Declarative Modelling and Analysis of Reversing Petri Nets

Yannis Dimopoulos[1], Eleftheria Kouppari[1] and Anna Philippou[1]

[1]Department of Computer Science, University of Cyprus

Abstract

Reversing Petri Nets (RPNs) extend classical Petri nets with the ability to undo the firing of transitions, capturing systems where actions may be reversed under well-defined semantic conditions. This supports modelling of biochemical interactions, fault recovery, and debugging scenarios where forward-only execution is insufficient. In this paper we present an ASP framework for the simulation and reachability analysis of RPNs, supporting three execution modes: forward, causal-order, and out-of-causal-order reversibility. We evaluate the three modes over a suite of randomly generated RPN instances with configurable place count, token count, and bond density, and analyse the computational impact of each execution mode across varying structural configurations.

Keywords

Reversing Petri Nets, Answer Set Programming, Reversibility, Reachability Analysis, Benchmark Generation

1. Introduction

Reversible computation extends classical models of computation with the ability to undo previously executed actions. It has recently been receiving increasing attention, especially in the context of concurrent computation, where several distinct notions have been identified and studied. *Backtracking* reverses actions in strict reverse order, retracing the execution path step by step. While this is natural in a sequential setting, in a concurrent setting it introduces artificial causal dependencies: independent actions that were executed in some order must be undone in the exact reverse order, even though no genuine causal relationship exists between them. *Causal* reversibility relaxes this constraint: an action may be undone once all actions that causally depend on it have been reversed, allowing independent actions to be undone in any order while still respecting the causal dependencies. *Out-of-causal*-order reversibility goes further still, allowing any executed action to be reversed regardless of causal dependencies. The last notion is particularly powerful: by reversing actions in an out-of-causal order, a system can reach states that are inaccessible by any forward-only execution, a property that arises naturally in biochemical signalling pathways, fault recovery mechanisms, and long-running transactional systems.

Reversing Petri Nets (RPNs) [1, 2] are a Petri-net framework designed to support all three notions of reversibility. RPNs extend classical Petri nets with *bonds*, which connect tokens and allow them to move together, and an *execution history*, which records the transitions that have fired. These two mechanisms together enable structured, history-aware reversal of transitions without requiring any modification to the net structure itself. RPNs have been applied to the modelling of biochemical systems [2] and wireless communications [3], and formal translations into Coloured Petri Nets have been proposed [4, 5] to study the relationship between the two models and leverage existing verification infrastructure.

Answer Set Programming (ASP) [6, 7] provides a natural setting for encoding RPN execution. Its declarative rule language allows transition rules, causal constraints, and reversal conditions to be expressed uniformly, with semantic variants introduced modularly. CLINGO, a state-of-the-art ASP solver [8], supports a multi-shot solving architecture that allows a grounded instance to be reused across multiple verification queries and enables incremental horizon extension [9]. ASP-based encodings of Petri nets have been proposed for simulation and reasoning about biological pathways [10, 11], and Telingo [12] has been applied to bounded verification of Petri nets using temporal equilibrium logic.

PNSE'26, International Workshop on Petri Nets and Software Engineering, 2026

yannis@ucy.ac.cy (Y. Dimopoulos); ekoupp02@ucy.ac.cy (E. Kouppari); annap@ucy.ac.cy (A. Philippou)

0000-0001-9583-9754 (Y. Dimopoulos); 0009-0006-3196-9289 (E. Kouppari); 0000-0002-1665-9913 (A. Philippou)

None of these works address reversible execution. ASP is well-suited to this setting: choice rules and negation-as-failure provide a direct encoding of non-deterministic transition firing and enabledness, aggregates over the execution history express causal and out-of-causal reversal conditions concisely, and CLINGO's multi-shot solving supports incremental horizon extension for shortest-path analysis.

An ASP encoding of RPNs supporting forward and causal execution was presented in [13]. In this work, we extend this encoding to also support out-of-causal-order reversal, enabling forward, causal, and out-of-causal-order execution within a single framework. Furthermore, we extend the short paper of [13] with an extensive evaluation of the approach for all three modes of execution. Specifically, this paper makes the following contributions:

ASP encoding. We extend the encoding of [13] to support out-of-causal-order reversal, completing the executable treatment of all three RPN execution modes, yielding the first executable implementation of out-of-causal order reversal for RPNs.

Benchmark generator. We introduce an RPN benchmark generator based on weighted stepwise structural refinement [14]. The generator produces structurally diverse instances with configurable place count, token count, and bond density, enabling systematic and reproducible evaluation of RPN analysis methods.

Empirical evaluation. We evaluate all three execution modes over a benchmark suite of 48,000 randomly generated instances, ranging from 10 to 100 places and 2 to 24 tokens, yielding 144,000 solver runs in total - one per instance per execution mode. By analysing each instance under identical structural configurations across all three modes, we evaluate and compare their performance. Our results confirm that the ASP-based framework provides a viable and tractable approach for the analysis of RPNs across all three execution modes.

Case study. We apply the framework to a simplified model of the ERK signalling pathway [15], demonstrating that out-of-causal reversal is necessary to reach the target configuration and illustrating shortest-path analysis within the encoding.

The remainder of this paper is organised as follows. Section 2 presents background on RPNs and ASP. Section 3 describes the ASP encoding of RPNs focusing on the encoding of the out-of-causal-order reversibility semantics, while Section 4 introduces the benchmark generator. Section 5 presents the experimental evaluation and Section 6 concludes the paper.

2. Background

2.1. Reversing Petri Nets

In this subsection, we recall the definition and semantics of RPNs from [1, 2]. We focus on causal and out-of-causal reversibility, as these capture the richer concurrent behaviours RPNs are designed to model. RPNs extend classical Petri nets with two mechanisms: *bonds*, which dynamically connect tokens and allow them to move together, and an *execution history*, which records which transitions have fired and in which order. The state of an RPN is therefore determined not only by the token distribution but also by the bond structure and the history of transitions. Formally, an RPN is defined as follows.

Definition 1. A *Reversing Petri Net* (RPN) is a tuple (A, P, B, T, F) where:

1. A is a finite set of *tokens*,
2. P is a finite set of *places*,
3. $B \subseteq \{(a, b) \mid a, b \in A, a \neq b\}$ is a set of undirected *bonds*,
4. T is a finite set of *transitions*,
5. $F : (P \times T \cup T \times P) \to 2^{A \cup B}$ defines directed *arcs* where $(a, b) \in F(x, y)$ implies $a, b \in F(x, y)$.

Intuitively, bonds connect specific tokens and allow them to behave as a single structure. An example of an RPN can be seen in the top-left net of Figure 1, where transition t_1 creates a bond between tokens a and b, transition t_2 creates a bond between c and d, and subsequently t_3 creates bond b–c, joining all four tokens into a single connected structure.

To ensure tokens and bonds are conserved across transition firings, preventing cloning, destruction, or splitting of connected components, we require RPNs to be well-formed. Given a transition $t \in T$, we write ${}^\bullet t$ for its preset (input places) and $t^\bullet$ for its postset (output places). We write $pre(t) = \bigcup_{p \in P} F(p, t)$ and $post(t) = \bigcup_{p \in P} F(t, p)$ for the unions of all labels on incoming and outgoing arcs of t, respectively.

Definition 2. A Reversing Petri Net (A, P, B, T, F) is *well-formed* if it satisfies the following conditions for all $t \in T$:

1. $A \cap {}^\bullet t = A \cap t^\bullet$,
2. if $a - b \in {}^\bullet t$ then $a - b \in t^\bullet$, and
3. $F(t, x) \cap F(t, y) = \emptyset$ for all $x, y \in P$, $x \neq y$.

Condition 1 ensures token conservation, condition 2 prevents bond destruction, and condition 3 prevents token cloning across output places. Together these conditions guarantee that every token and bond is preserved across transition firings: nothing is lost, destroyed, or duplicated beyond what the transition explicitly specifies. This property supports the reversible behaviour of RPNs, as it ensures that the effects of any transition are precisely specified, thus allowing their reversal.

A *marking* is a distribution of tokens and bonds across places, $M : P \to 2^{A \cup B}$, where $a{-}b \in M(x)$ implies $a, b \in M(x)$. A *history* assigns a memory to each transition, $H : T \to 2^{\mathbb{N}}$, where $H(t) = \emptyset$ indicates t has not executed, and $H(t) = \{k_1, \ldots, k_n\}$ that t has fired n times in the forward direction, with the k_i's recording the execution order of the firings. Note that a transition may execute multiple times due to the presence of cycles in the net's structure. A *state* is a pair $\langle M, H \rangle$.

Given $a \in A$ and $C \subseteq A \cup B$, $conn(a, C)$ denotes the connected component of a in C: the tokens connected to a via sequences of bonds, together with those bonds [2].

$$conn(a, C) = (\{a\} \cap C) \cup \{x \mid \exists\, w \text{ s.t. } path(a, w, C),\ (b, c) \in w,\ x \in \{(b, c), b, c\}\}$$

where $path(a, w, C)$ holds if $w = \beta_1, \ldots, \beta_n$ with $\beta_i = (a_{i-1}, a_i) \in C \cap B$, $a_i \in C \cap A$, and $a_0 = a$.

In Figure 1, after the execution of transitions t_1, t_2, and t_3, the bonds $a{-}b$, $c{-}d$, and $b{-}c$ together imply that tokens a, b, c, and d belong to the same connected component: $conn(a, y) = \{a, b, c, d, a{-}b, b{-}c, c{-}d\}$. Consequently, any transition that requires or produces token a must treat the entire component as a single structure, rather than a in isolation.

Forward Execution.

Definition 3. Consider an RPN (A, P, B, T, F) and a state $\langle M, H \rangle$. Transition $t \in T$ is *forward-enabled* in $\langle M, H \rangle$ if:

1. if $a \in F(x, t)$ then $a \in M(x)$,
2. if $\beta \in F(x, t)$ then $\beta \in M(x)$,
3. if $a \in F(t, y_1)$, $b \in F(t, y_2)$, $y_1 \neq y_2$, then $b \notin conn(a, M(x))$ for all $x \in {}^\bullet t$,
4. if $\beta \in F(t, x)$ and $\beta \in M(y)$ for some $y \in {}^\bullet t$, then $\beta \in F(y, t)$.

Conditions 1–2 require all tokens and bonds specified in incoming arcs to be present in the respective input places. Condition 3 prevents connected tokens from being split across distinct output places. Condition 4 ensures bonds produced by the transition are absent from the input marking unless explicitly required. The *effect* of t is the set of bonds created by firing it: $effect(t) = t^\bullet - {}^\bullet t$. When t fires, the resulting state $\langle M', H' \rangle$ is defined as follows:

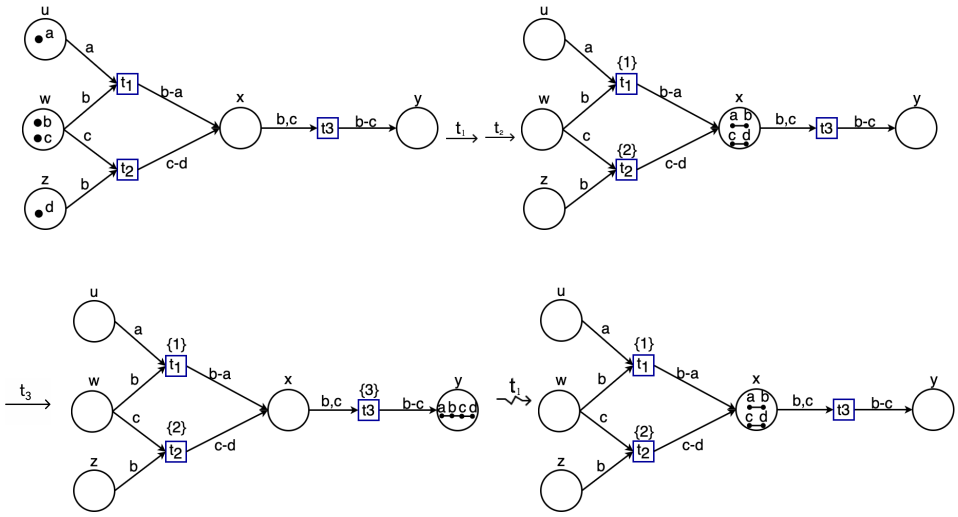


Figure 1: Causal reversibility. Starting from the initial marking (top left), transitions t_1 and t_2 fire independently (top right), followed by t_3 (bottom left). At this point, only t_3 is eligible for reversal since t_1 and t_2 causally precede it. Reversing t_3 (bottom right) returns the system to the state after execution of t_1 and t_2, from which both t_1 and t_2 become independently eligible for reversal and may be reversed in any order.

Definition 4. Given an RPN (A, P, B, T, F), a state $\langle M, H\rangle$, and a transition t forward-enabled in $\langle M, H\rangle$, we write $\langle M, H\rangle \xrightarrow{t} \langle M', H'\rangle$ where:

$$M'(x) = \begin{cases} M(x) - \bigcup_{a \in F(x,t)} conn(a, M(x)) & \text{if } x \in {}^\bullet t \\ M(x) \cup F(t,x) \cup \bigcup_{y \in P, a \in F(t,x) \cap F(y,t)} conn(a, M(y)) & \text{if } x \in t^\bullet \\ M(x) & \text{otherwise} \end{cases}$$

and

$$H'(t') = \begin{cases} H(t') \cup \{\max(\{0\} \cup \{k \mid k \in H(t''), t'' \in T\}) + 1\} & \text{if } t' = t \\ H(t') & \text{otherwise} \end{cases}$$

Firing t removes from each input place the connected components of its required tokens, and adds to each output place the specified tokens, bonds, and induced connected components. The history H is updated by assigning to t a fresh execution index greater than any previously recorded. An example of forward execution is illustrated in Figure 1 (first three transitions).

Causal-order Reversibility. Under causal-order reversibility, a transition t may be reversed only if all transitions that causally depend on it have been undone. Causal dependencies are induced by token flow: if a transition produces tokens later consumed by another, the latter causally depends on the former. The formal treatment of causal-order reversibility is presented in [2] and its ASP encoding in [13]. In this work we focus on the extension to out-of-causal reversal. Figure 1 illustrates a complete execution sequence demonstrating causal reversibility: after firing t_1, t_2, and t_3, only t_3 is eligible for reversal since t_1 and t_2 causally precede it; once t_3 is reversed, both become independently eligible.

It is worth noting that under causal-order reversibility, the set of reachable states coincides with the set of states reachable by forward-only execution [2]. This stands in contrast to out-of-causal-order reversibility, which, as we discuss next, strictly extends the reachable state space.

Out-of-causal-order Reversibility. Out-of-causal-order reversal permits any executed transition to be reversed regardless of causal dependencies. A transition t is eligible for reversal as long as it has been executed more times than it has been reversed:

Definition 5. Consider an RPN (A, P, B, T, F), a state $\langle M, H\rangle$, and a transition $t \in T$. We say that t is *o-enabled* in $\langle M, H\rangle$ if $H(t) \neq \emptyset$.

When t is reversed, all bonds introduced by its execution are broken. If this disconnects a connected component into smaller subcomponents, each subcomponent must be redistributed to its most recent location prior to t's execution. Following [2], two auxiliary functions track the history of token manipulation:

Definition 6. Given an RPN (A, P, B, T, F), an initial marking M_0, a history H, and a set $C \subseteq A \cup B$ of tokens and bonds:

$$last_T(C, H) = \begin{cases} t & \text{if } \exists\, t,\ post(t) \cap C \neq \emptyset,\ H(t) \neq \emptyset,\ \nexists\, t',\ post(t') \cap C \neq \emptyset, \\ & H(t') \neq \emptyset,\ \max(H(t')) \geq \max(H(t)) \\ \bot & \text{otherwise} \end{cases}$$

$$last_P(C, H) = \begin{cases} x & \text{if } t = last_T(C, H),\ \{x\} = \{y \in t^\bullet \mid F(t, y) \cap C \neq \emptyset\} \\ x & \text{if } \bot = last_T(C, H),\ C \subseteq M_0(x) \\ \bot & \text{otherwise} \end{cases}$$

Thus, $last_T(C, H)$ identifies the most recently executed transition whose effects overlap C, or $\bot$ if no such transition exists. $last_P(C, H)$ returns the output place of that transition for C, or the place in the initial marking where C resided if $last_T(C, H) = \bot$. Using these functions, out-of-causal-order reversal is defined as follows:

Definition 7. Given an RPN (A, P, B, T, F), a state $\langle M, H \rangle$, and a transition t that is o-enabled in $\langle M, H \rangle$, we write $\langle M, H \rangle \xrightarrow{t}_o \langle M', H' \rangle$ where:

$$M'(x) \quad = \quad \Big(M(x) \cup \bigcup_{(a,y) \in I_{t,x}} C_{a,y}\Big) - \Big(\mathit{effect}(t) \cup \bigcup_{a \in O_{t,x}} C_{a,x}\Big)$$

where $I_{t,x} = \{(a, y) \mid y \in P,\ a \in A \cap M(y) \cap t^\bullet,\ last_P(C_{a,y}, H') = x\}$, $O_{t,x} = \{a \mid a \in A \cap M(x) \cap t^\bullet,\ last_P(C_{a,x}, H') \neq x\}$, and $C_{b,z} = conn(b, M(z) - \mathit{effect}(t))$.

The catalysis example of Figure 2 illustrates the key features of out-of-causal-order reversibility. Starting from the initial marking, t_1 fires and creates bond $c{-}a$, relocating the connected component to a new place. Subsequently t_2 fires, creating bond $a{-}b$ and forming the component $c{-}a{-}b$. Now suppose t_1 is reversed out of causal order - that is, before t_2 has been reversed. The reversal destroys only the bond $c{-}a$ introduced by t_1, while the bond $a{-}b$ created by t_2 remains intact. The component $c{-}a{-}b$ is therefore split: token c is returned to its most recent prior location determined by function $last_P$, while tokens a and b remain bonded at their current place. The resulting state with token c disconnected from bonded pair $a{-}b$ is unreachable by any forward-only execution, illustrating the expressive power of out-of-causal-order reversibility.

2.2. Answer Set Programming

ASP is a declarative logic programming paradigm for combinatorial problem solving [6, 7]. An ASP program consists of rules of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$$

where A_0 is the *head* and the right-hand side is the *body*. The operator *not* denotes *negation-as-failure*: atom A_i is assumed false if it cannot be derived from the program. Intuitively, a rule states that if all body atoms $A_1, \ldots, A_m$ hold and none of $A_{m+1}, \ldots, A_n$ can be derived, then A_0 must hold. The semantics are given by *stable models*, minimal sets of atoms satisfying all rules; each stable model corresponds to a solution of the encoded problem. Modern solvers such as `clingo` extend the language with *choice rules* $\{A\} \leftarrow B$, which allow A to be freely included or excluded whenever B holds, introducing controlled non-determinism that is essential for encoding transition firing and reversal.

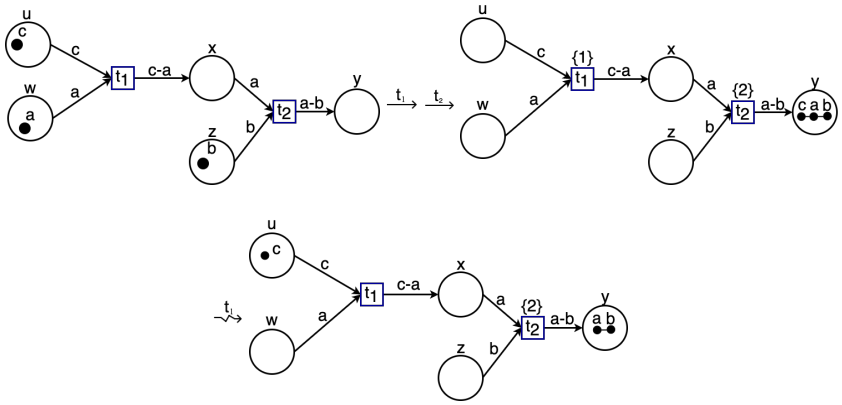


Figure 2: Out-of-causal-order reversal in a catalysis interaction. After executing t_1 and t_2, reversing t_1 breaks only the bond it introduced while preserving the bond created by t_2.

Evaluation proceeds in two phases. During *grounding*, first-order rules, that is, rules containing variables ranging over the constants of the program, are instantiated by substituting all possible combinations of constants for variables, producing an equivalent propositional program with no variables. During *solving*, the stable models of this grounded program are computed. The size of the grounded program often dominates overall runtime, particularly when rules range over large domains such as places, transitions, tokens, and time steps.

3. ASP Encoding

We encode Reversing Petri Nets in Answer Set Programming by representing markings, bonds, and execution history as logical predicates, and transitions as rules that describe how these evolve over time steps. The encoding follows the operational semantics of RPNs: forward execution creates tokens and bonds, while reversal removes only those bonds introduced by the corresponding transition, preserving bonds created by later transitions, as illustrated in Figure 2. The encoding builds upon a previous work-in-progress version [13], in which the forward and causal-order execution modes were introduced; these modes are included here refined for improved performance, and we additionally propose an extension to the out-of-causal execution mode.

The encoding considers an RPN over a bounded horizon of h time steps, where each time step represents a discrete system state. Predicates are indexed by time, enabling the explicit representation of how tokens, bonds, and transition histories evolve across successive steps.

The basic predicates that represent the input network are `trans(T)`, `token(Q)`, and `place(P)`, that correspond to the transitions, tokens and places, respectively, whereas predicates `ptarc(P,T,Q)` declare that `Q` $\in F($`P,T`$)$ (similarly for `tparc(T,P,Q)`) and `ptarcb(P,T,Q1,Q2)` refer to bonds, i.e. `Q1-Q2` $\in F($`P,T`$)$. The markings of an RPN are captured by predicates `holds` for tokens and `holdsbonds` for bonds. That is, `holdsbonds(P,Q1,Q2,TS)` expresses that bond $Q_1 \breve{} Q_2$ holds at place P at time step TS. Finally, `transHistory(T,H,TS)` records that the history value of transition T at time step TS is H.

In what follows, we present representative rules of the encoding; the complete encoding is available at `[https://github.com/ekoupp02-ucy/reversing-petri-nets-asp.git]`.

3.1. Well-Formedness

The well-formedness conditions of Definition 2 are expressed in ASP as follows.

Condition 1: Token conservation. A transition must neither erase nor create tokens.

```
notwellformed :- ptarc(_,T,Q),
     not tparc(T,_,Q),
     not tparcb(T,_,Q,_),
     not tparcb(T,_,_,Q).
notwellformed :- tparc(T,_,Q),
      not ptarc(_,T,Q).
```

A violation occurs when a token Q appears in an incoming arc of transition T but not in any outgoing arc, either as a single token (`tparc`) or as part of a bond (`tparcb`). Here _ denotes an anonymous variable, used when the value of a variable is irrelevant and does not need to be named. The second rule captures the symmetric case: a token appears in an outgoing arc but has no corresponding incoming arc.

Condition 2: Bond preservation. A transition must not destroy bonds.

```
notwellformed :- ptarcb(_,T,Q1,Q2),
    not tparcb(T,_,Q1,Q2).
```

A violation occurs when a bond between tokens Q_1 and Q_2 appears in an incoming arc of transition T but does not appear in any outgoing arc.

Condition 3: No token cloning. A token must not be sent to more than one output place by the same transition.
A violation occurs when token Q is assigned to two distinct output places P_1 and P_2 of the same transition T. The three rules cover all combinations: token to token, token to the first element of a bond, and token to the second element of a bond.

```
1 notwellformed :- tparc(T,P1,Q),
       tparc(T,P2,Q), P1!=P2.
2 notwellformed :- tparc(T,P1,Q),
     tparcb(T,P2,Q,_), P1!=P2.
3 notwellformed :- tparc(T,P1,Q),
     tparcb(T,P2,_,Q), P1!=P2.
```

Wellformed: Finally, an RPN instance is accepted only if none of the above violations are derived. If any violation is derived, `notwellformed` holds, and consequently `wellformed` cannot be derived. The integrity constraint `:- not wellformed` in line 2 then deadlocks the solution, meaning no stable model exists for the instance and it is discarded entirely.

```
1 wellformed :- not notwellformed.
2 :- not wellformed.
```

3.2. Forward Execution

We first present forward execution, which captures the standard token flow extended with bonds.

Listing 1: Representative rules for forward execution

```
1 notenabled(T,TS) :- tokenInArc(P,T,Q),
2                     not holds(P,Q,TS), not holdsbonds(P,Q,_,TS), time(TS).
3 notenabled(T,TS) :- ptarcb(P,T,Q1,Q2), not holdsbonds(P,Q1,Q2,TS), time(TS).
4 enabled(T,TS)    :- trans(T), time(TS), not notenabled(T,TS).
5 {fires(T,TS)}    :- enabled(T,TS).
6 holds(P,Q,TS+1)  :- holds(P,Q,TS), not del(P,Q,TS).
7 holdsbonds(P,Q1,Q2,TS+1) :- holdsbonds(P,Q1,Q2,TS),
8                             not delBond(P,Q1,Q2,TS), not delBond(P,Q2,Q1,TS).
```

Forward enabledness is encoded via negation-as-failure: a transition is enabled at time step TS whenever no disabling condition can be derived (lines 1-4). The choice rule for `fires(T,TS)` (line 5) permits the nondeterministic execution of any enabled transition. When a transition fires, tokens on outgoing arcs are added and tokens on incoming arcs are removed at the next time step, with symmetric rules for bonds. The inertia rules (lines 6–8) ensure that tokens and bonds persist across time steps unless explicitly deleted; note that bond deletion checks both orderings of the bond endpoints, since bonds are undirected.

3.3. Causal-order Reversal

Under causal reversibility, a transition t may be reversed only once all causally dependent transitions have been undone. Causal dependency is tracked via the execution history: transition t_2 causally

depends on t_1 if t_1 produces a token or bond later consumed by t_2. Before presenting the causal reversal rules, we introduce two auxiliary predicates derived from the static net structure.

```
1 tokenInArc(PT,T,Q)  :- ptarc(PT,T,Q).
2 tokenInArc(PT,T,Q)  :- ptarcb(PT,T,Q,_).
3 tokenInArc(PT,T,Q)  :- ptarcb(PT,T,_,Q).
4 tokenOutArc(T,TP,Q) :- tparc(T,TP,Q).
5 tokenOutArc(T,TP,Q) :- tparcb(T,TP,Q,_).
6 tokenOutArc(T,TP,Q) :- tparcb(T,TP,_,Q).
```

`tokenInArc(P,T,Q)` holds when token Q flows into transition T from place P, either as a single token or as part of a bond. Symmetrically, `tokenOutArc(T,P,Q)` holds when token Q flows out of transition T to place P. These predicates abstract over whether the token travels alone or as part of a bond.

The causal-order reversibility encoding tracks dependencies between transitions through the execution history and ensures that a transition may only be reversed once all transitions that causally depend on it have been undone. Representative rules are shown in Listing 2. In particular, `dependent(T1,T2,TS)` holds when T_2 has consumed a token produced by T_1 (lines 1–2), or a token connected to one produced by T_1 through bonds at time step TS (lines 3–4), capturing a direct or bond-mediated causal dependency between the two transitions. `dependentNotReversed(T1,TS)` holds when T_1 has a dependent transition T_2 that has not yet been reversed: this is determined by comparing the maximum history keys of T_1 and T_2, where a greater key for T_2 indicates it was executed after T_1 and has not been undone (lines 5–7). A transition is causally enabled for reversal when it has been executed, that is, its history is greater than zero, and no transition that depends on it remains unreversed (lines 8–9). The choice rule for `reversesC` nondeterministically selects an eligible transition for causal reversal (line 10), and `breakBond` removes exactly the bonds introduced by the reversed transition (line 11). Marking updates follow the same add/delete scheme as forward execution (Listing 1).

Listing 2: Representative rules for causal reversal

```
1  dependent(T1,T2,TS) :- H1>0, H2>0,  tokenOutArc(T1,_,Q), tokenInArc(_,T2,Q),
2          transHistory(T1,H1,TS), transHistory(T2,H2,TS).
3  dependent(T1,T2,TS) :- transHistory(T1,H1,TS), H1>0,  transHistory(T2,H2,TS),
4      H2>0, tokenOutArc(T1,_,Q), tokenInArc(_,T2,Q1), connected(_,Q,Q1,TS).
5  dependentNotReversed(T1,TS) :- dependent(T1,T2,TS),
6      H2 = #max{H : transHistory(T2,H,TS), history(H)},
7      H1 = #max{H : transHistory(T1,H,TS), history(H)}, T1 != T2, H2 > H1.
8  enabledC(T,TS) :- trans(T), transHistory(T,H,TS), H>0,
9      not dependentNotReversed(T,TS).
10 {reversesC(T,TS)} :- enabledC(T,TS).
11 breakBond(P,Q1,Q2,TS) :- reversesC(T,TS), tparcb(T,P,Q1,Q2).
```

3.4. Out-of-causal-order Reversal

We now introduce the out-of-causal reversal mode, which constitutes the main extension of the encoding in this work. Unlike causal reversal, which requires all dependent transitions to be undone before a transition can be reversed, the out-of-causal reversal ignores such dependencies. As specified in Listing 3, a transition is enabled for out-of-causal reversal whenever it has been executed (line 1). The choice rule for `reversesOC` nondeterministically selects an eligible transition for reversal (line 2).

The key mechanism for implementing the semantics of out-of-causal-order reversal, is selective bond removal. `breakBond` removes exactly those bonds introduced by the reversed transition (lines 3–5); note that line 3 ensures symmetry since bonds are undirected. `stillConnected` then recomputes the remaining connectivity between tokens after bond removal: a direct bond that is not broken remains connected (line 6), and connectivity is propagated transitively through surviving bonds (line 7).

Once the affected structure has been identified, `lastTrans` determines for each token the most recent transition that manipulated it (lines 8–12). The first rule covers tokens manipulated by a common transition; the second covers tokens directly involved in the reversal whose last manipulation was the reversed transition itself. Tokens that no longer remain connected are removed from their current places and restored to their most recent prior location, with broken bonds deleted and surviving bonds

preserved, following the same add/delete scheme as forward execution (Listing 1). This corresponds to the behaviour illustrated in Figure 2, where reversing t_1 removes the bond $c{-}a$ while preserving the bond $a{-}b$, resulting in a partial decomposition of the connected component.

Listing 3: Representative rules for out-of-causal reversal

```
enabledOC(T,TS) :- transHistory(T,H,TS), H > 0, time(TS).
{reversesOC(T,TS)} :- enabledOC(T,TS), trans(T).
breakBond(P,Q1,Q2,TS) :- breakBond(P,Q2,Q1,TS).
breakBond(P,Q1,Q2,TS) :- holdsbonds(P,Q1,Q2,TS), reversesOC(T,TS),
    tparcb(T,_,Q1,Q2).
stillConnected(P,Q1,Q2,TS) :- holdsbonds(P,Q1,Q2,TS), not breakBond(P,Q1,Q2,TS),
     time(TS).
stillConnected(P,Q1,Q2,TS) :- holdsbonds(P,Q1,Q3,TS), not breakBond(P,Q1,Q3,TS),
     stillConnected(P,Q3,Q2,TS), Q1!=Q2, Q2!=Q3, Q1!=Q3, time(TS).
lastTrans(T,Q,TS) :-  H1 = #max{H : usesCommon(T1,Q,TS), transHistory(T1,H,TS)},
    transHistory(T,H1,TS), H1 > 0, token(Q), time(TS).
lastTrans(T,Q,TS) :- reversalUsesToken(T,Q,TS), transHistory(T,H,TS), H > 0,
    H > #max{HH : usesCommon(T1,Q,TS), transHistory(T1,HH,TS) ; 0 : token(Q)},
    tokenOutArc(T,_,Q), token(Q), time(TS).
```

The three encodings provide a complete and uniform treatment of all RPN execution modes within a single ASP framework. All three share a common base of 34 forward execution rules covering enabledness, firing, and inertia for tokens and bonds, and 23 general rules for token, place, and transition derivation. The causal and out-of-causal encodings build on top of this shared base: the causal reversal encoding adds 31 rules for dependency tracking via `dependent` and `dependentNotReversed`, while the out-of-causal encoding adds 35 rules for selective bond removal, connectivity recomputation via `stillConnected`, and token relocation via `lastTrans`, rules that range over the bond structure and execution history, as reflected in the increased computational cost observed in our empirical evaluation.

3.5. Behavioural Properties

Reachability. A reachability query can ask whether a (partial) marking can arise during the execution of a system. For instance, asking whether bond $a{-}b$ can be produced at place y is expressed as an integrity constraint over a time step $\texttt{maxT} \leq h$:

```
:- not holdsbonds(y,a,b,maxT).
```

If no stable model satisfies this constraint, the target marking is unreachable within the given horizon.

Several additional classical Petri net properties can be encoded in a similar declarative style:

Shortest path. The minimum number of steps to reach a target state can be found using CLINGO's incremental mode [9], which partitions the program into a `base` program of static facts, a `step(t)` program evaluated at each increment, and a `check(t)` goal condition. The solver extends the horizon by one step at a time until the goal is satisfied:

```
#program check(t).
goal(t)      :- holdsbonds(y,a,b,t).
#external query(t).
:- query(t), t <= maxT, not goal(t).
:- query(t), t > maxT.
```

The solver incrementally extends the horizon until `goal(t)` is satisfied, returning the minimum number of steps required to produce the goal, or unsatisfiable if the target is unreachable within `maxT` steps.

Reachability and shortest-path analysis are used in our experimental evaluation (Section 5) and ERK case study (Section 5.3). Liveness, persistence, and home state are presented here to illustrate the extensibility of the framework; their systematic evaluation is left as future work.

Liveness. A transition t is *dead* if it is never enabled in any reachable state. Dead transitions are identified by:

```
notdead(T) :- enabled(T,TS), time(TS).
dead(T)    :- trans(T), not notdead(T).
```

A transition is *L1-live* if it is enabled at least once, encoded as `l1(T) :- enabled(T,TS)`. *L2-live* requires enablement at k distinct time steps, encoded by requiring k distinct time step values at which the transition is enabled.

Persistence. Two transitions are *persistent* if they are causally independent and do not compete for the same token. The encoding first derives `useTheSame(T1,T2)` when two transitions require the same token from the same place, and `causally(T1,T2)` when T_2 depends on the output of T_1. Two transitions are then persistent if neither condition holds:

```
persistent(T1,T2) :- trans(T1), trans(T2), not useTheSame(T1,T2),
    not persistent(T2,T1), not causally(T1,T2), not causally(T2,T1).
```

Persistence depends only on net structure and is independent of execution mode.

4. Random RPN Generator

4.1. Refinement-Based Generation

To evaluate our ASP framework over structurally diverse instances, we developed a benchmark generator for RPNs. The generator builds on the weighted stepwise refinement methodology of [14], in which nets are constructed incrementally by randomly applying structural refinement rules. These rules define local transformations that incrementally extend the net, producing structurally diverse instances rather than arbitrary random graphs.

Starting from a minimal single-place seed, the generator repeatedly applies nine construction rules until the desired place count, transition count are reached. The nine rules are divided into two classes. They are illustrated in Figure 3, with blue indicating the element where the rule is applied and green indicating newly-added elements. *Refinement rules* (R1–R6) expand the net while preserving structural properties [14]: R1 (place refinement) splits an arc by inserting a new place and transition; R2 (transition refinement) splits an arc by inserting a new transition and place; R3 (arc refinement) inserts an intermediate node on an existing arc; R4 (place duplication) duplicates a place with identical pre- and post-sets, introducing parallelism; R5 (transition duplication) duplicates a transition, enabling non-deterministic choice; and R6 (loop addition) inserts a self-loop transition on a place, introducing cyclic behaviour. *Bridge rules* (R7–R9) connect existing nodes, introducing complex dependencies and generally not preserving structural properties: R7 (place bridge) inserts a place between two existing transitions; R8 (transition bridge) inserts a transition between two existing places; and R9 (arc bridge) adds a direct arc from an existing node to another, creating shortcuts.

Rule selection is probabilistic and governed by configurable weights; different subsets induce distinct structural classes. The four rule sets used in our evaluation span a range of structural complexity: R1–R3 produces acyclic, free-choice nets; R1–R5 adds duplication and parallelism; R1–R6 introduces cycles via loop addition; and R1–R9 additionally introduces arbitrary connectivity through bridge rules.

4.2. Arc Labelling and Bond Construction

Once the net topology is fixed, arcs are labelled with tokens and bond-labelled arcs are introduced.

Token assignment. Each arc is labelled with a token drawn from a configurable set. For each transition, tokens are first assigned to its output arcs, then matched to its input arcs to ensure token conservation (Condition 1 of Definition 2). The assignment uses a balancing strategy based on token availability: tokens already present at an input place from a previous transition's output are preferred, distributing tokens across the net and avoiding configurations where all tokens accumulate at a single place. Each token appears on at most one transition-to-place arc per transition, preventing a single transition from cloning a token across multiple output places.

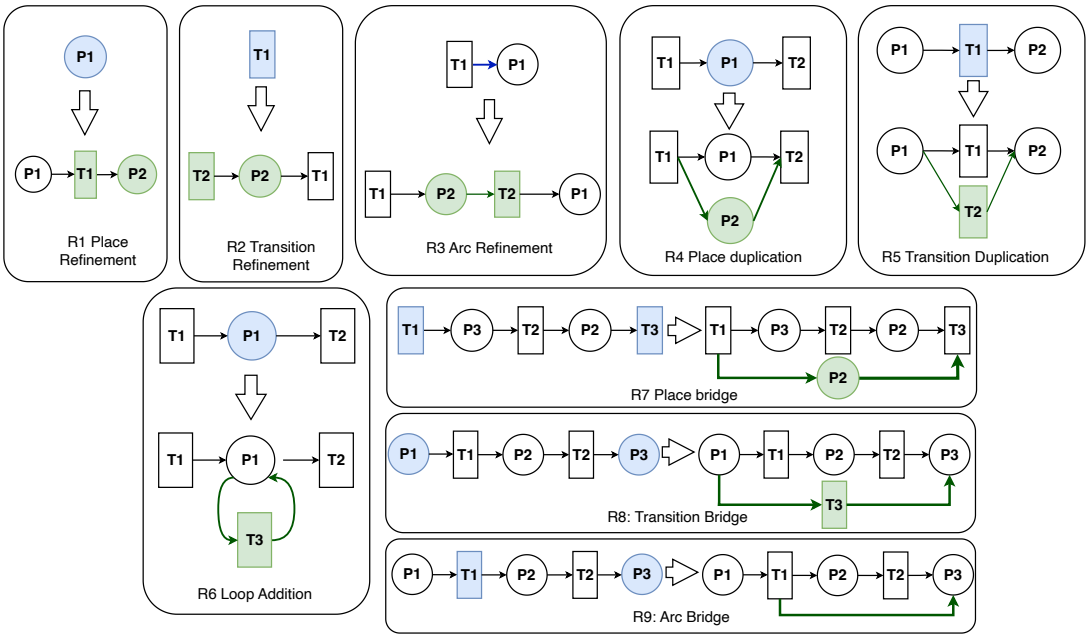


Figure 3: Refinement rules (R1–R6) and bridge rules (R7–R9) used in the benchmark generator. Blue indicates the element where the rule is applied; green indicates newly-added elements.

Bond construction. Bond-labelled arcs are introduced after token assignment under a configurable bond budget. This is the novel contribution of the generator: extending the plain net structure with bonds while preserving well-formedness. Two issues must be carefully handled. The first issue is *bond conservation*: well-formedness requires that if a bond appears on an incoming arc of a transition, it must also appear on an outgoing arc (Condition 2). To address this, bond introduction is performed via two operations. *Consume-and-preserve* selects two tokens already flowing through a transition and converts their individual arcs into a bond arc (`ptarcb`) on the input side, with a matching bond arc (`tparcb`) on the output side, preserving the bond across the transition. *Bond creation* introduces a new bond on an output arc between two tokens already flowing through the transition, without requiring a bonded input. The second issue is *token uniqueness*: since each token may appear in at most one place at any time, bonds may only be created between tokens assigned to the same output place of a transition; bonding tokens destined for different output places would require the connected component to reside in two places simultaneously, violating Condition 3 of Definition 2.

4.3. Initial Marking and Validation

After arc labelling, an initial marking is computed by placing each token at its earliest required input place. Bonds that cannot be created by any transition are pre-initialised; all others arise dynamically through transition behaviour. Each generated instance is validated against the well-formedness conditions of Definition 2 using the ASP well-formedness encoding of Section 3.1; instances violating any condition are discarded.

5. Experimental Evaluation

5.1. Setup

We evaluate our ASP framework on a suite of randomly generated RPN instances. Instances are parameterised by place count $(10, 20, \ldots, 100)$, number of tokens $(2, 4, \ldots, 24)$, rule set (R1–R3, R1–R5, R1–R6, R1–R9), and bond type (bonds, no_bonds), with 50 instances per configuration, yielding $10 \times 12 \times 4 \times 2 \times 50 = 48{,}000$ benchmark instances. Each instance is evaluated under all three execution

Table 1
Execution time statistics (SAT instances only, in seconds).

Mode	Mean	Median	Max	Std
Forward	0.14	0.13	1.27	0.08
Causal	10.92	3.73	451.0	24.95
NonCausal	26.57	14.10	322.0	34.66

modes, giving $144{,}000$ solver runs in total. For each generated RPN instance, three reachability queries are constructed—one for each of forward-only, forward and causal reversal, and forward and out-of-causal reversal execution—to define target configurations within a bounded horizon. A reachability file consists of integrity constraints over the final time step, requiring specific tokens and, where relevant, bonds to hold at designated places using predicates such as `holds` and `holdsbonds`.

Reachability targets are constructed through a stepwise process: the ASP encoding is executed on each instance under the corresponding execution mode, advancing one time step at a time over 10 steps, and the resulting marking at time step 11 is extracted as the target. The `holds` and `holdsbonds` atoms at that step are converted into integrity constraints that define the reachability goal. During trace construction, immediate reversals of transitions are avoided to prevent trivial back-and-forth behaviour, encouraging longer and more structurally informative traces. Reachability targets are therefore mode-specific, as the set of reachable states differs across execution modes.

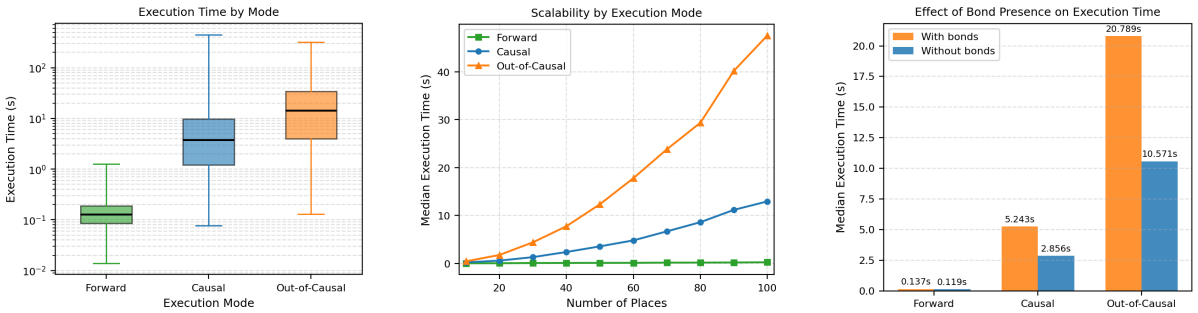
Before presenting the results, we note that under causal reversibility the set of reachable states coincides with those reachable by forward execution alone [2]. This raises the question of whether causal reversal is necessary for reachability analysis. Our interest in including it lies in characterising its computational cost relative to the other modes, and in laying the groundwork for settings where this equivalence no longer holds, such as when transitions are selectively irreversible, or in the context of broader analyses such as model checking.

All experiments were conducted using `clingo` 5.8.0 on a Linux HPC cluster with 8 cores and 64 GB RAM per node. Each run is subject to a time limit of 1,800 seconds and a horizon of $h = 10$ steps.

5.2. Results

Performance hierarchy. Table 1 summarises execution time across the three modes, and Figure 4(a) shows the corresponding distributions. Forward execution is by far the fastest, with a mean of 0.14 s and a maximum of 1.27 s, confirming that forward reachability in RPNs is computationally lightweight. Causal-order reversibility introduces significant overhead, with a mean of 10.9 s and a maximum of 451 s. Out-of-causal-order reversibility is the most expensive mode, with a mean of 26.6 s and a maximum of 322 s. At first glance, one might expect out-of-causal execution to be cheaper than causal, since it does not enforce causal dependency checks before allowing a transition to reverse. However, the increased cost arises not from enabledness checking but from the structural complexity of the reversal itself. Under out-of-causal reversal, every executed transition may be reversed at any time, requiring the encoding to track the full bond structure and compute token relocation via `stillConnected` and `lastTrans` at every time step. These predicates range over the entire execution history and bond connectivity graph, and the encoding must additionally compute the relocation of each disconnected subcomponent to its correct destination place, significantly increasing the size of the grounded program. Causal reversal, by contrast, restricts reversals to transitions with no outstanding dependencies, reducing the number of candidate reversals at each step and consequently the size of the search space.

Scalability. Figure 4(b) shows median execution time as a function of place count. Forward execution remains essentially flat across all configurations (median < 0.25 s at 100 places), confirming that the forward encoding scales well with net size. This is expected: the forward rules range only over places, transitions, and tokens at each time step, and the grounded program grows linearly with net size. Causal-order and out-of-causal-order reversibility both grow with place count, reaching medians of



(a) Execution time distribution by mode (log scale)

(b) Median execution time vs. place count

(c) Effect of bond presence on median execution time

Figure 4: Experimental results across execution modes.

13.0 s and 47.6 s respectively at 100 places, with out-of-causal consistently more expensive across all configurations. The growth under causal reversal is driven by the dependency tracking rules, which must consider all pairs of executed transitions at each time step — a cost that grows with the number of transitions and therefore with net size. Under out-of-causal reversal, the additional cost of `stillConnected` and `lastTrans` compounds this growth, as these predicates range over the full bond connectivity graph whose size also increases with the net. Despite this growth, all three modes remain tractable across the full range of 10 to 100 places, suggesting that the ASP encoding scales acceptably for these net sizes.

Effect of bonds. Figure 4(c) shows the effect of bond presence on median execution time. Bonds roughly double execution time for causal (2.9 s vs 5.2 s, 83% increase) and out-of-causal (10.6 s vs 20.8 s, 96% increase) modes. Forward execution is largely unaffected (0.12 s vs 0.14 s). This asymmetry is expected. Under forward execution, bonds are simply additional predicates that persist via inertia — they do not affect the enabledness computation in any structurally complex way. For causal reversal, bonds introduce additional causal dependencies between transitions: a transition that creates a bond causally precedes any transition that consumes it, expanding the dependency graph and increasing the cost of `dependent` and `dependentNotReversed`. For out-of-causal reversal, the effect is more pronounced: bonds directly drive the size of the `stillConnected` and `lastTrans` computations, since these predicates must track connectivity across the full bond graph. The more bonds present, the larger and more dense this graph becomes, increasing both grounding size and solving time.

A finer-grained analysis over bond-enabled instances reveals that bond count is a strong predictor of out-of-causal execution time ($r = 0.61$), with median time increasing steadily as bond count grows (Figure 5, right). The non-monotonic behaviour at low bond counts is an artefact of larger nets being overrepresented in those groups; within fixed net sizes, the trend is consistently increasing. This confirms that the cost of `stillConnected` and `lastTrans` grows directly with the size of the bond connectivity graph.

Effect of rule set. Figure 5 shows median execution time by rule set for causal and out-of-causal execution. All rule sets achieve 100% solve rates; however, execution time varies significantly across structural classes. The R1–R3 rule set, which produces acyclic free-choice nets, is consistently the fastest across both modes. This is expected: acyclic nets have no cycles in the execution history, limiting the depth of dependency chains under causal reversal and the number of candidate reversals under out-of-causal reversal. Adding duplication rules R4–R5 introduces parallelism, which modestly increases cost by expanding the set of concurrently enabled transitions. Adding loop rule R6 introduces the greatest overhead, particularly under out-of-causal execution: cycles allow transitions to fire multiple times, growing the execution history and significantly increasing the number of entries that `lastTrans` and `stillConnected` must consider. The R1–R9 rule set, which additionally includes bridge rules

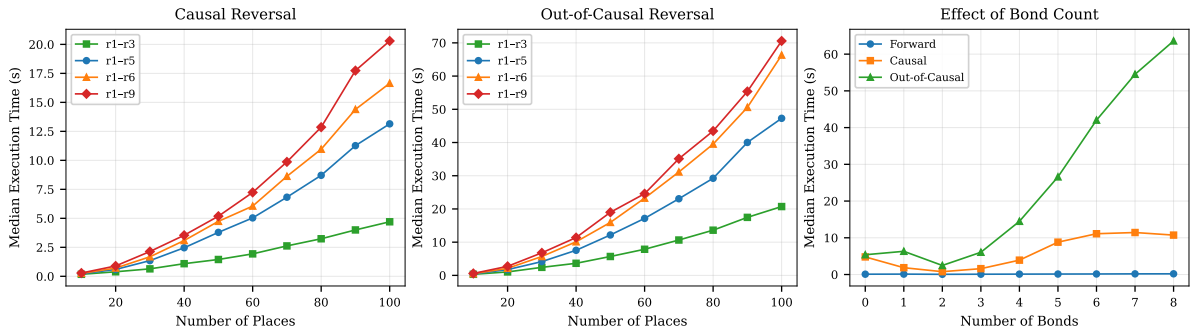


Figure 5: Median execution time by rule set for causal (left) and out-of-causal (centre) execution, and effect of bond count on execution time (right).

introducing arbitrary connectivity, is slightly more expensive than R1–R6 under out-of-causal execution. This suggests that both cyclicity and connectivity contribute to execution cost: loop rules create cyclic execution paths that compound history tracking costs, while bridge rules add further arc density that increases the size of the grounded program.

5.3. The ERK Case Study

To illustrate the application of our approach, we consider a simplified model of the ERK signalling pathway (Figure 6), originally introduced in [15]. The ERK (extracellular signal-regulated kinase) pathway is a central signalling cascade that transmits signals from cell surface receptors to the nucleus, regulating cell proliferation, differentiation, and survival. Mutations in this pathway are implicated in many cancers, making it a key target for therapeutic intervention. In the RPN model [16], tokens represent molecular species (e.g., r for Ras, f for Raf, m for MEK, p for phosphate, e for ERK) and bonds represent biochemical interactions such as phosphorylation. The model requires inhibitor conditions specifying transitions enabled only when specific tokens are absent. In Figure 6, such conditions are depicted as tokens with a bar $\bar{\cdot}$; for example, on the arc from place FMP to transition c, $\bar{\text{f}}$ indicates that c is enabled only when token f is absent from FMP. These absence conditions are encoded as:

```
notenabled(T,TS) :- ptarcabsence(P,T,Q), holds(P,Q,TS), time(TS).
notenabled(T,TS) :- ptarcbabsence(P,T,Q1,Q2), holdsbonds(P,Q1,Q2,TS), time(TS).
```

To apply the framework to this model, the user provides the net structure, the initial marking, and a reachability query. A representative sample of the input is shown below:

```
ptarc(rp,a1,r).
ptarc(fp,a1,f).
tparcb(a1,rfp,r,f).
holds(rp,r,0).  holds(fp,f,0).
holds(mp,m,0).  holds(pp,p,0).
holds(ep,e,0).
ptarc(fmpp,c,m).
ptarcabsence(fmpp,c,f).
ptarc(ep,c,e).
```

Lines 1–3 encode the arc structure of transition a_1: it consumes tokens r and f from places `rp` and `fp`, and produces bond $r-f$ at place `rfp`. Lines 4–6 define the initial marking, placing each molecular species at its designated place. Lines 7–9 encode transition c: it requires token m from `fmpp` and token e from `ep`, but also requires token f to be *absent* from `fmpp` via `ptarcabsence`, capturing the biochemical condition that Raf must have been released before ERK activation can proceed.

A reachability query `:- not holdsbonds(prep,r,p,maxT).` asks whether bond $r-p$ can be produced at place `prep`. The solver returns a witness trace within approximately two seconds:

Listing 4: Witness trace for the ERK query.

```
fires(a2t,0) fires(p1t,1) reversesOC(a2t,2) fires(a1t,3) fires(ct,4)
reversesOC(p1t,5) fires(p2t,6) fires(bt,7) reversesOC(a1t,8) reversesOC(p2t,9)
fires(p3t,10)
```

The trace illustrates the necessity of out-of-causal reversal. The transition names in the trace correspond to the transitions in Figure 6: `a2t` is a_2 (Raf binds MEK), `p1t` is p_1 (phosphorylation of

MEK), a1t is a_1 (Raf binds RKIP), ct is c (ERK activation), p2t is p_2 (phosphorylation of ERK), bt is b (signal passing to nucleus), and p3t is p_3 (phosphorylation of RKIP). Transition p3t requires token f to be absent from place fremp, but f is introduced by a1t at step 3. Consequently, a1t must be reversed at step 8 before p3t can fire, as the reversal is driven by an absence condition rather than causal dependency.

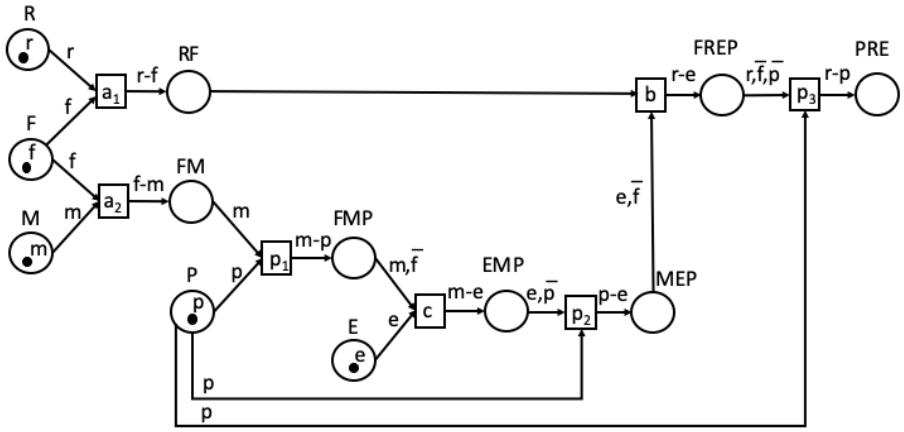


Figure 6: The ERK signalling pathway encoded as an RPN

Applying the shortest path to the ERK reachability query, the encoding confirms that the shortest path to bond $r - p$ at prep under out-of-causal execution is exactly 11 steps, with 8 distinct shortest paths of this length, resolved in 0.61 seconds.

6. Conclusions and Future Work

We have presented an ASP framework for Reversing Petri Nets supporting forward execution, causal-order reversibility, and out-of-causal-order reversibility within a uniform declarative encoding. The framework provides the first executable implementation of out-of-causal reversibility and enables bounded reachability analysis without structural translation to alternative net variants. A random RPN generator based on weighted stepwise refinement supports systematic structural variation across diverse net classes. A systematic evaluation over 48,000 benchmark instances confirms the feasibility of the approach across all three execution modes, with forward and causal-order execution fully tractable across all tested configurations (10–100 places) at median execution times of 0.14 s and 3.73 s respectively, and out-of-causal-order execution remaining tractable despite higher cost (median 14.10 s).

The declarative structure of the encoding naturally supports behavioural properties beyond reachability: liveness, persistence, home-state detection, and shortest-path analysis can be expressed by adding constraints to the base encoding without modifying the core rules. Shortest-path analysis is demonstrated on the ERK case study; a systematic evaluation of the remaining properties under the three execution modes is left as future work. The framework is currently restricted to single-token RPNs where only a single token of each type exists in a net. Extending the approach to multi-token RPNs [16, 17], removing this restriction, is architecturally natural but expected to increase grounding size significantly. Further directions include recovery analysis, where reversibility determines whether a system can return to a safe state after reaching an undesirable configuration, and adapting the encoding to Telingo's temporal language [12] for more concise expression of temporal properties.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] A. Philippou, K. Psara, Reversible computation in Petri nets, in: Proceedings of RC 2018, volume 11106 of *LNCS*, Springer, 2018, pp. 84–101. doi:10.1007/978-3-319-99498-7_6.

[2] A. Philippou, K. Psara, Reversible computation in nets with bonds, Journal of Logical and Algebraic Methods in Programming 124 (2022) 100718. doi:10.1016/j.jlamp.2021.100718.

[3] H. Siljak, K. Psara, A. Philippou, Distributed antenna selection for massive MIMO using reversing Petri nets, IEEE Wireless Commununication Letters 8 (2019) 1427–1430. doi:10.1109/LWC.2019.2920128.

[4] K. Barylska, A. Gogolinska, L. Mikulski, A. Philippou, M. Piatkowski, K. Psara, Reversing computations modelled by coloured Petri nets, in: Proceedings of ATAED 2018, volume 2115 of *CEUR Workshop Proceedings*, 2018, pp. 91–111.

[5] K. Barylska, A. Gogolinska, L. Mikulski, A. Philippou, M. Piatkowski, K. Psara, Formal translation from reversing Petri nets to coloured Petri nets, in: Proceedings of RC 2022, volume 13354 of *LNCS*, Springer, 2022, pp. 172–186. doi:10.1007/978-3-031-09005-9_12.

[6] Y. Dimopoulos, B. Nebel, J. Koehler, Encoding planning problems in nonmonotonic logic programs, in: Recent Advances in AI Planning, volume 1348 of *LNCS*, Springer, 1997, pp. 169–181. doi:10.1007/3-540-63912-8_11.

[7] E. T. Mueller, Commonsense reasoning using answer set programming, in: Commonsense Reasoning, 2nd ed., Morgan Kaufmann, 2015, pp. 249–269. doi:10.1016/B978-0-12-801416-5.00015-2.

[8] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, P. Wanko, Potassco user guide, 2019. Available at https://github.com/potassco/guide/releases/.

[9] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[10] S. Anwar, C. Baral, K. Inoue, Encoding Petri nets in answer set programming for simulation based reasoning, Theory and Practice of Logic Programming 13 (2013).

[11] S. Anwar, C. Baral, K. Inoue, Encoding higher level extensions of Petri nets in answer set programming, in: Logic Programming and Nonmonotonic Reasoning (LPNMR), volume 8148 of *LNCS*, Springer, 2013, pp. 116–121. doi:10.1007/978-3-642-40564-8_12.

[12] F. Di Cosmo, T. Prince, Bounded verification of Petri nets and EOSs using Telingo: an experience report, in: Proceedings of CILC 2024, volume 3733 of *CEUR Workshop Proceedings*, 2024. URL: https://ceur-ws.org/Vol-3733/short6.pdf.

[13] Y. Dimopoulos, E. Kouppari, A. Philippou, K. Psara, Encoding reversing Petri nets in answer set programming, in: Reversible Computation, volume 12227 of *LNCS*, Springer, 2020, pp. 264–271. doi:10.1007/978-3-030-52482-1_17.

[14] K. M. van Hee, Z. Liu, Generating benchmarks by random stepwise refinement of Petri nets, in: Proceedings of the Workshops of PETRI NETS 2010 and ACSD 2010, volume 827 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2010, pp. 403–417.

[15] I. Phillips, I. Ulidowski, S. Yuen, A reversible process calculus and the modelling of the ERK signalling pathway, in: Proceedings of RC 2012, volume 7581 of *LNCS*, Springer, 2012, pp. 218–232. doi:10.1007/978-3-642-36315-3_18.

[16] A. Philippou, K. Psara, A collective interpretation semantics for reversing Petri nets, Theoretical Computer Science 924 (2022) 148–170. doi:10.1016/j.tcs.2022.05.016.

[17] A. Philippou, K. Psara, Token multiplicity in reversing Petri nets under the individual token interpretation, in: Proceedings of EXPRESS/SOS 2022, EPTCS, 2022, pp. 131–150. doi:10.4204/EPTCS.368.8.