A Graph Rewriting-Based Semantics and Implementation for $\rho\pi$

Julie Cailler[0000−0002−6665−8089] and Martin Vassor[0000−0002−2057−0495]

Université de Lorraine, CNRS, Inria, LORIA, 54000, Nancy, France
first.last@univ-lorraine.fr

Abstract. Reversibility in the higher-order process calculus $\rho\pi$ relies on tracking causal dependencies between processes and their (causal) ancestors. This bookkeeping introduces extra complexity to the calculus both syntactically—through the use of tags—and semantically, via additional structural congruence rules to manage them. Although this additional complexity is not a problem for the familiar reader, it creates a steep learning curve for newcomers. To soften this learning curve, we present a graph rewriting-based semantics for $\rho\pi$. We also present an implementation of this graph-based approach with a step-by-step interface, allowing one to familiarize quickly with concepts and intuitions of $\rho\pi$ in a graphical and user-friendly way.

Keywords: Reversible Process Calculus · Graph Rewriting · Causal Consistent Reversibility.

1 Introduction

Motivation. The reversible higher-order π calculus $\rho\pi$ [18,19] is a simple, yet expressive, causally-consistent reversible process calculus. Its simplicity lies in its semantics, which consists in only two rules: one for forward computation that performs a communication, and one for backward computation that undoes one of the *last* communications[1]. This minimal semantic core served as the foundation for numerous variants proposed in recent years [17,16,29].

The counterpart of this simplicity comes with the complexity of the structural congruence rules and the heavy syntax. Processes are annotated with *keys*, which act as references to track causal dependencies. Upon communication, the sender and the receiver are frozen in a memory, and the continuation is created with a fresh key, which is included in the memory. However, due to name restriction and parallel composition, the structural congruence has to accommodate splitting keys and moving key binders.

In this paper, we introduce an alternative semantics for $\rho\pi$ based on graph rewriting [9], as well as a $\rho\pi$ interpreter using this graph rewriting-based semantics. Our goal, with this semantics and this tool, is to provide a more visual approach to this reversible process calculus, helping readers unfamiliar with the domain as well as students to develop intuitions of what happens under the hood.

On the calculus side, using graphs to track causal dependencies allows us to drop keys (which are replaced with edges), which in turn allows us to drop key binders and the need for configurations.

[1] Due to concurrency, we consider the order of events with respect to causal order (see Lamport's *happen-before* relation [15]), which is a partial order; not with respect to temporal order. As such, there might be multiple *last* events at once.

Memories still exist, but are shown as vertices with outgoing edges. On the interpreter side, we provide a Python implementation of our semantics. It supports both *interactive* executions—where the user explicitly selects a graph node and a rule—and an automatic mode.

Contributions. Our contributions are (i) a new semantics for $\rho\pi$, based on graph rewriting instead of structured operational semantics rules; and (ii) G$\rho\pi$F[2], a publicly available implementation of graph rewriting-based $\rho\pi$ in Python [5].

Outline. In Section 2, we informally introduce the $\rho\pi$ calculus and graph rewriting, the formalism used to specify the semantics of $\rho\pi$. The graph structures employed to encode $\rho\pi$, together with the encoding of rewrite rules, are described in Section 3. In this paper, we do not aim to prove the correctness of our encoding, but we nonetheless state the conjecture that one would need to prove to formally establish the correspondence between $\rho\pi$ and the graph-based approach. We then present the interpreter in Section 4. Finally, in Section 5, we discuss the benefits of graphical approaches, showing how improvements of $\rho\pi$ can easily be expressed in terms of graph rewriting, and we discuss related works.

2 Preliminaries

2.1 The $\rho\pi$ Calculus

In this section, we briefly introduce $\rho\pi$, a reversible higher-order process calculus. In the rest of this paper, we are mostly interested in (i) the syntax of processes; (ii) the recording of memories; and (iii) the tracking of causal dependencies.

The syntax of $\rho\pi$ is given in Figure 1. Processes (noted P, Q...) are processes of the asynchronous higher-order π calculus. In $\rho\pi$, processes do not execute directly. Instead, they are embedded in *configurations*, which contain as well memories of past actions and track causal dependencies between different parts of a term, thanks to tags.

$$
\begin{array}{lll}
\mathcal{P},\mathcal{Q} & ::= 0 \mid X \mid \nu c.\,(\mathcal{P}) \mid \mathcal{P} \parallel \mathcal{Q} \mid c\,\langle \mathcal{P} \rangle \mid c(X) \triangleright (\mathcal{P}) & \textit{Processes} \\
\mathcal{M},\mathcal{N} & ::= 0 \mid \nu u.\,(\mathcal{M}) \mid \mathcal{M} \parallel \mathcal{N} \mid \kappa : P \mid [\mu; k] & \textit{Configurations} \\
\kappa & ::= k \mid \langle h, \tilde{h} \rangle \cdot k & \textit{Tags} \\
\mu & ::= \kappa_1 : c\,\langle \mathcal{P} \rangle \parallel \kappa_2 : a(X) \triangleright (\mathcal{Q}) & \textit{Memory content}
\end{array}
$$

Fig. 1: Syntax of $\rho\pi$ ($c \in \mathbb{N}$: channel names; $k \in \mathbb{K}$: keys; $X \in \mathbb{V}_p$: process variables; $u \in \mathbb{N} \cup \mathbb{K}$: identifiers).

The two communication primitives of $\rho\pi$ are those of **HO**π: either a message P is sent on a channel c: $c\,\langle P \rangle$; or a message is received on that channel (called a *trigger*): $c(X) \triangleright (Q)$. A trigger binds variable X in Q. Upon reception of P, X is substituted by P in Q. Channel names used for communication can be restricted to some processes only: $\nu c.\,(P)$ binds c in P, thus preventing communications on that name between P and its surrounding environment. Other forms of processes include parallel composition ($P \parallel Q$); terminated process (0); and process variables, to be substituted upon communication (X).

[2] Which stands for *Graph-based $\rho\pi$ Framework*.

In order to record memories and to track causal dependencies, processes are embedded into *configurations*: upon communication, e.g. $c\langle P\rangle \parallel c(X) \triangleright (Q)$ not only the communication happens, but a memory is created to record previous processes. The communication above reduces to $Q\{^P/_X\} \parallel [c\langle P\rangle \parallel c(X) \triangleright (Q)]$ where $Q\{^P/_X\}$ denotes the process Q where free occurrences of variable X are substituted with P[3]. The memory is the part of the resulting configuration denoted with squared brackets. In order to keep track of the causal dependency between the memory and the continuation, $\rho\pi$ uses *tags*: for instance, upon communication, a fresh tag k is generated, which annotates both the memory and the continuation:

$$\nu k.\,(k :: Q\{^P/_X\} \parallel [c\langle P\rangle \parallel c(X) \triangleright (Q); k])$$

Notice the head restriction, which works as channel name restriction, to ensure tag freshness[4].

Formally, the forward semantics of $\rho\pi$ consists of the least evaluation-closed relation on terms that includes the following rule[5,6]:

$$\text{(R.Fw)}\ \frac{}{\kappa_1 :: P \parallel \kappa_2 :: Q \twoheadrightarrow \nu k.\,(k :: Q\{^P/_X\} \parallel [\kappa_1 :: P \parallel \kappa_2 :: Q; k])}$$

Conversely, reversible behavior consists of removing a live process (i.e. not contained in a memory), and restoring the memory that created that process. Tags are used to find which memory to restore:

$$\text{(R.Bw)}\ \frac{}{\nu k.\,(k :: Q\{^P/_X\} \parallel [\kappa_1 :: P \parallel \kappa_2 :: Q; k]) \rightsquigarrow \kappa_1 :: P \parallel \kappa_2 :: Q}$$

As shown above, the continuation $P' = Q\{^P/_X\}$ of a communication is tagged with a fresh k. When P' is a parallel composition $k :: P' = k :: L \parallel R$, we have to find a way to tag L and R independently. To do so, $\rho\pi$'s structural congruence allows splitting tags: e.g. $\langle h_1, \{h_1, h_2\}\rangle \cdot k$ is a split tag, which indicates that tag k has been split into two tags (h_1 and h_2), and the shown tag is the first of those two tags. Thus, structural congruence allows splitting $k :: L \parallel R$ as $\langle h_1, \{h_1, h_2\}\rangle \cdot k :: L \parallel \langle h_2, \{h_1, h_2\}\rangle \cdot k :: R$.

Splitting tags allows treating both sides of a parallel composition independently, while keeping track of causal links: a single memory can causally precede multiple processes, which have been split around.

Example 1. Consider the following configuration:

$$\nu a.\,(k :: a\langle P\rangle \parallel l :: a(X) \triangleright (X \parallel Q))$$

In this process a sender emits a message P on channel a and a receiver waits for a message X on the same channel before running $X \parallel Q$. Thus a communication can occur, resulting in $P \parallel Q\{^P/_X\}$. When that communication occurs, a memory is created in order to store the state of the sender and receiver. A new tag, m, is created to record the causal dependence between the memory and continuation:

$$\nu a.\,(\nu m.\,(m :: P \parallel Q\{^P/_X\} \parallel [k :: a\langle P\rangle \parallel l :: a(X) \triangleright (X \parallel Q); m]))$$

[3] In the following, we also use this notation for substituting channel names.

[4] Formally, processes before the communication are also tagged, and thus also processes occurring in the memory. Tags not shown for the sake of simplicity.

[5] See [18] for more details on structural congruence and evaluation-closed contexts.

[6] We prefix the names of the reduction rules with R, to distinguish them from (Fw) and (Bw) below.

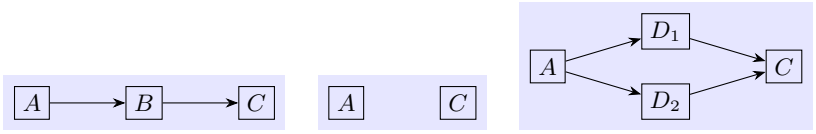
2.2 Graph Rewriting

Using graphs to represent dynamic systems has a long history, dating at least from the early 1970s [11], with applications in a wide range of domains [7,8]. In this paper, a graph $G = (V, E)$ consists of a finite set V of vertices and a set $E \subseteq V \times V$ of edges. Intuitively, graph rewriting operates by applying a set of *rewriting rules*, each of which specifies how to transform a graph L (the left-hand side) into a graph R (the right-hand side). These rules are applied to graphs G, for example transforming G in Figure 3a into H in Figure 3c. Formally, applying a rule $L \longrightarrow R$ to G involves finding a subgraph of G that matches L and replacing it with R to obtain a new graph H. Different approaches exist for implementing this substitution, which leads to different semantics of graph rewriting. In this work, we adopt the *Double Pushout* (DPO) approach [9,10,3].

The main difference with other approaches lies in how L is matched in G. The idea behind the DPO approach is to distinguish a subset of nodes and edges that should appear both in L and R. These nodes form the *interface* with the surrounding graph: they are the only nodes and edges allowed to maintain connections to the rest of G during the application of the rule (as explained below and illustrated in Example 2). In DPO, a rule is a triplet $\langle L, K, R\rangle$ where the graph K is the interface mentioned above. Applying the rule to a graph G proceeds in three steps: (i) find a mapping m from nodes and edges of L to G; (ii) remove nodes and edges of L not in K to obtain an intermediate graph $D = G \setminus m(L \setminus K)$; and (iii) add nodes and edges of R not in K to produce the final graph $H = D \cup (R \setminus K)$. In short, $L \setminus K$ identifies elements to be removed, while $R \setminus K$ identifies elements to be added.

The important point, w.r.t. other approaches, is that the intermediate graph D *has to be a graph*, without dangling edges. In particular, this means that nodes to be removed *must not have* other edges than the one specified in the rule.

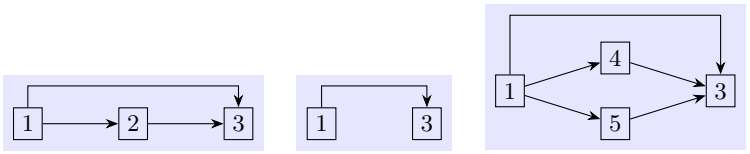
Example 2 (Double pushout rule). Consider the rule $r = \langle L, K, R\rangle$ in Figure 2. We can apply this rule to the graph G, as shown in Figure 3. First, we have to find a candidate mapping: consider $m = \{A \mapsto 1, B \mapsto 2, C \mapsto 3\}$ (for the sake of clarity, we do not detail the mapping of edges as it is trivial). In the rule, A and C belong to K, so they are not removed. B does not appear in K, so it is removed: we obtain graph $D = G \setminus m(L \setminus K)$. Then, in R, two nodes D_1 and D_2 do not exist in K and have to be created, which gives us graph H.



(a) Graph L (b) Graph K (c) Graph R

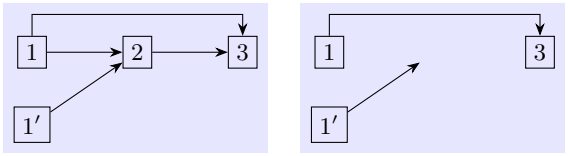
Fig. 2: A graph rewriting rule consisting of a triplet $\langle L, K, R\rangle$ of graphs.

Conversely, consider the rule in Figure 2 and G' shown in Figure 4. If we try to apply the rule with the same mapping, then D, obtained after removing one node and two edges, would be as shown in Figure 4b. Notice the dangling edge. D being ill-formed, the rule does not apply. Other approaches, such as *single pushout* [21], allow the existence of temporary dangling edges, as illustrated in this example. In our case, we will exploit the fact that all edges must be explicitly shown, which can prevent rule application in certain situations.



(a) Graph G (b) Graph D (c) Graph H

Fig. 3: Application of the rule in Figure 2 to a graph G.



(a) Graph G' (b) H', which is not a graph

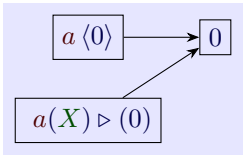
Fig. 4: Trying to apply rule r (Figure 2) when it creates a dangling edge.

3 Encoding $\rho\pi$ as a Graph Rewriting System

3.1 Informal Presentation

Communication and causal dependencies. Our goal is to represent $\rho\pi$ configurations as directed graphs. Nodes of graphs represent processes that appear in the configuration we encode. Edges in the graphs represent the causal dependencies between processes. Notice that, with this approach, tags are not needed, and memories require no explicit syntax: processes in memories are represented by nodes with outgoing edges.

Example 3. We encode $\nu k.\,([\kappa_1 :\ a\,\langle 0\rangle \ ||\ \kappa_2 :\ a(X) \triangleright (0); k] \ ||\ k :\ 0)$ as:



The two nodes on the left have outgoing arrows: there is a process (0 here) that causally depends on them, and they occur in a memory in $\rho\pi$. In this case, we have only one causal link: the interaction between the sender and the receiver creates the continuation. The tag k is used to track this causal link in the $\rho\pi$ configuration, and the two edges are used in the graph.

Name restrictions. Catching the structure of the causal dependence relations is straightforward. However, we also need to deal with names and ν binders. In particular, we must avoid name aliasing to respect $\rho\pi$'s original semantics. Our strategy consists in α-renaming channels with a fresh name every time a leading $\nu a.\,(...)$ is encountered, allowing us to safely remove the binder. To generate fresh names, we use the unique special node $\texttt{gen}(a_i)$ as a name generator, creating channel names a_i, with increasing indices. Throughout this paper, we assume that no such channel a_i is present in the initial process.

Example 4 (Managing name restrictions). Consider the following $\rho\pi$ term:

$$T = \kappa : \ \nu b.\,(\nu c.\,(b\,\langle 0\rangle \ ||\ c\,\langle 0\rangle))$$

Initially, the graph encoding of that term has a single process node, and the generator node (see Figure 5, left). Since the term contains a head restriction[7], a fresh name a_1 is generated (using the $\texttt{gen}(a_1)$ node as memory), and free occurrences of b in the continuation are replaced by a_1. We therefore only keep the continuation without the head restriction (i.e. $\nu c.\,(a_1\,\langle 0\rangle \ ||\ c\,\langle 0\rangle)$) in a new node. The generator is updated with the next fresh name available (here $\texttt{gen}(a_2)$). This leads to Figure 5, middle. Proceeding similarly with the next restriction ($\nu c.\,(...)$), we replace free occurrences of c with a_2, and we update the generator, leading to Figure 5, right.

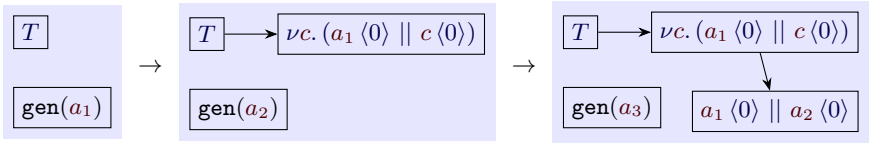


Fig. 5: Renaming channels and removing restrictions. Notice how the $\texttt{gen}(a_i)$ node always contains the next available name.

3.2 Graph Structure

Our graphs are composed of labeled vertices and unlabeled edges. Vertex labels are terms of **HO**π or $\texttt{gen}(a_i)$ for any i. Only a single vertex has a label $\texttt{gen}(a_i)$ for some i (thus ensuring freshness of names), others are labeled with **HO**π terms. The initial graph of a computation contains only two nodes: one labeled with the initial process, and one labeled with $\texttt{gen}(a_1)$.

3.3 Graph Rewriting Rules

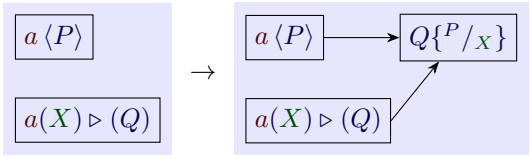
We now focus on the graph rewriting rules required to implement the semantics of $\rho\pi$. We begin with two core rules, (Fw) and (Bw), which correspond to the forward and backward reductions of $\rho\pi$, respectively. Contrary to $\rho\pi$, we deal with ν binders by substituting bound occurrences of the name with a fresh name and removing the binder. This enables the communication rule to operate simply by checking name equality, ignoring binders. To achieve this result, we introduce in Section 3.3 a (NuFresh) rule, which performs this substitution. Finally, although our graph-based approach removes the need for most structural equivalence rules, we must still address parallel composition: a parallel composition must be splittable into its sub-processes. In this section, we introduce the rules for splitting and merging nodes.

[7] We say a term has a head restriction when it is of the form $\nu a.\,(...)$ for some a.

Forward Computation. The only forward computation that $\rho\pi$ processes can perform is communication. For that, we match two vertices—a sender and a receiver—using the same channel name. When a match is found, we create a new node whose label corresponds to the continuation of the communication. As in $\rho\pi$, if a trigger (a.k.a. a receive) binds X in Q, upon receiving a message P, the process continues with $Q\{^P/_X\}$. We then add causal dependency edges from the original processes to this continuation. To prevent a communication from occurring multiple times, the rule is restricted: it can only be applied if both the sender and the receiver have no outgoing causal edges.

Notice that, formally, this rule allows communication on open channel names: the a in the rule is not bound nor restricted. The (NuFresh) rule, introduced below, opens restrictions, thereby enabling all communications as in $\rho\pi$'s original semantics. This approach allows us to rely solely on name equality to identify matching channels, avoiding complex binder analysis.

Rule: (Fw)

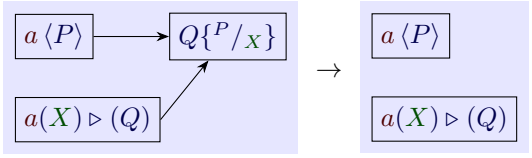


If there is no outgoing edge from the nodes in the left-hand side.

Backward Computation. In $\rho\pi$, backward computation is performed by removing processes and restoring the memory that generated them. The causal dependencies are tracked by using keys. In our approach, these causal dependencies are represented by graph edges.

Two points require special attention. First, the process to be removed has to be *live*, i.e. it must not have any consequences. This is checked by ensuring it has no outgoing edge[8]. Second, all consequences of the restored communication must be removed simultaneously. This is ensured by forbidding the existence of other outgoing edges from the restored process. In order to trigger the rule, all sub-processes of the continuation have to be merged together in a single vertex.

Rule: (Bw)



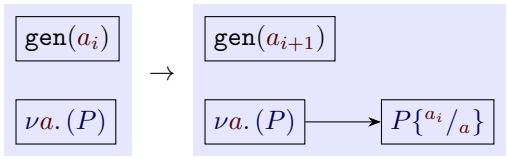
If there is no other outgoing edge from the three vertices of the left-hand side.

[8] Notice that the DPO approach prevents removed vertices to have edges except those that are explicitly shown in the left-hand side of the rule. We explicitly add the (spurious) condition in the rule for the sake of clarity.

Name Binding Rules. As explained above, to identify matching channels for communication, we rely on channel name equality, disregarding binders. To this end, we introduce the rule (NuFresh), which *opens* restrictions. This rule transforms a term of the form $\nu a.\,(P)$ into $P\{^{a_i}/_{a}\}$, where a_i is a fresh name. Freshness is ensured by using the unique generator node of the graph, which is incremented simultaneously with the application of the rule.

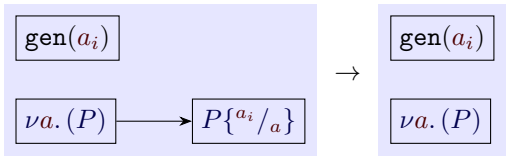
Notice that, since a_i is fresh, all surrounding open channel names are distinct from a_i. Moreover, bound names cannot participate in communications with (Fw) before being open themselves, i.e. freshly renamed. The correctness of this approach relies on the observation that, for all P, Q, a: $P \parallel \nu a.\,(Q) \equiv \nu a_i.\,(P \parallel Q\{^{a_i}/_{a}\})$, provided that a_i does not occur free in P, which is guaranteed when it is fresh. This term behaves similarly to $P \parallel Q\{^{a_i}/_{a}\}$ [9].

Rule: (NuFresh)



If there is no outgoing edge from $\nu a.\,(P)$.

Rule: (NuRestrict)



If there is no outgoing edge from $P\{^{a_i}/_{a}\}$.

Structural Equivalence Rules. As mentioned in the introduction of the section, using a graph approach simplifies most structural equivalence rules (e.g. commutativity and associativity of the parallel composition are ensured by the very structure of the graph). Nonetheless, one point must still be addressed: when a single vertex contains a parallel composition of processes, we have to be able to split it into two vertices, each containing one side of the composition. Conversely, this implies having the ability to merge such split vertices in order to reassemble them into a single vertex before backward computation. This mechanism is essentially the counterpart of tag splitting and merging in $\rho\pi$.

In this graph rewriting-based approach, the split operation is performed by replacing a vertex by two new vertices (and conversely for merging). The only point of attention is the preservation of incoming edges: they have to be replicated in the newly generated nodes. Since incoming edges can originate from different kind of ancestors (either from a communication or from a restriction

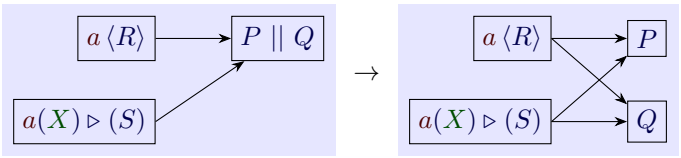
[9] Note, however, that this approach breaks barbed similarity of processes. In our graph approach: we cannot freely merge two graphs.

opening), we actually have a *family* of split (resp. merge) rules. In the following, we first introduce the splitting rules, and then their merging counterparts.

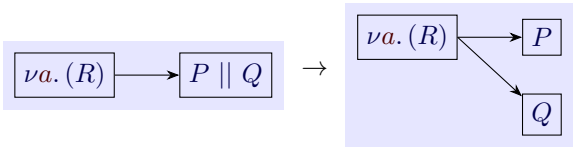
Splitting parallel processes. Splitting a parallel process consists in replacing a single node containing a parallel composition into two nodes containing both sides of the parallel composition[10].We introduce three splitting rules, depending on the causal ancestor(s) of the node being split: (SplitCom) applies when the process originates from a communication, (SplitNu) when it results from opening a restriction, and (SplitInit) when it corresponds to the initial process.

Moreover, no rule introduces outgoing edges from parallel processes. This enforces the invariant that parallel processes have no causal descendants, ensuring that the nodes being split do not belong to memories.

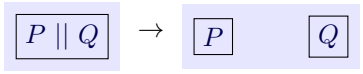
Rule: (SplitCom)



Rule: (SplitNu)



Rule: (SplitInit)



If there is no incoming edges to $P \parallel Q$ in the left-hand side.

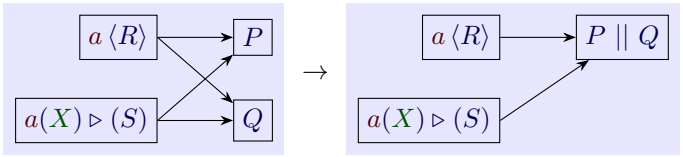
In the case of (SplitInit), we explicitly require that the initial $P \parallel Q$ node has no incoming edges, to indicate that it corresponds to the initial process. This restriction is, however, redundant: under the DPO semantics, any rule must explicitly specify all edges incident to removed nodes, which already prevents the rule from applying when incoming edges are present.

Merging parallel vertices. The (Bw) rule requires the vertex to be removed to be the only successor of the memory being restored. Consequently, if the continuation of a communication has been split

[10] Note that, contrary to $\rho\pi$'s structural congruence, which allows (and requires) a tag to be split into all subprocesses at once, our approach must rely on binary composition. This is because graph rewriting rules are defined over (finite and fixed) graphs: allowing an arbitrary number of parallel components to be split simultaneously would require an infinite family of rules.

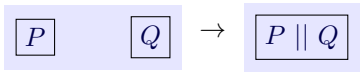
in order to continue the forward computation, the corresponding sub-processes must first be merged back into a single vertex to perform backward computation. For that, we introduce a family of merge rules. Similar to split rules, we have to accommodate three different cases, depending on the origin of the vertices to be merged: continuations created by a communication (MergeCom), by opening a restriction (MergeNu), or belonging to the initial process (MergeInit).

Rule: (MergeCom)



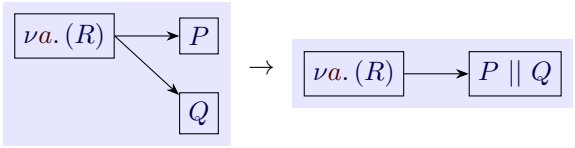
If there is no outgoing edges from P and Q in the left-hand side.

Rule: (MergeInit)



If there is no edges from or to P and Q in the left-hand side.

Rule: (MergeNu)



If there is no outgoing edges from P and Q in the left-hand side.

Notice that in the three rules, we explicitly add the condition that the vertices containing P and Q must have no outgoing edges. However, and as in the split rule, this restriction is already enforced by the DPO semantics, which prevents removed nodes to have edges other than those specified in the rule.

3.4 Correctness Criteria

In this section, we present the relationship between $\mathsf{G}\rho\pi\mathsf{F}$ and $\rho\pi$. A formal proof of the correspondence between the two systems lies beyond the scope of this work and is left for future research. First, we present how to convert a graph into a $\rho\pi$ configuration; and then we state the correctness conjecture.

Encoding $\rho\pi$ terms as graphs. Our goal is to encode any $\rho\pi$ configuration as a graph. To this end, we define a encoding function $[\![\cdot]\!]$ that maps a configuration to a graph. We assume this function is

constant for structurally equivalent configurations. Therefore, for the sake of simplicity, we restrict its definition to configurations in thread-normal form.

The encoding proceeds as follows. First, it ignores leading channel restrictions, since these are replaced with fresh names in $\mathsf{G}\rho\pi\mathsf{F}$ and, in thread-normal form, names are not aliased. It then creates a node for each tagged process or memory. Edges are introduced to represent causal dependencies, which are tracked via tags. Formally, we define two functions nodes : $C \mapsto V$ and edges : $C \mapsto E$, which return the set of nodes and edges, respectively, of the graph corresponding to the configuration to encode:

$$\text{nodes}(C) = \{\kappa :\ P \mid \kappa :\ P \in C\}$$

$$\text{edges}(C) = \{(\kappa_1 :\ P, \kappa_2 :\ Q) \mid \exists\, [\mu; k]^\circ \in_M C \cdot k >_C \kappa_2 \ \wedge\ \kappa_1 :\ P \in \mu\}$$

where $\kappa :\ P \in C$ holds iff $\kappa :\ P$ appears in C, including in memories; and $[\mu; k]^\circ \in_M C$ holds iff $[\mu; k]^\circ$ appears in C. Finally, we assume a function rmvTags which removes tags from processes in nodes of a graph, while preserving edges. Thus, our encoding function is $[\![C]\!] = \text{rmvTags}(\langle \text{nodes}(C), \text{edges}(C)\rangle)$.

In future works, our goal would be to prove a form of behavioural correspondence between $\rho\pi$ configurations and their encodings in $\mathsf{G}\rho\pi\mathsf{F}$. We conjecture the relation $\{\langle C, [\![C]\!]\rangle\}$ is a weak bisimulation. Reduction rules of $\rho\pi$ are matched by (Fw) and (Bw), but additional steps are needed in $\mathsf{G}\rho\pi\mathsf{F}$ to mimic structural equivalence reshaping. Having such bisimulation is sufficient to prove that our semantics preserves $\rho\pi$'s causal consistency.

3.5 Example

Figure 6 illustrate the derivation of the *replication* construct by adapting the **HO**π replicator introduced in [2]:

$$\nu c.\,(c(X) \triangleright (X \parallel c\langle X\rangle \parallel P) \parallel c\langle c(X) \triangleright (X \parallel c\langle X\rangle \parallel P)\rangle).$$

Intuitively, the left process $c(X) \triangleright (X \parallel c\langle X\rangle \parallel P)$ acts as a replication core: upon receiving a process X, it executes X, re-sends it along the same channel, and generates an occurrence of a process P. By sending this replication core to itself, we obtain an infinite behaviour generating a new process P each time a communication on c occurs. We illustrate the semantics of $\mathsf{G}\rho\pi\mathsf{F}$ using this example. For the sake of simplicity, throughout the example, the generator node is omitted whenever it plays no role in the transition. For the sake of conciseness, unless a communication rule occurs, we note $\mathbf{R}_c$ for the replication core $c(X) \triangleright (X \parallel c\langle X\rangle \parallel P)$, where the index denotes the channel used for replication, in nodes that do not take part in the transition.

The derivation starts with a (NuFresh) step, which introduces a fresh channel (namely c_1) to ensure locality and avoid name capture. The (SplitNu) rule then splits the body of the restriction, generating two nodes communicating through c_1.

Successive applications of (Fw) and (SplitCom) develops the communication structure, exhibiting a sender-receiver couple and making them interact.

Then, the subsequent backward steps ((Bw), (MergeCom), (MergeNu), (NuRestrict)) shows that the transformation is reversible and preserves the global structure of the system, without loss of information. Finally, an additional (NuFresh) step (e.g. producing c_2) highlights the α-renaming mechanism of channel management.

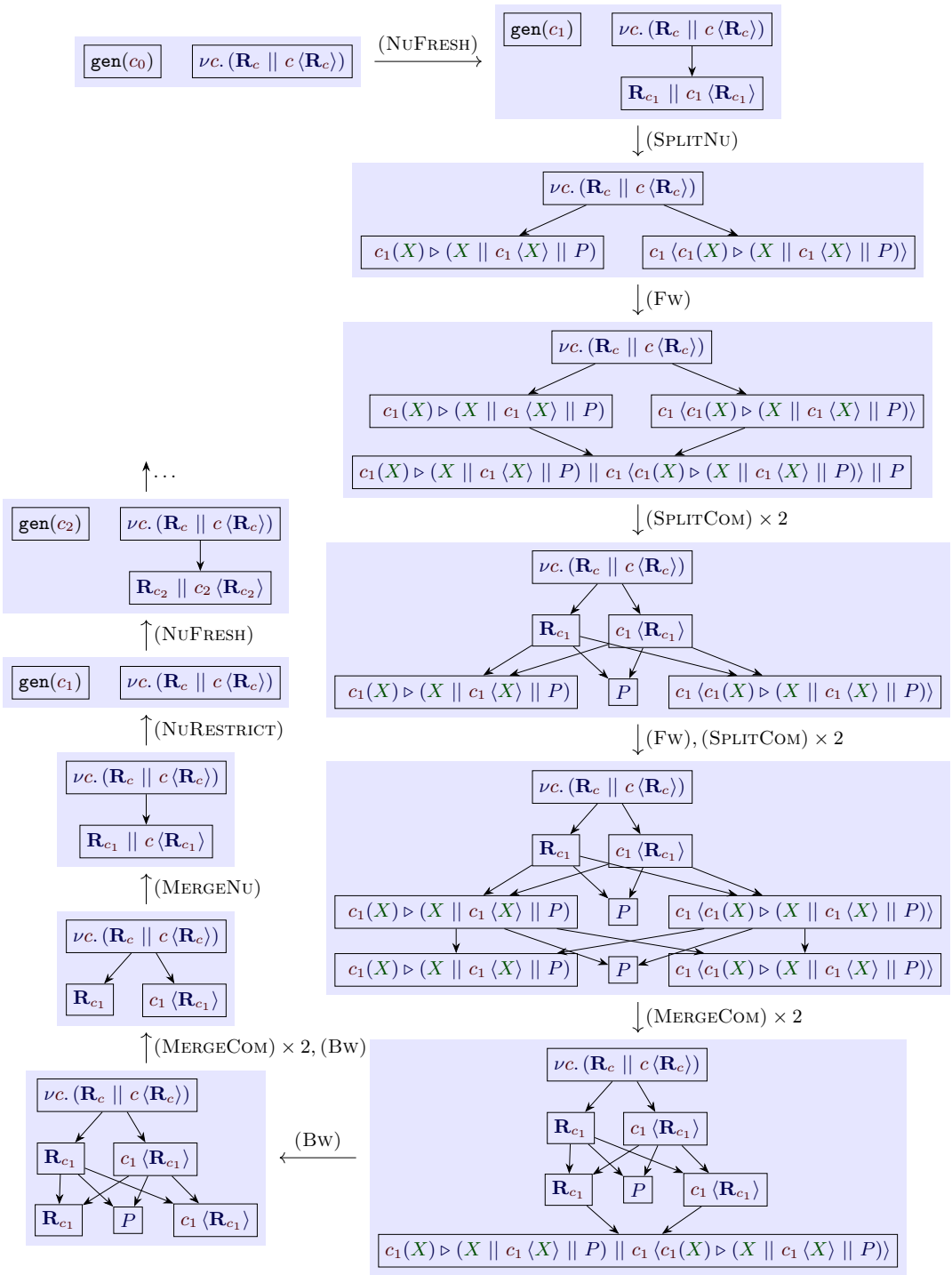


Fig. 6: Example of derivation starting from the process $\nu c.\,(\mathbf{R}_c \parallel c\,\langle \mathbf{R}_c \rangle)$ on the top-left corner.

4 Implementation

G$\rho\pi$F is the Python implementation accompanying this paper, designed to provide both a faithful realization of the graph-based semantics of $\rho\pi$ and an accessible interface for experimentation. The interpreter is available at [5], and instructions for running it are given in Appendix A.

Overview. Since $\rho\pi$ is uncontrolled and π-calculi are inherently nondeterministic, G$\rho\pi$F is designed to be interactive. Its main goal is to let users, including those unfamiliar with the calculus, experiment with processes by explicitly trying possible (and impossible) applications of rewriting rules. Users interact through a web interface (see Figure 7), where buttons at the top allow applying rules to nodes previously selected in the graph. This enables step-by-step exploration of process reductions and causal dependencies. For the sake of clarity, split (resp. merge) rules have a single button instead of three separate ones. Depending on the selected node, the adequate rule is inferred.

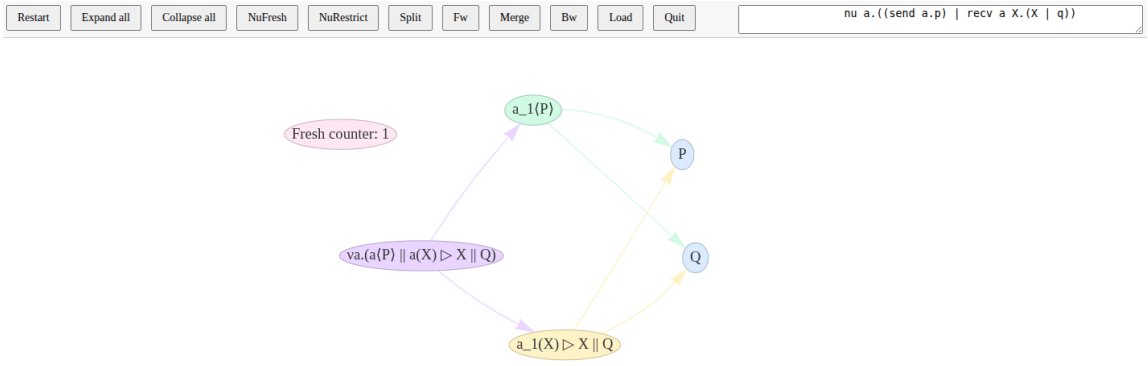


Fig. 7: Overview of the web interface of G$\rho\pi$F.

Automated mode. In addition to the interactive mode, G$\rho\pi$F provides an automated execution mode. The *Expand all* button repeatedly applies all enabled forward-oriented rules, namely (NuFresh) and the family of splitting rules ((SplitInit), (SplitNu), (SplitCom)), until no further application is possible. Conversely, the *Collapse all* button repeatedly applies merge rules ((MergeInit), (MergeNu), (MergeCom)) together with (Bw), rolling back the configuration as far as possible. These automated operations allow users to quickly explore maximal forward computations or revert configurations without manually selecting individual nodes[11]. Combined with the interactive mode, they provide a user-friendly way to explore the global behavior of $\rho\pi$ processes.

Parser. On the top-right corner of the page, the user can write a $\rho\pi$ term, which is then parsed and loaded as the initial term of the computation using the `Load` button. The interpreter parses $\rho\pi$ terms using a dedicated grammar that closely mirrors the formal syntax. Parallel composition is left-associative, and parentheses can be used to group processes. When writing terms, only variables

[11] Note that, in order to avoid skipping steps, rules must be applied in a specific order (i.e. by performing merge rules before (Bw))

can be capitalized, while channel and process names must be lowercase. Parallel composition is denoted with `|`, and the termination process is written as `0`. The accepted grammar is shown in Appendix A.

Example 5 (GρπF input terms). The following entry corresponds to the communication example shown in Example 1.

```
nu a.((send a.p) | recv a X.(X | q))
```

The following entry corresponds to the replicator example shown in Section 3.5.

```
nu c.(recv c X.((X | send c.X) | p)) | send c.(recv c X.((X | send c.X) | p))
```

5 Conclusion

5.1 Related Work

Multiple works use graph-based descriptions of reversible process calculi (at least informally). In this subsection, we review some of these contributions to highlight how our formalization can bridge the gap between intuitive graphical ideas and formal descriptions.

First, Petri Nets enjoy intuitive graph-based representations. Extensive work has been propose in recent years to study reversibility in Petri Nets [26,25,22].

Regarding process calculi, in [12], Fabbretti et al. informally present a mechanism for rollback when some causal dependencies are lost. While the authors did not focus on providing a formal framework to introduce their idea, it can easily be expressed in ours: they examine what happens when some vertices or edges in the graph are arbitrarily removed, and how to recover from such removal. From this description, we can close the gap and express their work in $\rho\pi$: they study what happens when some memories or tags (or even live processes) are erased in $\rho\pi$'s configurations.

On the other hand, graph-based approaches help with pattern identification. In [20], Lanese et al. develop a variant of **roll**$-\pi$ [17] (itself a variant of $\rho\pi$ with controlled reversibility) where some communications can be committed. Thanks to a graph-based approach, they realized a form of duality between rollbacks and commits: each operation removes extrema of the *happens-before* relation. Reverting communication consists in removing vertices without outgoing edges, while committing consists in removing vertices without incoming edges. This illustrates how graphical approaches could be useful, even for accustomed researchers.

On the practical side, graph transformation has led to the development of various tools and environments. General-purpose systems such as `AGG` [27] support algebraic graph transformation based on the DPO approach, together with features such as critical pair analysis and consistency checking. Similarly, `GROOVE` [14] provides a modelling and verification environment for graph transformation systems, with automated statespace generation, exploration, and model checking. More generally, the `Maude` [6] is a rewriting logic framework that offers a general infrastructure for specifying and analyzing rewrite systems.

Graph rewriting has also been applied in more specialized domains. In systems biology, rule-based languages such as `Kappa` [4] and `BioNetGen` [13] rely on graph rewriting to model molecular interactions and their causal dependencies. In model-driven engineering, `Henshin` [1] and `VIATRA` [28] support rulebased model transformations over the Eclipse Modeling Framework, where transformation rules are expressed as graph patterns and executed directly on model instances.

Although our system could in principle be implemented using existing tools, we developed a dedicated interpreter in order to better control the semantics and adapt design choices to the specific needs of reversible process calculi.

5.2 Future Work

We introduced a graph-based semantics for $\rho\pi$. While the rules are close to $\rho\pi$'s original semantics, in this work, we focused on the presentation and the implementation of the framework, without formally proving the correspondence with the original semantics. Therefore, in the current state, the framework ought to be understood as a supporting tool for intuitively introducing readers to $\rho\pi$, or to develop intuition on reversible process calculi. In the future, we hope to formally prove the correspondence with $\rho\pi$, thus allowing researchers to develop variants (such as the one mentioned in the related works above) in a graphical way, while keeping a formal relation to well established $\rho\pi$.

An example of such variants would be to merge memories to have coarsed grained memories. Such variant would be difficult to formalize in $\rho\pi$, due to the accumulation of tags, but would be very intuitive in our graph approach: one simply needs to allow merging vertices while preserving incoming and outgoing edges.

Finally, while G$\rho\pi$F currently supports both interactive exploration and automated forward/backward rule application, future work will focus on improving automation features. One possibility is to allow the tool to automatically apply sequences of rules, exploring the different ways a process can evolve or searching for paths between two states, inspired by e-graph search techniques [23,24]. This would help users experiment with process behaviors more efficiently and observe possible computations without manually selecting each step. Such automation could also serve as a first step toward checking simple properties, such as whether a particular state can be reached, while keeping the tool easy to use.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place emf model transformations. In: International Conference on Model Driven Engineering Languages and Systems. pp. 121–135. Springer (2010)
2. Barwell, A.D., Hou, P., Vassor, M., Yoshida, N.: Encoding Choice and Replication in roll-π. In: Glück, R., Kaarsgaard, R. (eds.) Reversible Computation. pp. 27–36. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-97063-4_3
3. Bonchi, F., Gadducci, F., Kissinger, A., Sobocinski, P., Zanasi, F.: String diagram rewrite theory i: Rewriting with frobenius structure. J. ACM **69**(2) (Mar 2022). https://doi.org/10.1145/3502719
4. Boutillier, P., Maasha, M., Li, X., Medina-Abarca, H.F., Krivine, J., Feret, J., Cristescu, I., Forbes, A.G., Fontana, W.: The kappa platform for rule-based modeling. Bioinformatics **34**(13), i583–i592 (2018)
5. Cailler, J., Vassor, M.: G$\rho\pi$F: Graphical $\rho\pi$ framework (Feb 2026). https://doi.org/10.5281/zenodo.18662588
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All about Maude-a High-Performance Logical Framework: How to Ppecify, Program, and Verify Systems in Rewriting Logic, vol. 4350. Springer (2007)
7. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation: vol. 2: Applications, Languages, and Tools. World Scientific Publishing Co., Inc., USA (1999)
8. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation: vol. 3: Concurrency, Parallelism, and Distribution. World Scientific Publishing Co., Inc., USA (1999)

9. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series, Springer-Verlag, Berlin/Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
10. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: 14th Annual symposium on switching and automata theory (swat 1973). pp. 167–180. IEEE (1973)
11. Ehrig, H., Rozenberg, G., rg Kreowski, H.J.: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 3. world Scientific (1999)
12. Fabbretti, G., Lanese, I., Stefani, J.B.: Reversibility with Holes. In: Mogensen, T.Æ., Mikulski, Ł. (eds.) Reversible Computation. pp. 69–74. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-62076-8_5
13. Faeder, J.R., Blinov, M.L., Hlavacek, W.S.: Rule-based modeling of biochemical systems with bionetgen. In: Systems biology, pp. 113–167. Springer (2009)
14. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using groove. International journal on software tools for technology transfer **14**(1), 15–40 (2012)
15. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984. pp. 558–565 (Jul 1978), https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/
16. Lanese, I., Lienhardt, M., Mezzina, C.A., Schmitt, A., Stefani, J.B.: Concurrent Flexible Reversibility. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 370–390. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_21
17. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.B.: Controlling Reversibility in Higher-Order Pi. In: Katoen, J.P., König, B. (eds.) CONCUR 2011 – Concurrency Theory. pp. 297–311. Springer Berlin Heidelberg, Berlin, Heidelberg (2011), https://link.springer.com/chapter/10.1007/978-3-642-23217-6_20
18. Lanese, I., Mezzina, C.A., Stefani, J.B.: Reversing Higher-Order Pi. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory. pp. 478–493. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_33
19. Lanese, I., Mezzina, C.A., Stefani, J.B.: Reversibility in the higher-order π-calculus. Theoretical Computer Science **625**, 25–84 (Apr 2016). https://doi.org/10.1016/j.tcs.2016.02.019
20. Lanese, I., Mezzina, C.A., Vassor, M.: Bounded Reversibility in HOπ. In: Mezzina, C.A., Schmitt, A. (eds.) Components Operationally: Reversibility and System Engineering: Essays Dedicated to Jean-Bernard Stefani on the Occasion of His 65th Birthday, pp. 24–45. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-99717-4_2
21. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science **109**(1-2), 181–224 (1993)
22. Melgratti, H., Mezzina, C.A., Ulidowski, I.: Reversing P/T Nets. In: Riis Nielson, H., Tuosto, E. (eds.) Coordination Models and Languages. pp. 19–36. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-22397-7_2
23. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. Journal of the ACM (JACM) **27**(2), 356–364 (1980)
24. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: International Conference on Rewriting Techniques and Applications. pp. 453–468. Springer (2005)
25. Philippou, A., Psara, K.: Reversible Computation in Petri Nets. In: Kari, J., Ulidowski, I. (eds.) Reversible Computation. pp. 84–101. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_6
26. Philippou, A., Psara, K., Siljak, H.: Controlling Reversibility in Reversing Petri Nets with Application to Wireless Communications. In: Thomsen, M.K., Soeken, M. (eds.) Reversible Computation. pp. 238–245. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_15

27. Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: International workshop on applications of graph transformations with industrial relevance. pp. 446–453. Springer (2003)
28. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transformation platform: Three generations of the viatra framework. Software & Systems Modeling **15**(3), 609–629 (2016)
29. Vassor, M.: Reversibility and Predictions. In: Yamashita, S., Yokoyama, T. (eds.) Reversible Computation. pp. 163–181. Lecture Notes in Computer Science, Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-79837-6_10

A Running the Interpreter

We assume the reader has a working Python environment on a Linux machine. The installation procedure below sets up a virtual environment in which the dependencies will be installed. The program can be easily uninstalled by removing the directory containing the virtual environment.

Getting the files. The source files can be obtained from [5]. After downloading the archive, extract it into a repository (or any working directory) where you wish to install the tool.

Setting up the virtual environment.

```
$ python3 -m venv path/to/venv/directory/
$ source path/to/venv/directory/bin/activate
```

Note that the virtual environment directory will be created if need be. Users of csh and fish have to source `activate.csh` (resp. `activate.fish`) instead.

To deactivate later:

```
$ deactivate
```

Installing dependencies. G$\rho\pi$F requires the following libraries: networkx (graph manipulation), pyvis (visualization), flask (web interface), and lark (parser) In order to install them, run the following command:

```
$ pip install networkx pyvis flask lark
```

Running the interpreter. The command line to run G$\rho\pi$F is:

```
$ python path/to/sources/app.py
```

This command will start a web server, which is the graphical interface of our interpreter. Go to `http://127.0.0.1:5000` (by default) to use the interpreter. You are now ready to use G$\rho\pi$F!

Parser Grammar. The grammar for $\rho\pi$ terms in the G$\rho\pi$F interpreter is as follows:

```
PROCESS : /[a-z][a-zA-Z0-9_]*/
VARIABLE : /[A-Z][a-zA-Z0-9_]*/
CHANNEL : /[a-z]+/

term : parallel

parallel : application
         | parallel "|" application

application : nu
            | send
            | recv
            | atom

nu : "nu" CHANNEL "." term
send : "send" CHANNEL "." term
recv : "recv" CHANNEL VARIABLE "." term

atom : PROCESS
     | VARIABLE
     | "0"
     | "(" term ")"
```

Loading examples. Several example terms are available in the example folder. You can copy and paste them directly into the input area of the interface and click *Load*.

Example 1: Communication. The entry

```
nu a.((send a.p) | recv a X.(X | q))
```

corresponds to

$$\nu a.\,(a\,\langle P\rangle \;||\; a(X) \triangleright (X \;||\; Q)),$$

where P and Q are unknown processes.

This example illustrates a single communication. Two processes, P and Q, interact through a restricted channel a. The sender broadcasts process P, which is then substituted for the variable X in the receiver, producing the continuation $P \;||\; Q$. This process can then be split into two parallel processes, P and Q.

Suggested steps in G$\rho\pi$F:

- Apply `NuFresh` to open the restriction and generate a fresh channel.
- Apply `Split` to separate the parallel components.
- Select the sender and the receiver, then apply `Fw` to perform the communication.
- A new node labeled $P \;||\; Q$ appears, with causal edges from both the sender and the receiver to the continuation.
- Apply `Split` again to obtain two separate processes P and Q.
- Optionally, apply `Merge` and then `Bw` to undo the communication and restore the original processes.

Example 2: Replication. The entry

```
nu c.(recv c X.(X | send c.X)) | send c.(recv c X.((X | send c.X) | p))
```

corresponds to

$$\nu c.\,(c(X) \triangleright (X \parallel c\,\langle X\rangle) \parallel c\,\langle c(X) \triangleright (X \parallel c\,\langle X\rangle \parallel P)\rangle).$$

This term is adapted from the **HO**π replicator shown in [2]. After a few communications on channel c, the process P is replicated arbitrarily many times.

Suggested steps in G$\rho\pi$F:

- Apply `NuFresh` to open the restriction on channel c.
- Repeatedly apply `Split` and `Fw` (or use `Expand all`) to observe the growth of the process and the accumulation of causal dependencies.
- Use `Collapse all` to explore backward reductions and progressively revert the computation.