} \\
& \text{label t1 "L1"} \\
& \text{arc} p1 t1 \\
& \text{arc} p1 t2 \\
& \text{arc} t1 p2 \\
& \text{arc} t2 p2
\end{align*}
\]

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

\[
\begin{align*}
\text{arcn} :: & \text{Arc} k \Rightarrow k \Rightarrow Co k \Rightarrow Int \Rightarrow PTConstrM l () \\
\text{arcn} & \text{a b n = replicateM n \& \& arc a b}
\end{align*}
\]

Similar library for constructing nested Petri nets –
NPNConstr – also has facilities for lifting \text{PTConstrM} code
into NPNConstr\( \mathfrak{I} \) 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: \( \forall m \in \mathcal{M}(N), \exists \sigma \in T^* : m \xrightarrow{\sigma} m' \wedge m' \xrightarrow{\tau} m'' \wedge t \in s \)

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 compo-
nent);
2) all net tokens in \( m_0 \) and all net constants in every arc
expression in NPN are live (if considered as separate compo-
nents);
3) for each net token \( \alpha \) in \( m_0 \), residing in a place \( p \), \( \alpha \) (if
considered as a separate component) is m-bisimilar to
the \( \alpha \)-trail net of \( p \).
Then \((NPN, m_0)\) is is live.

For proof of this theorem, definition of \(\alpha\)-trail net and algorithm for its construction see [3]. In our project we’ve implemented the \(\alpha\)-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 its 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.

Given a table with \(n\) philosophers and \(n\) forks, a net modeling the first \(n-1\) philosophers is shown in figure 2. However, the \(n\)-th philosopher is left-handed, and his net is a little bit different (see Fig. 3).
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 \( Pick_R \), 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 \( Fork_1 \).

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:

<table>
<thead>
<tr>
<th>Number of philosophers</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean execution time</td>
<td>8.23ms</td>
<td>144.9ms</td>
<td>2.17s</td>
<td>415.5s</td>
</tr>
</tbody>
</table>

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 NPTool – 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 \( \mathcal{M}(pt_1) \times \mathcal{M}(pt_2) \) 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

\( Ts_1 \leftarrow \{ t \mid t \in \text{trans}(pt_1) \land \text{enabled}(pt_1, m_1, t) \} \)
\( Ts_2 \leftarrow \{ t \mid t \in \text{trans}(pt_2) \land \text{enabled}(pt_2, m_2, t) \} \)
insert \( (m_1, m_2) \) in \( L \)
for \( t \in Ts_1 \) do
    \( l \leftarrow l_1(t) \)
    \( m'_1 \leftarrow \text{fire}(pt_1, m_1, l) \)
    \( \text{nodes} \leftarrow \{ n \mid n \in \mathcal{M}(pt_2) \land m_2 \Downarrow n \} \)
    \( \text{if null(nodes)} \) then
        return False
    return \( \bigwedge \{ mBisim(pt_1, pt_2, l_1, l_2, m'_1, m'_2, R) \mid m'_2 \in \text{nodes} \} \)
for \( t \in Ts_2 \) do
    \( l \leftarrow l_2(t) \)
    \( m'_2 \leftarrow \text{fire}(pt_2, m_2, l) \)
    \( \text{nodes} \leftarrow \{ n \mid n \in \mathcal{M}(pt_1) \land m_1 \Downarrow n \} \)
    \( \text{if null(nodes)} \) then
        return False
    return \( \bigwedge \{ mBisim(pt_1, pt_2, l_1, l_2, m'_1, m'_2, R) \mid m'_1 \in \text{nodes} \} \)
end
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**

```haskell
import NFNPTool.NPNConstr
import NFNPTool.NPNConstrM
import qualified NFNPTool.NPNConstr as NPC

-- Labels
data ForkLabel = PickR | PickL -- Labels

-- Code for the n-th philosopher
lastPhilAgent :: PTConstrM ForkLabel
lastPhilAgent = do
    NPC.arc put forkLast
    NPC.arc forkLast pickL
    NPC.arc fork1 pickLLast
    NPC.arc lastPhil putLast

-- Code for a single philosopher-agent
philAgent :: PTConstrM ForkLabel
philAgent = do
    NPC.arc put fork
    NPC.arc fork pickR
    NPC.arcExpr p1 x pickR
    NPC.arcExpr p2 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)

-- returns (Fork_i,PickL_i,Put_i)
phil :: NPNConstrM ForkLabel V
phil = do
    [forkLast,pickLLast,putLast] <- replicateM 4 NPC.mkPlace
    [pickL,turnpickR,put] <- replicateM 3 NPC.mkTrans
    NPC.label pickL PickL
    NPC.label pickR PickR
    NPC.label put Put

    let x = Var X
    NPC.mkTrans [pickL] [pickR,put] -- mark the Fork position with a single token
    NPC.mkTrans [fork] [Left] -- mark the philosopher position with an agent
    NPC.mkTrans [p1] [Right agent]

    return (fork,pickL,put)

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