Bounded Verification of Petri Nets and EOSs using Telingo: an Experience Report

Francesco Di Cosmo[1], Tephilla Prince[2]

[1]Free University of Bozen-Bolzano, Bolzano, Italy
[2]IIT Dharwad, Dharwad, India

Abstract

We report our preliminary results in implementing a tool to simulate and analyze Petri Nets and Elementary Object Systems from nets-within-net using the temporal ASP solver Telingo.

Keywords

Nets-within-nets, Verification, Temporal Equilibrium Logic

1. Introduction

Petri Nets (PNs) are a standard model of concurrent computation, introduced by Adam Petri in 1962 as a generalization of Finite State Automata [1]. At their barebones, they consist of a finite number of places hosting black tokens and a finite set of transitions that remove, add, and move the tokens on the places. Nowadays, several generalizations of PNs are available, which introduce additional features interesting for modeling. These range from simple inhibitory rules for the transitions [2] to sophisticated tokens carrying data items or additional structures. A notable example are nets-within-nets, a paradigm in which tokens can in turn carry a full-fledged PN, possibly obtaining several levels of nesting [3]. The simplest type of nets-within-nets are arguably Elementary Object Systems (EOSs), where the nesting is restricted to only two levels. Even if relatively simple, EOSs are appealing for modeling complex scenarios and multi-agent systems (MAS). In fact, the tokens at the higher level naturally represent agents, with their own internal state represented by the carried PN.

Recent works proposed EOS as a natural model to study robustness of MAS against spontaneous agent break-downs [4]. Under the let-it-crash perspective (as popularized by the Erlang language [5]), any perturbation results in completely breaking the functionalities of the agent, which stops performing actions of any kind. This is captured by inducing a deadlock of the carried PN, specifically by removing at once all its tokens. This approach is reminiscent of lossy PNs, i.e., PNs which, on top of to the standard dynamics, may non-deterministically loose some (possibly all) of their tokens [6]. Lossy PNs are less expressive than standard PNs and, in fact, enjoy easier verification problems. In our recent work[1] we applied the concept of PN lossiness to EOS, so as to capture less disruptive perturbations possibly resulting in a just partial degradation of the agent. In this setting, one can analyze EOS runs suffering of at most a fixed amount of losses (possibly unboundedly many) and attempt to check, e.g., reachability/coverability problems over lossy-runs. These problems are useful to determine whether a bad configuration cannot be reached as long as the number of perturbations is limited. Unfortunately, there is a lack of tools dedicated to (lossy) EOS. Our aim is to bridge this gap.

In this short paper, we report about our ongoing experience in implementing a prototype for the simulation and analysis of lossy-PNs and lossy-EOSs based on Answer Set Programming (ASP) technologies. Specifically, we aim at simulating and analyzing bounded lossy-EOS runs using Telingo [7], a specialization of the ASP system Clingo [8] to temporal domains. Implementations of PN variants in Clingo are [9] and [10]. However, none of them takes lossy PNs and EOSs in consideration and does

CILC'24: 39th Italian Conference on Computational Logic, June 26-28, 2024, Rome, Italy
francesco.dicosmo@unibz.it (F. D. Cosmo); tephilla.prince.18@iitdh.ac.in (T. Prince)
0000-0002-5692-5681 (F. D. Cosmo); 0000-0002-1045-3033 (T. Prince)

[1]Under review at an international workshop.

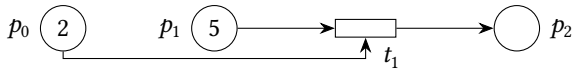

Figure 1: The PN with initial marking in Ex. 1.

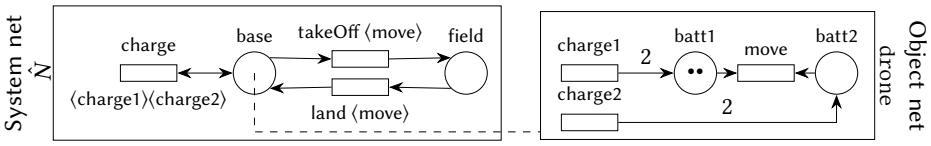

Figure 2: EOS in Example 2 with marking $\{\{\langle \texttt{drone}, \{\{\texttt{batt1}, \texttt{batt1}\}\}\rangle\}\}$. The idle transitions are omitted.

not experiment with the temporal features of Telingo. In fact, the most appealing feature of Telingo is its support in constraints of Temporal Equilibrium Logic over finite traces (TEL_f) formulas, which we expect to be an excellent tool to flexibly control the amount of lossiness in the simulated runs.

2. Preliminaries

2.1. EOSs

A PN (see [11]) is a tuple $N = (P_N, T_N, F_N)$ where P_N is a finite place set, T_N is a finite transition set (disjoint from P_N), and $F_N : (P_N \times T_N) \cup (T_N \times P_N) \longrightarrow \mathbb{N}$ is a flow function. Where useful, we equivalently interpret F_N via the functions $\texttt{pre}_N : T_N \to (P_N \to \mathbb{N})$ where $\texttt{pre}_N(t)(p) = F(p,t)$ and $\texttt{post}_N : T_N \to (P_N \to \mathbb{N})$ where $\texttt{post}_N(t)(p) = F(t,p)$. PN configurations are called markings, i.e., (finite) multisets $\mu : P \longrightarrow \mathbb{N}$ of places. We denote markings in set notation between {{ and }}, possibly by prefixing the places with their multiplicity.

Example 1. Fig. 1 depicts the PN $(\{p_0, p_1, p_2\}, \{t_1\}, F)$ where $F(p_0, t_1) = F(p1, t_1) = F(t_1, p_2) = 1$ and $F(t_1, p_0) = F(t_1, p_1) = F(p_2, t_1)$, with initial marking $\mu_0 = \{\{2p_0, 5p_1\}\}$. The net reaches $\mu_1 = \{\{3p_0, 2p_2\}\}$, covers $\mu_2 = \{\{\}\}$, reaches a deadlock after two steps, and is not 1-safe.

An EOS (see [12]) is, intuitively, a *system* PN $\hat{N} = \langle \hat{P}, \hat{T}, \hat{F} \rangle$, whose tokens carry an internal PN taken from a fixed finite set $\mathcal{N}$ of disjoint *object* PNs $N = (P_N, T_N, F_N)$. $\mathcal{N}$ contains the special object $\blacksquare = (\emptyset, \emptyset, \emptyset)$. Each system net place can hold a single type of object net, according to a typing function $d : \hat{P} \longrightarrow \mathcal{N}$. EOSs fire events $\langle \hat{\tau}, \theta \rangle$, where $\hat{\tau} \in \hat{T}$ and θ maps each $N \in \mathcal{N}$ to a multiset of transitions of N itself. We denote by $\theta(N)$ the sum of all $t \in \mathit{supp}(\theta)$ such that $t \in T_N$, counting multiplicities. EOS tokens are nested, i.e., each token at a system place $\hat{p} \in \hat{P}$ carries a PN marking μ for the object net $d(p)$. Such a token is denoted by $\langle \hat{p}, \mu \rangle$. The nested token is graphically represented by connecting, via a dashed line, $\hat{p}$ with a representation of $d(p)$ marked by μ. EOS markings, also called nested markings, are finite multisets of nested tokens. With a slight abuse of notation, we denote markings omitting double curly brackets from multiset notation and, if needed, concatenate tokens using +.

Example 2. Fig. 2 depicts the system net $\hat{N}$ (the idle transitions are omitted) and object net `drone` of an EOS $\mathfrak{E} = \langle \hat{N}, \mathcal{N}, d, \Theta \rangle$ modeling a drone that *(1)* moves between a base and a field, *(2)* has two batteries, *(3)* consumes one charge-unit per battery per movement, and *(4)* charges its batteries by multiples of two charge-units when at base. Technically, $\mathcal{N} = \{\texttt{drone}, \blacksquare\}$ (even if $\blacksquare$ is unused), $d(\texttt{base}) = d(\texttt{field}) = \texttt{drone}$, and Θ synchronizes `takeOff` and `land` (respectively `charge`) in $\hat{N}$ with `move` (`charge1` and `charge2`) in `drone`. Formally, $\Theta = \{\langle \texttt{takeOff}, \{\{\texttt{move}\}\}\rangle, \langle \texttt{land}, \{\{\texttt{move}\}\}\rangle, \langle \texttt{charge}, \{\{\texttt{charge1}\}\}\rangle, \langle \texttt{charge}, \{\{\texttt{charge2}\}\}\rangle\}$. The marking $\mu = \langle \texttt{drone}, \{\{\texttt{batt1}, \texttt{batt1}\}\}\rangle$ represents a single partially charged drone at `base`, with two charge units in the first battery.

Given two markings λ and μ, we say: $\lambda \leq_s \mu$ if μ contains more tokens than λ at the system level; $\lambda \leq_o \mu$ if it is possible to injectively map each nested token $\langle \hat{p}, M \rangle$ of λ to a nested token $\langle \hat{p}, M' \rangle$ of μ

where $M \leq M'$; finally, $\lambda \leq_f \mu$ if there is some λ' such that $\lambda \leq_s \lambda' \leq_o \mu$ or, equivalently $\lambda \leq_o \lambda' \leq_s \mu$. Given a nested marking μ, $\Pi^1(\mu)$ is the PN marking for the system net obtained by retaining only the system net tokens. Given also an object net $N \in \mathcal{N}$, $\Pi^2_N(\mu)$ is the PN marking for the object N obtained by merging all markings carried by tokens at places p of type $d(p) = N$.

When an event $e = \langle \hat{\tau}, \theta \rangle$ fires, $\hat{\tau}$ consumes nested tokens at the system level in the standard way. For each object net, the carried markings of the consumed tokens are merged and the transitions in θ are fired, obtaining PN markings π_N for each $N \in \mathcal{N}$. Nested tokens with empty carried markings are produced at the system level according to the post-conditions of $\hat{\tau}$. Finally, the markings μ_N are non-deterministically distributed on the new nested marking, according to the typing function. This process can happen only if e is enabled on μ under a mode (λ, ρ), i.e., a pair of nested markings. This happens iff $\lambda \leq_s \mu$, $\Pi^1(\lambda) = \mathtt{pre}(\hat{\tau})$, $\Pi^2_N(\lambda) \geq \mathtt{pre}_N(\theta(N))$, $\Pi^1(\rho) = \mathtt{post}(\hat{\tau})$, and $\Pi^2_N(\rho) = \Pi^2_N(\lambda) - \mathtt{pre}_N(\theta(N)) + \mathtt{post}_N(\theta(N))$, for each $N \in \mathcal{N}$.

2.2. Telingo

Telingo specializes the ASP solver Clingo to temporal domains and uses a logic program to specify finite runs. The program can use the scopes: *initial*,*dynamic*,*always*,*final*, which are evaluated at the first, each except first, each, and last step, respectively. Rule bodies can refer to the extension of the previous configuration by prefixing the literals with a prime. Finite-, linear-time formulas can appear in the dedicated atom `&tel` in constraints and behind default negation. For example, the constraint `:- &tel{>?(a>a)}` filters out all runs that eventually reach (`>?` stands for eventually reach) a configuration C with successor C' (`>` stands for next) where `a` is true on both. Telingo can be called setting the option `--imax` to a number of maximal step to be simulated. In the standard configuration, Telingo stops as soon as it finds a finite run satisfying the program or exceeds `--imax`.

3. PNs in Telingo

3.1. PNs in Logic Programs

The standard syntax for PNs is the PNML language [13]. For example, the input PNs provided by the MCC [14] are provided in PNML syntax. The first task we faced was the translation of PNML files into Logic Program (LP) files for Telingo. This required to fix a syntax to specify PNs in LP. Inspired by previous works (see, e.g., [10, 9]), we used the following solutions:

- The number n of places and m of transitions are specified by two constants via the directives `#const numPlaces=n` and `#const numPlaces=m`. The names of the places and transition is abstracted away, i.e., they range in $\{0, \dots, n-1\}$ and $\{0, \dots, m-1\}$ respectively.
- Each pre-condition $F(p,t) = n$ and post-condition $F(t,p) = n$ is explicitly specified only if $n > 0$ using the fact `pre(p,t,n)` and `post(p,t,n)`, respectively. These facts must be available during the whole computation and, thus, are put in the scope of the `#program always` directive.
- The file contains also the specification of the initial marking, which is represented as a function from places to numbers. Specifically, for each place p hosting $n \in \mathbb{N}$ tokens (possibly $n = 0$), we add the fact `mark(p,n)` to the scope of the `#program initial` directive. Finally, we clean the Telingo output using the directive `#show mark/2`.

Example 3. The LP specification of the PN in Ex. 1 is:

```
1 #program always.
2 #const numPlaces=3.
3 #const numTransitions=1.
4 pre(0,0,1).
5 pre(1,0,1).
6 post(2,0,1).
7 #program initial.
8 mark(0,2).
9 mark(1,5).
10 mark(2,0).
11 #show mark/2.
```

Using ANTLR [15], we built an open source tool [16], implemented in C++, to produce LP specifications out of MCC benchmarks.

3.2. PN dynamics

The PN dynamics can be easily implemented in Telingo using the standard guess-and-check methodology of ASP: at each step, we *1)* sample a transition using a choice rule, *2)* check whether it is enabled on the previous marking using a constraint, and *3)* deduce the facts encoding the new marking using a couple of simple rules that take in consideration the sampled transition, its conditions, and the previous marking. This last step is done using the rules

```
mark(P,K-N+M) :- pre(P,T,N), post(P,T,M),
    'mark(P,K), fire(T).
mark(P,K+M) :- not pre(P,T,_), post(P,T,M),
    'mark(P,K), fire(T).
mark(P,K-N) :- pre(P,T,N), not post(P,T,_),
    'mark(P,K), fire(T).
mark(P,K) :- not pre(P,T,_), not post(P,T,_),
    'mark(P,K), fire(T).
```

The lossy dynamics is supported by allowing Telingo to possibly sample, next to the transitions at phase 1, also the `lossy` flag and, consequently, a sub-marking by firing the choice rule

```
{mark(P,1..N)}=1 :- 'mark(P,N),lossy.
```

Assuming that the dynamics is encoded in `dynamic.lp`, the simulation of the runs of maximum length n of a PN encoded in `pn.lp` can be executed as follows:

```
Telingo dynamic.lp pn.lp 0 --imax=n
```

3.3. PN Verification Problems

We considered several problems from the MCC. Since Telingo does not natively support branching time formulas, we focused on linear paths namely reachability/coverability, deadlock detection, and 1-safeness. Since Telingo simulates runs incrementally, it is sufficient to check these properties just at the last step of the finite run. In fact, reachability/coverability and deadlock detection are all eventuality properties. Moreover, also the opposite of 1-safeness, i.e., whether a non-1-safe marking can be reached, is of the same type.

Example 4. To check the reachability in Ex. 1 of the marking $(1, 0, 2)$, we need the rules

```
#program final.
:- not mark(0,1).
:- not mark(1,0).
:- not mark(2,2).
```

Coverability can be similarly be specified using the rules

```
#program final.
:- mark(1,N), N<1.
:- mark(1,N), N<0.
:- mark(2,N), N<2.
```

Deadlock reachability requires to check the enabledness of all transitions and, so, requires their enumeration in a dedicated predicate.

```
#program final.
transition(0..numTransitions-1).
disabled(T) :- transition(T), pre(P,T,N),
    mark(P,M), M<N.
enabled(T) :- not disabled(T), transition(T).
nonDeadlock :- enabled(T).
deadlock :- not nonDeadlock.
:- not deadlock.
```

Finally, (the opposite of) 1-safeness is specified by

```
#program final.
unsafe :- mark(P,N), N>1.
:- not unsafe.
```

By adding these rules, if no maximum number m of steps is signaled, Telingo will return a finite run witnessing the property, if it exists, or will never terminate, otherwise. If m is provided, Telingo will always terminate, but will return a witnessing run of at most m steps.

For all these properties, we can seamlessly restrict the analysis to runs with at most $\ell \in \mathbb{N} \cup \{|\mathbb{N}|\}$ many lossy-steps for any $\ell \in \mathbb{N}$. For example, the rules

problem	lossiness	−imax =5 (s)	−imax =10 (s)	−imax =20 (s)	TAPAAL (s)
deadlock	none	UNSAT in 0.052	SAT in 0.622	SAT in 82.754	SAT in $5e-6$
deadlock	any	SAT in 0.009	SAT in 0.010	SAT in 0.009	NA
1-safeness	none	UNSAT in 0.010	UNSAT in 0.014	UNSAT in 0.027	UNSAT in 0
1-safeness	any	UNSAT in 0.013	UNSAT in 0.018	UNSAT in 0.032	NA

Table 1
Comparative Results with TAPAAL for the Eratosthenes-PT-010 PN from the MCC benchmarks [14].

```
1 #program initial.
```

```
2 :- &tel{>?(lossy >(>? lossy))}.
```

filter out all runs with at least two distinct lossy steps, so as to output only solutions witnessing the existence of suitable runs with at most $\ell = 1$ lossy step. The parameter ℓ can be controlled by nesting the string `>(>? lossy)` appropriately.

3.4. PN experiments

In the running example above, we used a simple PN to illustrate the concepts. However, for our experimentation, we considered several standard benchmarks taken from MCC [14]

which range across various industrial case studies and have different sizes of the PNs. We experimented with the analysis of deadlock reachability and 1-safeness, for various maximal numbers of steps (5, 10, and 20): these properties can be expressed without any expert knowledge on the PN structure. We compared the output of our prototype for all benchmarks with the output provided, on the same instances, by the state-of-the-art tool TAPAAL 3.9.3 [17]. The outputs match exactly; an indication of the correctness of the results, thereby giving our prototype a tool confidence of 100% (despite noncompetitive times). We analyzed the benchmarks for both runs without lossy steps and with arbitrarily many lossy steps. A subset of our results and comparisons (for no loss) on a single PN is reported in Tab. 1.

4. EOSs in Telingo

4.1. EOSs in Logic Programs

As for PNs, we started by fixing a syntax for EOS in LP. The major difference with PNs is the presence of events and of nested tokens. Events $\langle \hat{\tau}, \theta \rangle$ are specified by a name in a predicate `event/1`, a mapping of the name to τ in a predicate `eventSys/2`, and a mapping of the name to each transition in θ with multiplicity in a predicate `eventObj/3`

Example 5. The event `takingOff` $= \langle$`takeOff`$, \{\{$`move`$\}\}\rangle$ in Ex. 2 is specified by the facts *(1)* `event(takingOff)`, *(2)* `eventSys(takingOff, takeOff)`, and *(3)* `eventObj(takingOff1, move, 1)`.

We came up with two encoding of nested markings. The first aims at representing nested tokens directly by linearizing the carried marking in a tuple. This requires to provide an order among the places and the usage of functional symbols as well as external Python functions.

Example 6. 2 nested tokens $\langle$`base`$, \{\{$`2batt1`$\}\}\rangle$ for the EOS in Ex. 2 are specified by the fact `nM`$(2,$ `base`$, (2, 0))$ assuming the order `batt1` $\leq$ `batt2` for the object net `drone`.

The second representation employs a purely relational representation of each token in a relation `tok/3`. It keeps track of a token id, the token place, and the parent token (i.e., the token at the system level carrying it), if it exists (otherwise, a dummy constant `_sys` is put in its place). The choice of identifiers is irrelevant as long as they form a primary key for `tok`.

Example 7. One nested token from Ex. 5 is equivalently captured by the facts *(1)* `tok`$(0,$ `base`$,$ `_sys`$)$, *(2)* `tok`$(1,$ `batt1`$, 0)$, and *(3)* `tok`$(2,$ `batt1`$, 1)$.

4.2. EOS dynamics

When compared to PNs, EOS dynamics has to take care of two main aspects: the choice of enabling modes (λ, ρ) next to events e, and the manipulation of nested markings. While modes range, in principle, over an infinite domain, we can restrict the choice over a finite set, by recalling that λ should be a sub-marking of the current marking. After choosing a λ and checking its compatibility with the enabling predicate for e, the PN markings $\Pi^1(\rho)$ and $\Pi^2_N(\rho)$ are univocally determined, for each object net $N \in \mathcal{N}$. To materialize these projection in a nested marking ρ, we make use of the relational representation of tokens. This allows us to easily assign children tokens to their parents non-deterministically. Afterwards, we convert ρ in the nested token representation and update the marking. This solution requires the call of external Python functions handle the tuples for internal markings. The preliminary simulations on the EOS in Ex. 2 were not optimal. We are currently working on a better implementation.

4.3. EOS verification

We specify EOS properties analogously to Sec. 3.3, but taking into account nested tokens. For example, the reachability of the target $\langle \texttt{base}, \{\{\texttt{batt1}, \texttt{batt1}\}\}\rangle$ is checked by the rules

```
#program final.
:- not nM(1,base,(2,0)).
:- nM(N,base,Tup), Tup != (2,0).
:- nM(N,field,Tup).
```

Its coverability at the system net level is specified by rules 1 and 2 above. Coverability at the object level is specified by the rules

```
#program final.
covered :- nM(1,base,Tup), Tup <= (2,0).
:- not covered.
```

Full coverability is checked by

```
#program final.
covered :- nM(N,base,Tup), Tup <= (2,0).
:- not covered.
```

Even in this case, the specification of the amount of lossiness is orthogonal to that of the verification property and is performed using the same constraints as shown in Sec. 3.3.

5. Conclusions

We explored the applicability of Telingo to the simulation and analysis of lossy PNs and EOSs. Our approach is similar to [10, 9], but we additionally encoded lossiness, EOSs, and provided a translator from PNML to LP syntax. We also conducted preliminary tests on the verification of deadlock reachability and 1-safeness of lossy PNs on runs with zero or arbitrarily many lossy steps. When compared with state-of-the-art tools like TAPAAL, we obtained sound results, yet with optimizable performances. On the one hand, the specification of lossy runs was especially elegant and flexible in Telingo, e.g., when compared to SMT-based PN verifiers [18]. Our approach addresses PNs in general and is applicable to PNs with reset, transfer, and inhibitory arcs. On the other hand, TAPAAL does not natively support the analysis of properties under lossiness, unless somehow encoded in the PN itself. As a byproduct of our experiments, we provide a standalone utility [16] compatible with Linux to translate standard PNML files to the PN LP syntax. We hope that this translator will provide ASP practitioners with a convenient tool to approach PN benchmarks. The prototype for EOS needs further development before tests can be meaningfully conducted. This task is challenging, since PN reachability is non-elementary and several reachability problems for lossy EOSs are undecidable in general [19]. However, to the best of our knowledge, there are no available tools dedicated to lossy EOSs.

References

[1] C. Petri, Communication with Automata, AD-630, RADC, 1966.

[2] C. Dufourd, et al., Reset nets between decidability and undecidability, in: ICALP, volume 1443 of *LNCS*, Springer, 1998, pp. 103–115.
[3] R. Valk, Object Petri nets: Using the nets-within-nets paradigm, in: APN, volume 3098 of *LNCS*, 2003, pp. 819–848.
[4] M. Köhler-Bussmeier, L. Capra, Robustness: A natural definition based on nets-within-nets, in: PNSE, 2023.
[5] F. Huch, Verification of Erlang programs using abstract interpretation and model checking, Ph.D. thesis, RWTH Aachen, 2001.
[6] A. Bouajjani, R. Mayr, Model checking lossy vector addition systems, in: STACS, '99.
[7] P. Cabalar, Temporal ASP: from logical foundations to practical use with telingo, in: Reasoning Web. Declarative Artificial Intelligence, volume 13100 of *LNCS*, 2021, pp. 94–114.
[8] M. Gebser, et al., Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (2019) 27–82.
[9] S. Anwar, et al., Encoding higher level extensions of Petri nets in answer set programming, in: LPNMR, volume 8148 of *LNCS*, 2013, pp. 116–121.
[10] Y. Dimopoulos, et al., Encoding reversing Petri nets in answer set programming, in: RC, volume 12227 of *LNCS*, 2020, pp. 264–271.
[11] T. Murata, Petri nets: Properties, analysis and applications, IEEE 77 (1989) 541–580.
[12] M. Köhler-Bußmeier, A survey of decidability results for elementary object systems, Fundamenta Informaticae 130 (2014) 99–123.
[13] L. Hillah, et al., Extending pnml scope: A framework to combine Petri nets types, TOPNOC 6 (2012) 46–70.
[14] F. Kordon, et al., Complete Results for the Model Checking Contest, https://mcc.lip6.fr/2023/results.php, 2023.
[15] T. J. Parr, R. W. Quong, ANTLR: A predicated- *LL(k)* parser generator, Softw. Pract. Exp. 25 (1995) 789–810.
[16] T. Prince, F. Di Cosmo, Nets within nets Telingo analyser, 2024. URL: https://doi.org/10.5281/zenodo.11401876.
[17] J. F. Jensen, et al., TAPAAL and reachability analysis of P/T nets, TOPNOC 11 (2016) 307–318.
[18] T. Prince, DCModelChecker 2.0: A BMC tool for Unbounded PN, 2022. URL: https://doi.org/10.5281/zenodo.7352391.
[19] F. Di Cosmo, et al., Deciding reachability and coverability in lossy EOS, in: PNSE, 2024. To appear.