A Graph-Based Tool to Embed the π-Calculus into a Computational DPO Framework

Jakob L. Andersen[1], Marc Hellmuth[2], Daniel Merkle[1,3],
Nikolai Nøjgaard[1,2], and Marco Peressotti[1]

[1] Department of Mathematics and Computer Science, University of Southern
Denmark, Odense, DK
`{jlandersen,daniel,nojgaard,peressotti}@imada.sdu.dk`
[2] Institute of Mathematics and Computer Science, University of Greifswald, DE
`mhellmuth@mailbox.org`
[3] Harvard Medical School, Department of Systems Biology, Boston, MA, US

Abstract Graph transformation approaches have been successfully used to analyse and design chemical and biological systems. Here we build on top of a DPO framework, in which molecules are modelled as typed attributed graphs and chemical reactions are modelled as graph transformations. Edges and vertexes can be labelled with first-order terms, which can be used to encode, e.g., steric information of molecules. While targeted to chemical settings, the computational framework is intended to be very generic and applicable to the exploration of arbitrary spaces derived via iterative application of rewrite rules, such as process calculi like Milner's π-calculus. To illustrate the generality of the framework, we introduce `EpiM`: a tool for computing execution spaces of π-calculus processes. `EpiM` encodes π-calculus processes as typed attributed graphs and then exploits the existing DPO framework to compute their dynamics in the form of graphs where nodes are π-calculus processes and edges are reduction steps. `EpiM` takes advantage of the graph-based representation and facilities offered by the framework, like efficient isomorphism checking to prune the space without resorting to explicit structural equivalences. `EpiM` is available as an online Python-based tool.

Keywords: Double Pushout, Process Calculi, Typed Attributed Graphs, Graph Isomorphism

1 Introduction

Graph transformation approaches have been shown to provide formalisms that elegantly facilitate the construction of reaction rules in organic chemistry and biology [2, 10, 14]. Many of these frameworks, while aimed at modelling organic chemistry and biology, are constructed as generic foundations which can be used to model in a wider variety of domains.

The framework focused on in this paper is MØD and is traditionally used to model organic chemistry [2, 3]. Here, graphs represent molecules, transformation rules specify how molecules can interact, and direct derivations represent

concrete chemical reactions. Specifically, graph transformations are in `MØD` performed with the double-pushout (DPO) approach, with injective graph morphisms [11, 13]. A higher-level "strategy framework" then allows for programming the sequence of rules to apply on sets of graphs, while performing the necessary graph isomorphism checks [1]. Additionally, graphs in `MØD` are labelled with first-order terms which allows for the specification of even more abstract rules with named variables as attributes.

In this paper we will use the foundational framework of `MØD` to implement a tool for computing execution spaces of π-calculus processes. π-calculus is a process algebra that, like all process algebras, is concerned with the problem of formally modelling concurrent systems [18,19]. A process in π-calculus represents a "unit" of computation. In a concurrent system several of such processes can run concurrently and communication between processes occurs via channels on which channel names are sent. Given a set of processes, all running concurrently, the execution space is then all possible state transitions up to structural congruence.

The modelling of π-calculus as a graph transformation system is not a new concept [12, 17]. Similarly, tools already exist that simulates π-calculus [7, 20]. Few tools, however, simulates the execution space of a process up to structural congruence, and no tools, to our knowledge, does this by using established graph transformation concepts such as the DPO approach.

Here, we present the tool `EpiM`, which is a Python-based library that embeds the language of π-calculus into the framework of `MØD`. It allows the modelling of processes as simple Python expressions which are then encoded into graphs. The execution space of processes is modelled as a set of transformation rules applied on their corresponding graph encodings. Structural congruence is checked using efficient graph isomorphism checking of graphs labelled with first-order terms provided by `MØD`. The encoding and the simulation of process communication via graph transformations are based on the results established in [12], with the notable modification that our encoding obtains simple labelled graphs, while their model is concerned with directed typed hypergraphs. A web front-end of `EpiM` is provided at `http://cheminf.imada.sdu.dk/epim`.

2   Preliminaries

2.1   The π-calculus

The π-calculus is a mathematical model of concurrent interacting processes (P, Q, R, ...) that communicate using names (x, y, z, ...). A name represents a communication channel or session, and can be sent and received as a message [18,19]. Process terms are given by the following grammar:

$$
\begin{aligned}
P, Q ::= \ & \overline{x}[y].P & & \textit{output name y on channel x and continue as P} \\
\mid \ & x(y).P & & \textit{input a name on x, bind it to y and continue as P} \\
\mid \ & P \mid Q & & \textit{run processes P and Q in parallel (parallel composition)} \\
\mid \ & P + Q & & \textit{run either P or Q (choice)} \\
\mid \ & (\nu x)P & & \textit{bind x in P (restriction)} \\
\mid \ & 0 & & \textit{terminated process}
\end{aligned}
$$

The sets $fn(P)$ and $bn(P)$ of free and bound names in a process P are defined as expected as well as α-conversion.

Intuitively, process terms that differ solely on the order of parallel composition $|$, sums $+$, and restrictions ν represent the same process. This intuition is formalised by *structural equivalence*, i.e., the least relation $\equiv$ on process terms that is a congruence w.r.t. the grammar above and closed under the Abelian laws for $\diamond \in \{|, +\}$ under 0:

$$P \diamond Q \equiv Q \diamond P \qquad (P \diamond Q) \diamond R \equiv P \diamond (Q \diamond R) \qquad P \diamond 0 \equiv 0$$

and under the distributivity laws for restriction:

$$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \quad (\nu x)0 \equiv 0 \quad (\nu x)(P \,|\, Q) \equiv (\nu y)P \,|\, Q \text{ for } x \notin fn(Q).$$

For the sake of simplicity, we adopt the common assumption that choices are always guarded (i.e. every branch in a sum $+$ is either a choice or a communication): in the sequel we assume, similarly to [12], the following grammar:

$$P, Q \coloneqq M \mid P \,|\, Q$$
$$M, N \coloneqq 0 \mid x(y).P \mid \overline{x}[y].P \mid M + N$$

The semantics of process terms is given by the reduction relation $\rightarrow$, i.e., the least relation closed under the rules below:

$$\frac{}{(x(y).P + M) \,|\, (\overline{x}[z].Q + N) \rightarrow P\{y/z\} \,|\, Q} \text{ Com}$$

$$\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q} \text{ Str} \qquad \frac{P \rightarrow Q}{(\nu x)P \rightarrow (\nu x)Q} \text{ Res} \qquad \frac{P \rightarrow P'}{P \,|\, Q \rightarrow P' \,|\, Q} \text{ Par}$$

Rule Com models synchronous communication between two processes, possibly under a non-deterministic context (subterm M may simply be 0). Rule Str ensures structurally equivalent processes have the same behaviour. Rule Par models the (interleaved) execution of parallel components. Rule Res allows execution under a restriction.

2.2   Graphs, Transformation Rules, and the `MØD` framework

In this section we will be giving a very brief overview of the `MØD` framework and refer to [2] for a full overview. Encodings of processes will be represented as simple labelled graphs, where vertices and edges of graphs are labelled with first-order terms. We use the common definitions of graph and monomorphisms for labelled graphs, where we require a most general unifier to exist for the set of term mappings induced by the underlying graph morphisms. For full details see Companion Technical Report [6].

Graph transformation rules are modelled in the traditional DPO framework with injective morphisms. See [13] of an overview of the DPO approach. The span of a rule, $L \leftarrow K \rightarrow R$ is illustrated in Fig. 4. Using first-order terms as labels is useful, since it allows us to specify abstract graph transformation rules

such as in Fig. 4, where any term leading with an underscore, e.g., $\_X$, specifies a variable term.

The application of graph transformation rules on some input graph, is done by using a strategy framework, where a specific order in which the graph transformation rules should be applied can be specified. The direct derivations obtained from applying a given strategy on an input graph is illustrated by a derivation graph. A derivation graph is a directed graph, where vertices are graphs obtained from rule applications and where an edge (G, H), represents a direct derivation $G \Rightarrow H$. Any pair of graphs in the derivation graph are non-isomorphic, and as a result the closure up to isomorphism of a given strategy can be automatically computed by MØD. See Fig. 2 for an example.

3   Encoding π-calculus

Suppose we wanted to model the process $P = x(z).\bar{z}[w] \,|\, \bar{x}[y]$ as a graph. To encode P in EpiM we can write the following:

```
1   x, y, z, w = names("x y z w")
2   p1 = Process().input(x, z).output(z, w)
3   p2 = Process().output(x, y)
4   P = p1 | p2
5   G = P.encode()
6   G.print()
```

Line 1 defines the names to be used in P. Line 2–3 defines the behaviour of each subprocess in P, while line 4 defines P as the parallel composition between them. Finally, line 5 encodes P as a graph that is printed in line 6. The resulting graph is illustrated in Fig. 1a. The encoding is based on [12] modified to work on simple labelled graphs instead of typed directed hypergraphs.

We denote the graph encoded by a process P as $[P]$. The overall topology of $[P]$ can be thought of as tree-like, describing the behaviour of P, just like a syntax tree for a context-free grammar, but where names are shared between operators. Since MØD works on undirected graphs, $[P]$ is equipped with a special vertex with the label go, representing the "root" of the tree. This will be useful later, to be able to specify the "top" of the graph in transformation rules.

All other vertices in $[P]$ is wrapped in one of two functions $v(\_X)$ or $t(\_X)$. Any vertex equipped with the function v represents a name with the value of $\_X$. On the other hand, vertices equipped with t describes the behaviour of the corresponding process or represents an implementation detail that will prove useful later. More precisely, the argument of t can represent input/output operators $t(in)/t(out)$, or parallel/sum composition operators $t(p)/t(s)$.

We allow vertices with the terms $t(p)$ or $t(s)$ to have a degree less than 3 as seen in Fig. 1a. In this case, the vertices do not actually represent a corresponding parallel or sum composition, but are useful implementation details when implementing the reduction mechanism with graph transformation. Any input or output operator is followed by a vertex with the term $t(p)$, regardless of $t(p)$ representing a parallel composition or not. Moreover, every vertex with the term $t(p)$, is either a leaf or immediately followed by a vertex with the label $t(s)$.
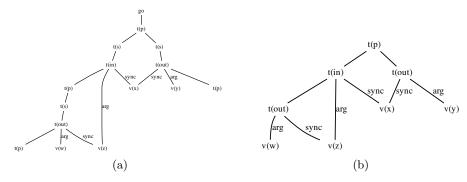
Figure 1: (a) The encoded process $[P]$ for $P = x(z).\overline{z}[w] \,|\, \overline{x}[y]$. (b) The simplified version of $[P]$.

Any prefix operator points to a name corresponding to the received or sent channel. Since the graphs considered here are not embedded, and hence do not have an order on the neighbourhood of a vertex, we label an edge emanating from a prefix operator with "sync" or "arg" whether the corresponding name is used for synchronization or as the argument for the operator. If a name is both the channel and argument of an operator, then the corresponding edge is equipped with "arg-sync" since MØD does not support parallel edges.

The vertices corresponding to names of the process are shared among vertices corresponding to input and output operators. In this way, if an input and output operator synchronize on the same name, it is illustrated in the resulting graph by both of them pointing to the vertex corresponding to that name. For example, the outermost input and output operator of the process encoded in Fig. 1a both synchronize on the free name x, and hence they both contain an edge to the vertex with the term $v(x)$.

Many of the vertices found in $[P]$ are implementation details, that are useful for simulating process reductions, but makes it difficult to interpret the behaviour of P simply by looking at $[P]$. In this regard, by default, EpiM filters away such implementation details, and instead depicts a simplified version of the encoding illustrated in Fig. 1b.

4   Computing Execution Spaces

Suppose we were given the process $P = x(z).\overline{z}[w] \,|\, (\overline{x}[y] + \overline{x}[y])$ and we are able to encode it into its graph equivalent $[P]$. Supplied with $[P]$, we want to compute the execution space of P. In EpiM we can write the following:

```
1  x, y, z, w = names("x y z w")
2  p1 = Process().input(x, z).output(z,w)
3  p2 = Process().output(x, y) + Process().output(x, y)
4  P = (p1 | p2)
5  exec_space = ReductionDG(P)
```
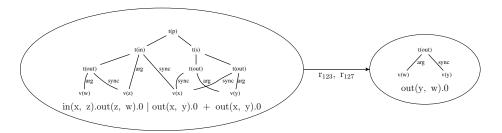
Figure 2: The execution space for $P = x(z).\overline{z}[w] \,|\,(\overline{x}[y] + \overline{x}[y])$.

```
6   exec_space.calc()
7   exec_space.print()
```

Line 1–4 specifies the process P as explained in the previous section. Line 5–6 computes all possible reductions, while the result is printed in line 7. The result is illustrated in Fig. 2. The execution space of P is illustrated as a directed graph G where each vertex in G corresponds to a process encoding, $[Q]$, derived from reductions on P, while each edge, $([Q], [R])$, represents the reduction $Q \to R$, for some processes Q and R. For instance, Fig. 3 contains the transformation that encodes the reduction $x(z).\overline{z}[w] \,|\,(\overline{x}[y] + \overline{x}[y]) \to \overline{y}[w]$.

In practice, the reduction step is simulated as three distinct strategies. The first strategy, $\mathcal{R}_\pi$, takes care of the actual reduction $(\to)$, while the two other strategies, $\mathcal{R}_{gc}$, $\mathcal{R}_m$, functions as "house-cleaning", ensuring that the transformed graphs represent process encodings.

The strategy $\mathcal{R}_\pi$ consists of two rules, one of which is depicted in Fig. 4. The rule in Fig. 4 simulates the reduction $(x(y).P + M) \,|\,(\overline{x}[w].Q + R) \to P\{y/w\} \,|\, Q$. The match of the rule searches for the parallel composition on the outermost prefix operators, i.e., a vertex labelled $t(p)$ attached to the root. Recall that every vertex corresponding to a parallel composition is followed by the sum composition vertex with the label $t(s)$. One of these branches of the sum composition vertex is matched such that we find an input and output operator vertex both synchronizing on the same name.

When a match is found, the reduction is then simulated by deleting the prefix operators and marking their continuations (which are always followed by a parallel composition vertex) to be coalesced into the "top" parallel composition operator. Additionally, the name y is now bound to w, which is simulated by merging the corresponding vertices.

An example of applying $\mathcal{R}_\pi$ to an encoded process is shown in Fig. 3 as the direct derivation $[P] \Rightarrow G'$, where $[P]$ describes the process defined in the start of this section. Note, that the rules are constructed such that when applying a rule any resulting connected component not containing the root vertex will never be part of the corresponding process encoding and hence such components are not depicted here. The input operator in P can synchronize on the output operator in both branches of P, and similarly there exists two matches of the rule depicted in Fig. 4 when applied to $[P]$. Either match of the rule, however,
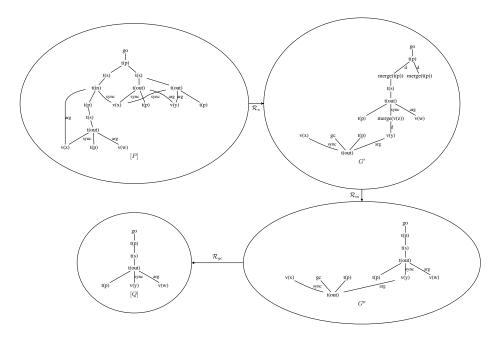
Figure 3: The application of each strategy, $\mathcal{R}_\pi$, $\mathcal{R}_m$, $\mathcal{R}_{gc}$, to simulate the reduction $P \to Q$.

leads to a graph isomorphic to G'. The graph G' does not represent an actual encoding of a process. The reason for this is two-fold: First, we have simply marked vertices that must be coalesced, but not actually coalesced them yet, due to the DPO approach in MØD only supporting injective morphisms. Second, the encoding of any branch not chosen, in this case one of the output operators of P, must be deleted. The coalescing of vertices is done by $\mathcal{R}_m$, obtaining the graph G'' from G' in Fig. 3, while the deletion of non-chosen summation branches is done by $\mathcal{R}_{gc}$, finally resulting in the graph $[Q]$ obtained from G'' in Fig. 3. For more details on $\mathcal{R}_m$ and $\mathcal{R}_{gc}$ see Companion Technical Report [6].

It was shown in [12] that for the resulting graph $[Q]$ there exists some process Q with the encoding $[Q]$. Let $[P] \xRightarrow{\mathcal{R}_\pi}_* [Q]$ be the sequence of derivations obtained by applying $\mathcal{R}_\pi$, $\mathcal{R}_m$, and $\mathcal{R}_{gc}$ on $[P]$ as illustrated in Fig. 3 transforming the encoded process $[P]$ into some encoded process $[Q]$. Then, it was additionally shown that given two processes P and Q, the reduction $P \to Q$ exists iff $[P] \xRightarrow{\mathcal{R}_\pi}_*$ $[Q]$ exists.

The actual derivation graph obtained from computing every possible transition $[P] \xRightarrow{\mathcal{R}_\pi}_* [Q]$, involves many direct derivations, as is evident from Fig. 3, that does not correspond to any specific state of a process, but are necessary implementation details to transition from one process state to another.

Equipped with the derivations illustrated in Fig. 3, however, we can identify all direct derivations between actual graphs corresponding to processes. We can
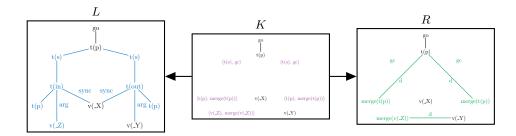
Figure 4: The rule for synchronization in $\mathcal{R}_\pi$ when channel and argument differ, represented as the span $L \leftarrow K \rightarrow R$. Terms starting with an underscore represents variables, e.g., the match in L is searching for a name "v(_X)", where the actual name is unspecified and contained in "_X". The variable can then be used in R. Matched vertices and edges coloured blue in L will be modified by the rule while vertices and edges coloured black will be in the context K of the rule. We see, e.g., that the vertex with the term "go" remains unchanged, while the vertex with the term "t(in)" in L is deleted in R. Vertices whose terms will be relabelled will be coloured purple in the context. We see that vertices with the term "t(s)" in L will be renamed to "gc" in R, represented in the context with "$\langle t(s), gc \rangle$". Finally, vertices marked for coalescing will be wrapped in a term called "merge" and connected to the vertex they should be coalesced into. This is seen by the renaming of the name "v(_Z)" to "merge(v(_Z))" and connecting it to "v(_Y)", representing the name "_Y" being bound to "_Z" in the simulated reduction.

then use such direct derivations to create a new abbreviated derivation graph, representing our execution space, where any sequence of derivations $[P] \overset{\mathcal{R}_\pi}{\Longrightarrow}_* [Q]$ is modelled as an edge. The result is an execution space as illustrated in Fig. 2, as first introduced in the start of this section.

5   A Final Example

Suppose we want to model the following behaviour:
- a patient stumbles into a hospital with two doctors; Dr. Jekyll and Mr. Hyde;
- the patient will be treated by the first doctor the patient comes across;
- Dr. Jekyll will cure the patient while Mr. Hyde will kill the patient.

The behaviour can be simulated with the expression $Hospital = P \mid J \mid H$, for the process ϕ defined by:

$$P =_\phi \overline{stumble}[name].name(d).P' \qquad P' =_\phi kill(x) + cure(x).P$$

$$J =_\phi stumble(n).\overline{n}[jekyll].\overline{cure}[j] \qquad H =_\phi stumble(n).\overline{n}[hyde].\overline{kill}[h]$$

We refer to the Companion Technical Report [6] for details about the encoding of recursive processes and how to compute the execution spaces including recursive processes.

The arguments are not given for the recursive processes, as they are just the set of free names of the respective process. A patient walks into the hospital and gives their name to either Jekyll or Hyde via the stumble channel. The doctor then sends his own name back to the patient as a greeting via the patients name as a channel (note, this serves no practical purpose for the model). Finally, the patient is either cured or killed, depending on the doctor received. If cured, *P* is called recursively simulating that the patient can get sick and stumble into the hospital again. If the patient died, we reached a deadlock as there is no cure for death. Note, many of the arguments sent over the channels are not used, but are simply there for illustrative purposes.

We can model the above example in `EpiM` as follows:

```
1   s, n, pn, j, h, cu, ki, d, x = names("s n pn j h cu ki d x")
2   free_names = [s, n, ki, cu]
3
4   rp = RecursiveProcess()
5
6   P = Process().output(s, n).input(n, d).call("Pp", free_names)
7   rp.add("P", free_names, P)
8
9   Pp = Process().input(ki, x) + Process().input(cu, x).call("P", free_names)
10  rp.add("Pp", free_names, Pp)
11
12  J = Process().input(s, pn).output(pn, j).output(cu, j).call("J", free_names)
13  rp.add("J", free_names, J)
14
15  H = Process().input(s, pn).output(pn, h).output(ki, h).call("H", free_names)
16  rp.add("H", free_names, H)
17
18  Hospital = P | J | H
19
20  exec_space = ReductionDG(Hospital, rp)
21  exec_space.calc()
22  exec_space.print()
```

Note, we have shortened the names to simple single letter names for illustrative purposes. The resulting execution space, representing the simulation of *Hospital*, is illustrated in Fig. 5. Although, it might be difficult to parse the graph, the expected behaviour should be clear from the graph: if the patient is treated by Dr. Jekyll, *P* is called recursively and as a result the behaviour is modelled as the only cycle in the derivation graph. If on the other hand the patient was killed off then *P* is terminated. Hence the behaviour is modelled in the graph as a path that end in a vertex with no outgoing edges (reached a deadlock).

6 Conclusion

We have introduced `EpiM`, available as a web front-end at `http://cheminf.imada.sdu.dk/epim`; a tool using graph transformation for computing execution spaces of π-calculus processes. In this regard we presented a brief overview
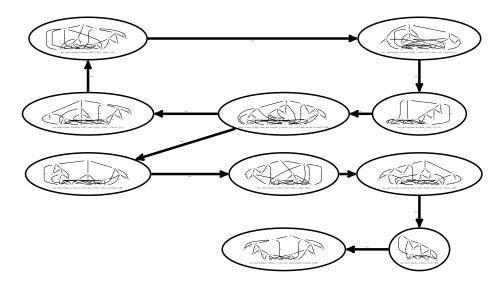
Figure 5: The execution space for $Hospital$. The start of the simulation, i.e., $[Hospital]$, is the single vertex with a degree of 3. The vertex for $[Hospital]$ contains two out-edges corresponding to the patient P either synchronizing on Dr. Jekyll J or Mr. Hyde H. If P synchronizes with J, we see that we end up in a cycle, representing P being cured and called recursively. On the other hand synchronizing on H, leads to a vertex with no out-edges, and hence no reductions are possible from the corresponding encoding, representing the patient being killed off and we are left with the encoded process $[H \,|\, J]$.

of the encodings and transformations involved. Practically, execution spaces can be used for basic analysis of processes such as determining liveness. Since `EpiM` is directly embedded into `MØD`, it would be possible to use the range of features provided by the `MØD` framework, notably [5]. Since structural congruence can be determined directly from graph isomorphism of the encoded processes, canonicalization of processes is given for free. `MØD` provides a framework for stochastic simulations [4], suggesting the possibilities of using a version of the encoding of π-calculus presented here to simulate stochastic π-calculus processes [9].

The type of DPO supported by `MØD` does not allow rewriting rules that perform bulk duplication or deletion of subgraphs. This means that duplicating or deleting processes cannot be implemented with a single rewrite operation. On one hand, this elicits the cost of process duplication or recursion often ignored in process calculi, on the other, it suggests to explore the use of `MØD` with "resource aware calculi" like linear variations of the π-calculus [8, 15, 16].

References

1. Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. Generic strategies for chemical space exploration. *Int. J. Comp. Bio. and Drug Design*, 2014.

2. Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. A software package for chemically inspired graph transformation. In *Graph Transformation - 9th International Conference, ICGT 2016, Proceedings*, volume 9761 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2016.

3. Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. Chemical transformation motifs — modelling pathways as integer hyperflows. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2018. in press, early access online.

4. Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. Rule-based gillespie simulation of chemical systems. In *3rd Workshop on Verification of Engineered Molecular Devices and Programs, VEMDP 2016, Proceedings*. Springer, 2018.

5. Jakob L. Andersen and Daniel Merkle. A generic framework for engineering graph canonization algorithms. In *2018 Proceedings of the 20th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 139–153, 2018.

6. Jakob Lykke Andersen, Marc Hellmuth, Daniel Merkle, Nikolai Nøjgaard, and Marco Peressotti. A Graph-Based Tool to Embed the π-Calculus into a Computational DPO Framework, 2019.

7. Anja Bog and Frank Puhlmann. A tool for the simulation of π-calculus systems. *Open. BPM*, 2006.

8. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

9. Luca Cardelli and Radu Mardare. Stochastic pi-calculus revisited. In *International Colloquium on Theoretical Aspects of Computing*, pages 1–21. Springer, 2013.

10. Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Rule-based modelling of cellular signalling. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 17–41. Springer, 2007.

11. Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 167–180. IEEE, 1973.

12. Fabio Gadducci. Graph rewriting for the π-calculus. *Mathematical Structures in Computer Science*, 17(3):407–437, 2007.

13. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

14. Leonard A. Harris, Justin S. Hogg, José Juan Tapia, John A. P. Sekar, Sanjana Gupta, Ilya Korsunsky, Arshi Arora, Dipak Barua, Robert P. Sheehan, and James R. Faeder. Bionetgen 2.2: advances in rule-based modeling. *Bioinformatics*, 32(21):3366–3368, 2016.

15. Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

16. Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: a fully-abstract semantics for classical processes. *PACMPL*, 3(POPL):24:1–24:29, 2019.

17. Barbara König. A graph rewriting semantics for the polyadic pi-calculus. In *Proc. of GT-VMT'00 (Workshop on Graph Transformation and Visual Modeling Techniques)*, pages 451–458, 2000.
18. Robin Milner. *Communicating and mobile systems - the Pi-calculus.* Cambridge University Press, 1999.
19. Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press, 2001.
20. Björn Victor and Faron Moller. The mobility workbench : A tool for the pi-calculus. In *Proceedings of CAV'94 :*, pages 428–440, 1994.