Efficient Computation of Graph Overlaps
for Rule Composition: Theory and Z3 Prototyping

Nicolas Behr*

Center for Research and Interdisciplinarity
Université de Paris, INSERM U1284
8-10 Rue Charles V, 75004 Paris, France

nicolas.behr@cri-paris.org

Reiko Heckel

School of Informatics
University of Leicester
LE1 7RH Leicester, UK

rh122@leicester.ac.uk

Maryam Ghaffari Saadat

School of Informatics
University of Leicester
LE1 7RH Leicester, UK

mgs17@leicester.ac.uk

Graph transformation theory relies upon the composition of rules to express the effects of sequences of rules. In practice, graphs are often subject to constraints, ruling out many candidates for composed rules. Focusing on the case of sesqui-pushout (SqPO) semantics, we develop a number of alternative strategies for computing compositions, each theoretically and with an implementation via the Python API of the Z3 theorem prover. The strategies comprise a straightforward generate-and-test strategy based on forbidden graph patterns, a variant with a more implicit logical encoding of the negative constraints, and a modular strategy, where the patterns are decomposed as forbidden relation patterns. For a toy model of polymer formation in organic chemistry, we compare the performance of the three strategies in terms of execution times and memory consumption.

1 Introduction

When applying graph transformation rules sequentially or in parallel, we are often interested in capturing their combined preconditions and effects in a single composed rule, e.g. to compare the effects of derivations during confluence analysis [22], to show the preservation of behaviour under different composition operations on rules [36, 39], or to define morphisms between systems that map individual to composed rules [29]. Usually, a notion of rule composition, such as *concurrent productions* in the algebraic approaches, relies on a specification of the relation or overlap of the given rules followed by a construction of the combined rule. A particular application scenario for such types of computations are stochastic graph rewrite systems (SRS) [8, 3, 5, 9], a categorical notion of stochastic rewrite systems such as Kappa [20, 18, 12], using an algebraic structure on rewrite steps to support the analysis of biological systems. Instead of stochastic simulation or model checking with their well-known scalability issues, SRS aim at the derivation of a system of evolution equations which, for a set of graph patterns as observables, predict their number of occurrences over time. For example in models of biochemistry [21, 28, 17], a pattern could represent a type of molecule, such that the number of occurrences of the pattern in a given graph that models the number of molecules of this type in a given cell. In the semantics of SRS based upon rule algebras [8, 3, 5, 9], the evolution equations for such average expected pattern counts are derivable via so-called *commutators*, which are computed by counting the distinct sequential compositions of rules representing *transitions* and rules representing *pattern observables*.

In many such application scenarios graphs are subject to structural constraints. This means that when composing rules, we have to ensure that any resulting composed rule respects these constraints, in the sense that an application of the composed rule reflects a sequential application of two given rules with a legal intermediate graph. This causes the algorithmic challenge of avoiding where possible the

---

construction of rule compositions forbidden by the constraints. Since a composition of two rules is defined based on a relation between the first rule's right-hand side and the second rule's left-hand side, we have to determine the set of all such relations that lead to composed rules respecting the structural constraints.

In this paper, we develop a number of alternative strategies for addressing this challenge, both theoretically and using the Z3 theorem prover with its Python interface. Specifically we tackle the computation of rule compositions in the Sesqui-Pushout (SqPO) approach [14], which provides the right level of generality for applications in biochemistry.

We first consider a *direct* or *"generate-and-test"* strategy, whereby in a "generation" phase all monic partial overlaps without taking into account the structural constraints are determined, and where in an ensuing "testing" phase each candidate overlap is scrutinised for whether or not its pushout object admits the embedding of any of the *forbidden graph patterns* used to express the structural constraints.

As an alternative, a more modular strategy is based on a decomposition of the aforementioned forbidden graph patterns as *forbidden relation patterns*, which may then be utilised to design a search algorithm for monic partial overlaps that combines the search and the constraint-checking into a single operation, and thus in principle permits to avoid constructing a potentially enormous number of candidate overlaps not respecting the structural constraints. Mathematically, this second option is based upon an original contribution of this paper in terms of the theory of constraints: while the verification of a negative constraint is traditionally formulated in terms of a non-embedding condition of a *forbidden pattern* into the target graph, for the special case that the target graph is the pushout of a monic span one may equivalently reformulate this verification task as a non-embedding condition of a *forbidden relation* into the span. We explicitly demonstrate the equivalence of the two approaches, i.e. given a graph pattern P and all its relational decompositions S, for any composed graph G over a span s there exists an embedding of P iff there exists a span in S with a triple of compatible embeddings into s.

Our third strategy for determining constraint-preserving rule overlaps is a variant of the "generate-and-test" strategy that is based upon a more *implicit* encoding of the condition that the pushout of a given candidate rule overlap should respect the structural constraints. Approaches to the implementation of analysis techniques on graph transformation systems can be divided into *native*, *translation-based*, and *hybrid* [31]. We follow a hybrid approach where all three strategies are implemented in Python using the Z3 SMT solver and theorem prover [37] for computing decompositions and embeddings. Our approach was inspired by an active research area in graph rewriting that features a number of other implementations of analysis techniques, e.g., for termination, confluence, or reachability as discussed in [31]. At the core of our implementation, we precisely mirror the category-theoretical structures encoding graphs, morphisms, overlaps, compositions of graphs and constraints in terms of a class architecture in Python and utilising the Z3 Python API. The declarative nature of SMT solving, where models are only determined up to isomorphism, is a good fit for the categorical theory of graph rewriting, allowing a very direct translation. The aim of this implementation is to evaluate the relative performance in terms of time and space requirements of our three strategies. As a case study, we consider an example of SqPO rewriting of rigid multigraphs representing polymers in organic chemistry. A multigraph (which, in general, allows parallel edges) is rigid if it does not contain edges of the same type in parallel, starting or ending in the same node, or as parallel loops (see definition of patterns in Example 2). Based on a simple set of rules for creating and deleting edges we can model polymer formation, such that the evolution equations derivable via the commutators computed could predict the number of polymers of a particular length and shape at any time of the reaction. The particular contribution of this paper is in developing the theory and experimental implementation of rule compositions with graph constraints following the three alternative strategies and a preliminary comparative evaluation of their scalability.

2   Rule Compositions for Conditional SqPO Rewriting

We provide some background on the recent extension of *Sesqui-Pushout (SqPO) rewriting* [14] to the setting of rules with application conditions [6] and describe the additional assumptions on the underlying category to admit a rule algebra construction [3, 7]. We refer the readers to [7] for the extended discussion of the mathematical concepts.

2.1   Categorical Setting

Assumption 1 ([6])**.** We assume that $\mathbf{C} \equiv (\mathbf{C}, \mathscr{M})$ is a finitary $\mathscr{M}$-adhesive category with $\mathscr{M}$-effective unions, $\mathscr{M}$-initial object, an epi-$\mathscr{M}$-factorisation, existence of final pullback complements (FPCs) for all pairs of composable $\mathscr{M}$-morphisms and with stability of $\mathscr{M}$-morphisms under FPCs.

While a full discussion of these technical concepts is out of the scope of the present paper, we will comment at various points on some of the salient ideas and uses of the various assumptions made. Suffice it here to note that $\mathscr{M}$ is a class of monomorphisms, and that we impose the finitarity-constraint for practical reasons, in that in our intended applications only *finite* structures and their transformations are of interest, with a prominent example given as follows:

Example 1. The prototypical category suited for rewriting in the above sense is the category **FinGraph** of *finite directed multigraphs*. It is well-known [34] that the category **Graph** of all (not necessarily finite) multigraphs is *adhesive* (i.e. $\mathscr{M}$-adhesive for $\mathscr{M} = \mathsf{mono}(\mathbf{Graph})$), and thus due to results of [26] so is its finitary restriction **FinGraph**. The finitary restriction preserves the epi-mono-factorisation, the property of mono-effective unions as well as the mono-initial object $\varnothing$ (the empty graph). Finally, according to [14], FPCs exist for arbitrary pairs of composable monos, and monos are stable under FPCs.

2.2   Conditions in SqPO Rewriting

For the readers' convenience, we recall some of the relevant background material and standard notations (including a convenient shorthand notation for simple, i.e. non-nested conditions). Conceptually, application conditions are defined such as to constrain the *matches* of rewriting rules. Another important special case are conditions over the $\mathscr{M}$-initial object $\varnothing$. Such *global constraints* describe properties of all objects, such as invariants.

Definition 1 (cf. e.g. [27, 23])**.** Given an $\mathscr{M}$-adhesive category $\mathbf{C}$ satisfying Assumption 1, (nested) *conditions* $\mathsf{cond}(\mathbf{C})$ over $\mathbf{C}$ are recursively defined as follows:

1. For all objects $X \in \mathsf{obj}(\mathbf{C})$, true_X is a condition.

2. For every $\mathscr{M}$-morphism $(f : X \hookrightarrow Y) \in \mathscr{M}$ and for every condition $\mathsf{c}_Y \in \mathsf{cond}(\mathbf{C})$ over Y, $\exists(f, \mathsf{c}_Y)$ is a condition.

3. If $\mathsf{c}_X \in \mathsf{cond}(\mathbf{C})$ is a condition over X, so is $\neg \mathsf{c}_X$.

4. If $\mathsf{c}_X^{(1)}, \mathsf{c}_X^{(2)} \in \mathsf{cond}(\mathbf{C})$ are conditions over X, so is $\mathsf{c}_X^{(1)} \wedge \mathsf{c}_X^{(2)}$.

The *satisfaction* of a condition c_X by an $\mathscr{M}$-morphism $(h : X \hookrightarrow Z) \in \mathscr{M}$ is recursively defined as follows:

1. $h \vDash \mathsf{true}_X$.

2. $h \vDash \exists(f : X \hookrightarrow Y, \mathsf{c}_Y)$ iff there exists $(g : Y \hookrightarrow Z) \in \mathscr{M}$ such that $h = g \circ f$ and $g \vDash \mathsf{c}_Y$.

3. $h \vDash \neg \mathsf{c}_X$ iff $h \neg \vDash \mathsf{c}_X$.

4. $h \vDash c_X^{(1)} \wedge c_X^{(2)}$ iff $h \vDash c_X^{(1)}$ and $h \vDash c_X^{(2)}$.

Two conditions $c_X, c_X' \in \mathsf{cond}(\mathbf{C})$ are *equivalent*, denoted $c_X \equiv c_X'$, iff for every $(h : X \hookrightarrow Z) \in \mathscr{M}$ we find that $h \vDash c_X$ implies $h \vDash c_X'$ and vice versa.

We will employ the following standard shorthand notations:

$$\exists(f : X \hookrightarrow Y) := \exists(f : X \hookrightarrow Y, \mathsf{true}_Y), \quad \forall(f : X \hookrightarrow Y, c_Y) := \neg\exists(f : X \hookrightarrow Y, \neg c_Y).$$

An auxiliary construction [27] permits to extend conditions into a larger context.

Theorem 1. *With notations as above and for* $(f : X \hookrightarrow Y) \in \mathscr{M}$, *there exists a* shift construction Shift *such that*

$$\begin{aligned}
&\forall c_X \in \mathsf{cond}(\mathbf{C}), \forall(h : X \hookrightarrow Z) \in \mathscr{M} : \exists(g : Y \hookrightarrow Z) \in \mathscr{M} : h = g \circ f \\
&\Rightarrow \quad (h \vDash c_X \Leftrightarrow g \vDash \mathsf{Shift}(f, c_X)).
\end{aligned} \tag{1}$$

For the computation of SqPO rule compositions for rules with conditions, we need an additional auxiliary construction for the calculus of conditions:

Theorem 2 ([6], Thm. 7). *Given a linear rule* $r = (O \hookleftarrow K \hookrightarrow I) \in \mathsf{Lin}(\mathbf{C})$ *and a condition* $c_O \in \mathsf{cond}(\mathbf{C})$ *over O, there exists a* transport construction Trans *such that for any object* $X \in \mathsf{obj}(\mathbf{C})$ *and for any SqPO-admissible match* $m \in \mathsf{M}_r^{sq}(X)$ *of r into X, if* $(m^* : O \hookrightarrow r_m(X)) \in \mathscr{M}$ *denotes the comatch of m, the following holds:*

$$m^* \vDash c_O \quad \Leftrightarrow \quad m \vDash \mathsf{Trans}(r, c_O). \tag{2}$$

While an interesting set of theoretical results in its own right, the concrete implementation of the Shift and Trans constructions are not of relevance to the present paper, so we refer the interested readers to [6, Thms. 4 and 7] for the precise details and further technical discussions.

2.3 Rigidity

Example 2. Our running example throughout this paper will be the category **rGraph** of *finite rigid directed multigraphs*. Referring to [19] for an extended discussion of the rigidity phenomenon, suffice it here to introduce this category as a refinement of the category **FinGraph** via imposing the following *global constraint* formulated via a set $\mathscr{N}$ of *forbidden patterns*:

$$c_{\mathscr{N}} := \bigwedge_{N \in \mathscr{N}} \neg\exists(\varnothing \hookrightarrow N), \quad \mathscr{N} := \{\ \text{⟳} , \text{⟷}, \text{•→•←•}, \text{•←•→•}\}. \tag{3}$$

Connected components of graphs in **rGraph** are thus either individual vertices, "directed paths" π_n of edges (with lengths $n \geq 1$) or closed "directed loops" λ_n of edges (with lengths $n \geq 2$),

$$\pi_1 = \text{•→•}, \qquad \pi_2 = \text{•→•→•}, \qquad \pi_3 = \text{•→•→•→•}, \qquad \pi_4 = \text{•→•→•→•→•}, \ \ldots$$

$$\lambda_2 = \text{⟷}, \qquad \lambda_3 = \text{▷}, \qquad \lambda_4 = \text{□}, \ \ldots$$

possibly in addition with individual loops on vertices, albeit we will only consider loop-less graphs in our applications. It is instructive to consider for this "self-loop-less" special case of graphs in **rGraph** the numbers of isomorphism classes as a function of the number of vertices. Following standard combinatorial arguments along the lines of [25], we can construct an ordinary generating function for these numbers by (1) dressing each vertex with a power of some formal variable x, each edge in a "path" with y

and each edge in a "directed loop" with z, and (2) multiplying the OGFs for copies of the different types of connected components:

$$\mathscr{G}(x,y,z) := \sum_{k,\ell,m \geq 0} x^k y^\ell z^m \cdot (\text{\# of iso-classes with } k \text{ vertices and } \ell + m \text{ edges})$$

$$= \frac{1}{1-x} \prod_{p \geq 1} \frac{1}{1-x^{p+1}y^p} \prod_{q \geq 2} \frac{1}{1-(xz)^q} \xrightarrow{y,z \to 1} \frac{1-x}{((x;x)_\infty)^2} \, .$$

The expansion of $\mathscr{G}(x,1,1)$ (i.e. of the OGF of iso-classes counted by number of vertices) permits to identify this series as the OGF of the integer sequence A000990 on the OEIS database[1], with sequence entries (for $n = 0, \ldots, 10, \ldots$)

$$1, 1, 3, 5, 10, 16, 29, 45, 75, 115, 181, \ldots \tag{4}$$

Comparing this to the sequence A000273 on the OEIS database, which gives the number of isomorphism classes of directed simple graphs without self-loops counted by number of vertices, and which reads for $n = 0, \ldots, 10$

$$1, 1, 3, 16, 218, 9608, 1540944, 882033440, 1793359192848,$$
$$13027956824399552, 341260431952972580352, \ldots, \tag{5}$$

it transpires that the rigid directed graphs are enormously constrained in their structure compared to general simple directed graphs. This fact will in turn have a strong impact on the complexity of computing compositions of transformation rules based upon rigid graphs, in that any computation strategy based upon a "generate-and-test" approach will suffer from the enormous number of candidate rule overlaps. The way rigid graphs are restricted in their structure as compared to generic simple graphs is furthermore quite analogous to the way e.g. molecular graphs in the rewriting theory for organic chemistry [10, 1] or for biochemistry [20, 17, 12] are restricted as compared to generic (typed) simple graphs, which is why we take **rGraph** as a prototypical example of such types of applications of rewriting theory.

2.4 SqPO Direct Derivations and Rule Compositions

The theory of "compositional" SqPO rewriting as introduced in [3, 6, 7] is an extension of the traditional SqPO theory [14] by concurrency and associativity theorems that hold under suitable assumptions on the underlying categories.

Remark 1. Contrary to standard conventions we will read rewriting rules from **input** to **output**, i.e. in particular from **right to left**. This is so as to be compatible with the standard mathematical convention of left-multiplication for matrices (see e.g. [5] for an extended discussion).

Definition 2. Let **C** be an $\mathscr{M}$-adhesive category satisfying Assumption 1, and let $\overline{\mathsf{Lin}}(\mathbf{C})$ denote the class of *linear rewriting rules with (nested) conditions*,

$$\overline{\mathsf{Lin}}(\mathbf{C}) := \{(O \hookleftarrow K \hookrightarrow I; \mathsf{c}_I) \mid (K \hookrightarrow O), (K \hookrightarrow I) \in \mathscr{M}, \mathsf{c}_I \in \mathsf{cond}(\mathbf{C})\}. \tag{6}$$

Let $\overline{\mathsf{Lin}}(\mathbf{C})_\sim$ be the class of *equivalence classes of linear rules with conditions* under the equivalence relation $\sim$ defined as follows:

$$(r, \mathsf{c}_I) \sim (r', \mathsf{c}_I') :\Leftrightarrow r \cong r' \wedge \mathsf{c}_I \equiv \mathsf{c}_I'. \tag{7}$$

Here, $r \cong r'$ if and only if there exist isomorphisms $(\omega : O \to O')$, $(\kappa : K \to K')$ and $(\iota : I \to I') \in \mathsf{iso}(\mathbf{C})$ such that the evident diagram commutes.

---

[1]N. J. A. Sloane, The On-Line Encyclopedia of Integer Sequences (https://oeis.org)

The following definition provides a notion of direct derivations for SqPO rules with conditions.

Definition 3 ([6], Def. 17; compare [14], Def. 4)**.** Given an object $X \in \mathsf{obj}(\mathbf{C})$ and a linear rule with condition $R = (r, \mathsf{c}_I) \in \overline{\mathsf{Lin}}(\mathbf{C})$, we define the *set of admissible matches* $\mathsf{M}_R^{\mathsf{sq}}(X)$ as

$$\mathsf{M}_R^{\mathsf{sq}}(X) := \{(m : I \to X) \in \mathscr{M} \mid m \vDash \mathsf{c}_I\}. \tag{8}$$

A *direct derivation of X along R with match* $m \in \mathsf{M}_R^{\mathsf{sq}}(X)$ is defined via constructing the diagram on the right, with final pullback complement marked FPC and pushout marked PO. We write $R_m(X) := X'$ for the object "produced" by the above diagram, and we refer to m^* as the *comatch of m*.

$$
\begin{array}{ccccc}
O & \longleftarrow & K & \longrightarrow & I \\
m^* \downarrow & \text{PO} & k \downarrow & \text{FPC} & \downarrow m \\
X' & \longleftarrow & \overline{K} & \longrightarrow & X
\end{array} \tag{9}
$$

As is well-known in the graph-rewriting community, the notion of FPC along $\mathscr{M}$-morphisms has a very natural interpretation in the setting of **C** being some form of graph-based category, e.g. **C** = **FinGraph** (for which $\mathscr{M}$ is the class of monomorphisms of finite graphs): considering a linear rule $(\varnothing \hookleftarrow \varnothing \hookrightarrow \bullet)$ encoding (read from right to left) the deletion of a vertex, choosing a match (i.e. an embedding) of the input vertex of the rule into some finite graph X at a vertex with incident edges and forming the FPC effectively results in an intermediate graph $\overline{K}$ that is obtained from X by removing the matched vertex and all incident edges. In contrast, a *pushout complement (POC)* for the same rule and match to a vertex in X with incident edges would not exist, highlighting a characteristic feature in *Double-Pushout (DPO)* rewriting in that edges may not be "implicitly" deleted.

The second main definition of SqPO-type "compositional" rewriting theory is given by a notion of sequential composition for rules with application conditions.

Definition 4 ([6])**.** With notations as above, let $R_j = (r_j, \mathsf{c}_{I_j}) \in \overline{\mathsf{Lin}}(\mathbf{C})_\sim$ be two equivalence classes of linear rules with conditions ($j = 1, 2$). Fixing representatives $(O_j \hookleftarrow K_j \hookrightarrow I_j; \mathsf{c}_{I_j})$ ($j = 1, 2$), a monic span $\mu = (I_2 \hookleftarrow M_{21} \hookrightarrow O_1)$ is defined to be an *SqPO-admissible match of R_2 into R_1*, denoted $\mu \in \mathsf{M}_{R_2}^{\mathsf{sq}}(R_1)$, if the pushout complement marked POC in the diagram below exists[2],

$$\tag{10}$$

and if in addition $\mathsf{c}_{I_{21}} \not\equiv \mathsf{false}$, where

$$\mathsf{c}_{I_{21}} = \mathsf{Shift}(I_1 \hookrightarrow I_{21}, \mathsf{c}_{I_1}) \wedge \mathsf{Trans}(N_{21} \hookleftarrow K_1' \hookrightarrow I_{21}, \mathsf{Shift}(I_2 \hookrightarrow N_{21}, \mathsf{c}_{I_2})).$$

---

[2]Note that if the diagram is constructable, it follows from the properties of $\mathscr{M}$-adhesive categories and from further properties stated in Assumption 1 that as indicated in the diagram all arrows are $\mathscr{M}$-morphisms. More precisely, the technical properties utilised to prove this claim are the stability of $\mathscr{M}$-morphisms under pushouts and pullbacks [23] as well as the stability of $\mathscr{M}$-morphisms under FPCs. The proof that this type of composition indeed leads to a suitable notion of concurrency theorem hinges also upon $\mathscr{M}$-effective unions and the existence of an $\mathscr{M}$-initial object [3, 6]. See Section 2.2, Theorems 1 and 2 for notations and further details.

In this case, we define the *SqPO-type composition of R_2 with R_1 along μ_{21}* as the following equivalence class:

$$R_2 \stackrel{\mu}{\triangleleft} R_1 := [(O_{21} \hookleftarrow K_{21} \hookrightarrow I_{21}; c_{I_{21}})]_\sim. \tag{11}$$

Example 3 (Ex. 2 continued). Seeing that rigid graphs as described in Example 2 may be considered as a toy model of *polymers* that can form "directed chains" and "directed loops", one might ask how one could construct a stochastic dynamical model over **rGaph**. Referring the interested readers to [3, 5, 7] for a more in-depth discussion and the precise theoretical setup, for the present paper we focus on two types of contributions to a stochastic rewriting theory setup:

1. **Transitions** are modelled via rewriting rules with conditions $R_j = (r_j, c_{I_j}) \in \overline{\mathsf{Lin}}(\mathbf{rGraph})_\sim$, one for each transition of the model. Choosing *base rates* $\kappa_j \in \mathbb{R}_{\geq 0}$ (i.e. rates of "firings per second") for each transition, the so-called "mass-action semantics" [7] would render the likelihood of a given transition "firing" in an infinitesimal step proportional to its base rate times the number of matches of the rule into the current state.

2. **Observables** implementing the detection and counting of *patterns* are based upon rules of the special form [7]

$$R_{P, c_P} = [(P \stackrel{id_P}{\longleftrightarrow} P \stackrel{id_P}{\longrightarrow} P; c_P)]_\sim, \tag{12}$$

   where $P \in \mathsf{obj}(\mathbf{C})_\cong$ is the "pattern", and where the condition $c_P \in \mathsf{cond}(\mathbf{C})$ in effect permits to implement pattern counting e.g. of the sort "two vertices not linked by an edge".

For our polymer example, one could start the construction of a stochastic model by considering a set of transitions that in effect are capable of rendering arbitrary finite rigid graphs through repeated "firings" of the transitions, i.e. the vertex deletion/creation and edge deletion/creation rules depicted in Figure 1**(a)**. Note that the application conditions necessary for these rules to respect the rigid graph constraints may be derived e.g. according to the algorithm presented in Corollary 1 (Section 3). In order to model certain physical effects such as saturation and steric hindrance effects, one might also consider to include more sophisticated transitions involving larger input and output graphs, such as the ones depicted in Figure 1**(b-d)**. Typical observables for the rule-sets presented would then e.g. be constructed in the form $R_{I_j, c_{I_j}}$, i.e. from the input patterns and conditions (since these observables feature in the evolution equations of the respective stochastic rewriting systems, cf. [7]).

3 Constraint-checking Strategies in Rule Compositions

When computing SqPO-type compositions of conditional rules, an algorithmically expensive step consists in verifying the satisfaction of both the global and application conditions in a given overlap, followed by computing the derived application conditions of the admissible composites. Both in order to experiment with the implementations of such algorithms via Z3 (see next Section) and out of a theoretical interest, we consider different implementation strategies for the steps involved in this rule composition operation. We will present here the "direct" strategy (i.e. following precisely the traditional constructions involving Shift and Trans) as well as an alternative strategy based upon certain span (non-) embedding criteria. From this section onwards, we will fix a category $\mathbf{C}$ satisfying Assumption 1, and assume a set of global conditions $c_\mathcal{N}$ as in (2) on objects. Let us further assume that all rules are endowed with application conditions that ensure the preservation of $c_\mathcal{N}$, and that the rules themselves are constructed of objects satisfying the constraint $c_\mathcal{N}$.

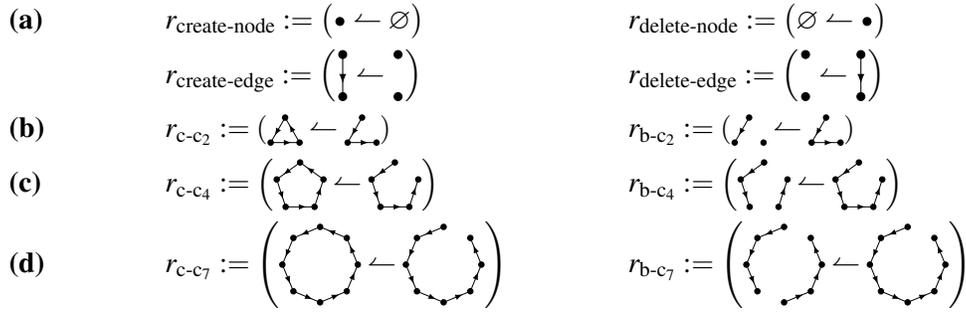**(a)** $r_{\text{create-node}} := \left(\bullet \leftharpoonup \varnothing \right)$ $\qquad$ $r_{\text{delete-node}} := \left(\varnothing \leftharpoonup \bullet \right)$

$r_{\text{create-edge}} := \left(\begin{smallmatrix} \bullet \\ \bullet \end{smallmatrix} \leftharpoonup \begin{smallmatrix} \bullet \\ \bullet \end{smallmatrix} \right)$ $\qquad$ $r_{\text{delete-edge}} := \left(\begin{smallmatrix} \bullet \\ \bullet \end{smallmatrix} \leftharpoonup \begin{smallmatrix} \bullet \\ \bullet \end{smallmatrix} \right)$

(b) $r_{\text{c-}c_2} := \left(\triangle \leftharpoonup \angle \right)$ $\qquad$ $r_{\text{b-}c_2} := \left(\diagup \leftharpoonup \angle \right)$

(c) $r_{\text{c-}c_4} := \left(\bigcirc \leftharpoonup \bigcirc \right)$ $\qquad$ $r_{\text{b-}c_4} := \left(\bigcirc \leftharpoonup \bigcirc \right)$

(d) $r_{\text{c-}c_7} := \left(\bigcirc \leftharpoonup \bigcirc \right)$ $\qquad$ $r_{\text{b-}c_7} := \left(\bigcirc \leftharpoonup \bigcirc \right)$

Figure 1: Examples of transitions $R_j = (r_j, c_{I_j}) \in \overline{\mathsf{Lin}}(\mathbf{rGraph})_\sim$ for rigid graph stochastic rewriting models: **(a)** the "running example" as a minimal setup, and **(b-d)** additional transitions involving larger graph patterns. Concretely, these rules model the two alternative options of closing a "chain" of edges into a loop ("<u>c</u>reate-<u>c</u>ycle"), or breaking up a chain of edges ("<u>b</u>reak-<u>c</u>hain"). The conditions c_{I_j} (not explicitly depicted) are defined as the respective constraint-preserving application conditions computed according to Corollary 1.

Definition 5 (Direct Strategy)**.** Given two linear rules with conditions $R_j \equiv (r_j, c_{I_j}) \in \overline{\mathsf{Lin}}(\mathbf{C})$ $(j = 1, 2)$ and a candidate match $\mu = (I_2 \hookleftarrow M_{21} \hookrightarrow O_1)$, the *Direct Strategy* to verify whether μ is an admissible match is defined as follows:

1. Construct the pushout $(I_2 \hookrightarrow N_{21} \hookleftarrow O_1)$ of μ. Verify that $N_{21} \models c_{\mathcal{N}}$, and that $(I_2 \hookrightarrow N_{21}) \models c_{I_2}$.

2. If the pushout complement of $(N_{21} \hookleftarrow O_1, O_1 \hookleftarrow K_1)$ exists, perform the SqPO-type composition according to Definition 4, verifying that the application condition of the composite rule satisfies $c_{I_{21}} \not\equiv \mathsf{false}$.

If both steps are successful, μ is an SqPO-admissible match of R_2 into R_1.

As an alternative strategy, let us restate the "forbidden patterns" $N \in \mathcal{N}$ via their *pushout decompositions*, defined as follows:

Definition 6. We define the *set of "forbidden relations"* $\mathscr{S}_{\mathcal{N}}$ as[3]

$$\mathscr{S}_{\mathcal{N}} := \left\{ s = (C_1 \hookleftarrow D \hookrightarrow C_2) \mid C_1, D, C_2 \models c_{\mathcal{N}} \wedge \exists N \in \mathcal{N} : \mathsf{PO}(s) \cong N \right\}. \tag{13}$$

Example 4. For the category **rGraph** as introduced in Ex. 2, one may compute the set of "forbidden relations" (with colours encoding the respective embeddings) as presented in Figure 2.

The pushout decompositions of forbidden patterns allow for a modular strategy to testing admissibility of rule overlaps that does not require to find embeddings of patterns into the pushout of the overlap, but *double-pullback embeddings (DPEs)* of forbidden spans into the monic spans representing the overlaps.

---

[3]As a further optimisation (tacitly employed from hereon and in our implementation), one may reduce the size of $\mathscr{S}_{\mathcal{N}}$ by only retaining one representative per isomorphism class of forbidden relations, with the same notion of isomorphism as the one utilised for linear rules (cf. Definition 2).

$$\mathscr{S}_{\mathscr{N}} = \left\{ \; \cdots , \cdots , \cdots , \cdots , \right.$$

$$\cdots , \cdots , \cdots , \cdots ,$$

$$\left. \cdots , \cdots \right\}$$
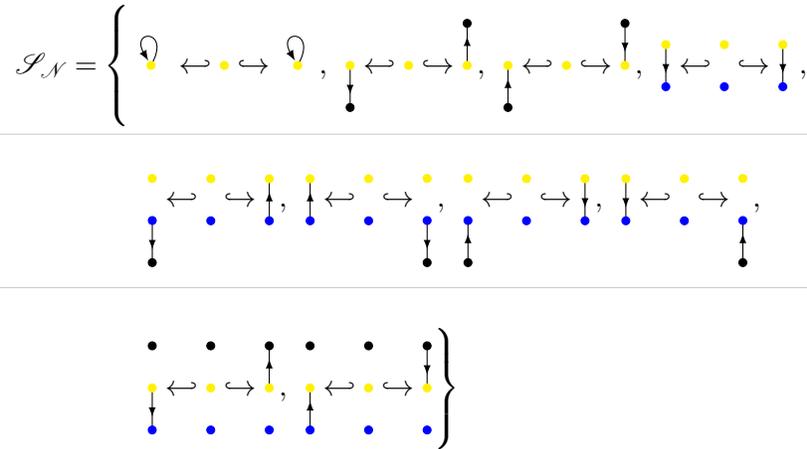
Figure 2: Set of "forbidden relations" for the category **rGraph**.

Theorem 3. *With notations and assumptions as above, given a pushout P of a monic span $(I_1 \hookleftarrow M_{21} \hookrightarrow O_1)$, the violation of $\mathsf{c}_{\mathscr{N}}$ is equivalently verified as*

$$
\begin{aligned}
P \not\Vdash \mathsf{c}_{\mathscr{N}} \;\Leftrightarrow\; &\exists s = (C_2 \hookleftarrow D \hookrightarrow C_1) \in \mathscr{S}_{\mathscr{N}} : \\
&\exists (C_2 \hookrightarrow I_2), (D \hookrightarrow M_{21}), (C_1 \hookrightarrow O_1) \in \mathsf{mono}(\mathbf{C}) : \\
&(C_2 \hookleftarrow D \hookrightarrow M_{21}) = \mathsf{PB}(C_2 \hookrightarrow I_2 \hookleftarrow M_{21}) \\
&\wedge \; (C_2 \hookleftarrow D \hookrightarrow M_{21}) = \mathsf{PB}(C_2 \hookrightarrow I_2 \hookleftarrow M_{21}).
\end{aligned}
\tag{14}
$$

Here, each DPE mentioned in (14) encodes a commutative diagram of the form

$$
\begin{array}{ccccc}
C_2 & \longleftarrow & D & \longrightarrow & C_1 \\
\downarrow & \text{PB} & \downarrow & \text{PB} & \downarrow \\
I_2 & \longleftarrow & M_{21} & \longrightarrow & O_1
\end{array}
\tag{15}
$$

Proof. The statement of the theorem follows from the $\mathscr{M}$-vanKampen property of the category **C**.
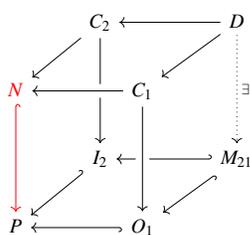
For the $\Rightarrow$ direction, suppose that $P \not\Vdash \mathsf{c}_{\mathscr{N}}$, which entails that there exists an $N \in \mathscr{N}$ and an embedding $(N \hookrightarrow P)$. Construct the commutative cube on the left via (1) taking pullbacks in order to obtain objects C_2 and C_2 and (2) letting D be defined as the pullback of $(C_2 \hookrightarrow N \hookleftarrow C_1)$. By stability of $\mathscr{M}$ morphisms and by their decomposition properties, all arrows in the top square and all vertical arrows are $\mathscr{M}$-morphisms. Since the bottom square is a pushout along $\mathscr{M}$-morphisms and thus a pullback, and since the front and top squares are pullbacks, by pullback-pullback decomposition the back square is a pullback, and analogously so is the right square. Thus by the $\mathscr{M}$-vanKampen property, the top square is a pushout, and we have proved that $(C_2 \hookleftarrow D \hookrightarrow C_1)$ is in $\mathscr{S}_{\mathscr{N}}$.

For the $\Leftarrow$ direction, suppose the bottom pushout square as well as the back and left pullback squares were given (with all involved morphisms in $\mathscr{M}$), and such that $(C_2 \hookleftarrow D \hookrightarrow C_1) \in \mathscr{S}_{\mathscr{N}}$. Then letting

N be the pushout of $(C_2 \hookleftarrow D \hookrightarrow C_1)$ (which by definition of $\mathscr{S}_\mathscr{N}$ entails that $N \in \mathscr{N}$), there exists by universal property of the pushout a morphism $(N \hookrightarrow P)$.

$$
\begin{array}{ccccc}
C_2 & \longleftarrow & D & == & D \\
\downarrow & (1) & \downarrow & (2) & \| \\
I_2 & \longleftarrow & M_{21} & \longleftarrow & D \\
\downarrow & (3) & \downarrow & (4) & \downarrow \\
P & \longleftarrow & O_1 & \longleftarrow & C_1
\end{array}
$$

It remains to demonstrate that $(N \to P)$ is in $\mathscr{M}$. To this end, let us assemble the auxiliary on the left. By assumption, squares (1) and (4) are pullbacks, square (3) a pushout along $\mathscr{M}$-morphisms (and thus a pullback), while squares of the form (2) are pullbacks by universal category theory. Consequently, we find by composition of pullbacks that D is the pullback of $(C_2 \hookrightarrow P \hookleftarrow C_1)$. Thus finally by virtue of the assumption of $\mathscr{M}$-effective unions, we can confirm that $(N \hookrightarrow P)$ is in $\mathscr{M}$.

□

The alternative test for constraint satisfaction via DPEs of spans of $\mathscr{S}_\mathscr{N}$ according to Theorem 3 permits to formulate the following alternative SqPO-type rule composition strategy:

Definition 7 (DPE Strategy). Given two linear rules with conditions $R_j \equiv (r_j, c_{I_j}) \in \overline{\mathsf{Lin}}(\mathbf{C})$ $(j = 1, 2)$ and a candidate match $\mu = (I_2 \hookleftarrow M_{21} \hookrightarrow O_1)$, the *DPE Strategy* to verify whether μ is an admissible match is defined as follows:

1. Verify that there does not exist any double-pullback embedding of a span of $\mathscr{S}_\mathscr{N}$ into the span μ (with DPEs as defined in (15)).

2. Verify that $(I_2 \hookrightarrow N_{21}) \models c_{I_2}$.

3. If the pushout complement of $(N_{21} \hookleftarrow O_1, O_1 \hookleftarrow K_1)$ exists, perform the SqPO-type composition according to Definition 4, verifying that the application condition $c_{I_{21}}$ of the composite rule satisfies $c_{I_{21}} \not\equiv \mathsf{false}$.

If all steps are successful, μ is an admissible SqPO-type match of R_2 into R_1.

Finally, a useful corollary of Theorem 3 is the following alternative algorithm for minimal constraint-preserving application conditions for SqPO-type rules (for the case as assumed in this section that rules themselves are constructed from objects satisfying the global constraint):

Corollary 1. *With notations as above, given a "plain" linear rule $r = (O \hookleftarrow K \hookrightarrow I) \in \mathsf{Lin}(\mathbf{C})$, perform the following steps: for each $(C_2 \hookleftarrow D \hookrightarrow C_1) \in \mathscr{S}_\mathscr{N}$,*

1. *find all pullback embeddings of $(C_2 \hookleftarrow D)$ into $(O \hookleftarrow K)$, i.e. pairs of $\mathscr{M}$-morphisms $(C_2 \hookrightarrow O)$ and $(D \hookrightarrow K)$ s.th. $D = \mathsf{PB}(C_2 \hookrightarrow D \hookleftarrow K)$, and*

2. *for each pullback embedding, construct $(C_1 \hookrightarrow P \hookleftarrow I)$ by taking the pushout of the span $(C_1 \hookleftarrow D \hookrightarrow K \hookrightarrow I)$; if $P \models c_\mathscr{N}$, then this pullback embedding contributes a negative condition of the form $\neg \exists (I \hookrightarrow P, \mathsf{true})$.*

The (minimal) constraint-preserving application condition c_I is given by the conjunction over all individual contributions $\neg \exists (I \hookrightarrow P, \mathsf{true})$ computed above.

Proof. For the $\Rightarrow$ direction, let us assume that a given SqPO-type direct derivation along rule $(O \hookleftarrow K \hookrightarrow I)$ with candidate match $(m : I \hookrightarrow X_0) \in \mathscr{M}$ results in an object X_1 with $X_1 \not\models c_\mathscr{N}$. By definition of satisfiability and of $c_\mathscr{N}$, this entails that there exists an $N \in \mathscr{N}$ and an $\mathscr{M}$-morphism $(N \hookrightarrow X_1) \in \mathscr{M}$. In

complete analogy to the proof of Theorem 3 as given in the previous section, construct the commutative cube below over the SqPO-type direct derivation diagram:

$$(16)$$

Here, C_1 and C_2 are constructed by taking pullbacks, which by stability of $\mathcal{M}$-morphisms under pullbacks entails that the newly constructed morphism are also in $\mathcal{M}$. D is constructed as a pullback of $(C_2 \hookrightarrow N \hookleftarrow C_1)$ (with $\mathcal{M}$-morphisms in the resulting span), while the morphism $(D \to K)$ is an $\mathcal{M}$-morphism by the decomposition property of $\mathcal{M}$-morphisms. Since the bottom left square is a pushout along $\mathcal{M}$-morphisms and thus a pullback, we find via pullback-pullback decomposition that also the squares $\square_{D,C_2,O,K}$ (back left) and $\square_{D,C_1,\overline{X}_0,K}$ (middle vertical) are pullbacks, and thus by the $\mathcal{M}$-VK property the top left square is a pushout. Next, construct the following three pushouts:

$$\overline{D} := \mathsf{PO}(C_1 \hookleftarrow D \hookrightarrow K), \overline{C}_2 := \mathsf{PO}(O \hookleftarrow K \hookrightarrow \overline{D}), P := \mathsf{PO}(\overline{D} \hookleftarrow K \hookrightarrow I).$$

As depicted in the diagram below left, by the universal properties of the relevant pushouts and via $\mathcal{M}$-effective unions, there exist morphisms (drawn below with dotted lines) that are in fact $\mathcal{M}$-morphisms:

$$(17)$$

Finally, there are two cases to consider: if $P \models \mathsf{c}_{\mathcal{N}}$, we have exhibited a $\mathcal{M}$-morphism $(I \hookrightarrow P)$ through which $(I \hookrightarrow X_0)$ factors, and which thus by the above construction proves that the rewrite will lead to an X_1 with at least one embedding of a "forbidden pattern" $N \in \mathcal{N}$. If on the other hand $P \not\models \mathsf{c}_{\mathcal{N}}$, we have proved that $X_0 \not\models \mathsf{c}_{\mathcal{N}}$; consequently, the $\mathcal{M}$-morphism $(I \hookrightarrow P)$ would in this case not contribute to a constraint-preserving application condition for $(O \hookleftarrow K \hookrightarrow I)$.

For the $\Leftarrow$ direction, let us assume we were given the data of the diagram below (i.e. a SqPO-type direct derivation of X_0 along candidate match $(I \hookrightarrow X_0) \in \mathcal{M}$ as well as a pattern P such that $(I \hookrightarrow X_0)$ factors through some $(I \hookrightarrow P)$ constructed according to the statement of the Corollary) as depicted in the diagram on the right of (17). Here, in order to demonstrate the existence of the morphism $(\overline{D} \to \overline{X}_0)$, let us first introduce a useful auxiliary formula: given a commutative diagram of $\mathcal{M}$-morphisms as below

left,

$$
\begin{array}{ccc}
A \longrightarrow B \searrow & & A \longrightarrow B === B \\
\downarrow \ \text{PO} \ \downarrow \ \searrow & \rightsquigarrow & \downarrow \ \text{PO} \ \downarrow \ \downarrow \\
C \longrightarrow D \longrightarrow E & & C \longrightarrow D \longrightarrow E
\end{array}
\tag{18}
$$

this data yields the diagram above right. Since the right square is a pullback and the left square a pushout along $\mathscr{M}$-morphisms and thus also a pullback, we find by composition of pullbacks that $\square_{A,C,E,B}$ is a pullback.

Back to the diagram on the right of (17), since $\square_{K,\overline{X}_0,X_0,I}$ is by assumption a final pullback complement and $\square_{K,\overline{D},X_0,I}$ is a pullback by virtue of the auxiliary formula, the universal property of FPCs entails the existence of $(\overline{D} \to \overline{X}_0)$, which by the decomposition property of $\mathscr{M}$-morphisms is an $\mathscr{M}$-morphism. Moreover, by FPC-pushout decomposition [6], $\square_{\overline{D},\overline{X}_0,X_0,P}$ is an FPC.

It then suffices to construct the pushout $\overline{C}_2 := \mathsf{PO}(O \hookleftarrow K \hookrightarrow \overline{D})$, which through the universal properties of the various pushouts involved yields the existence of morphisms $(N \to \overline{C})$ and $(\overline{C} \to X_1)$ (resulting in a diagram of the form in (17). Utilising pushout-pushout decompositions (yielding that $\square_{C_2,N,\overline{C}_2,O}$ and $\square_{\overline{D},\overline{C}_2,X_1,\overline{X}_0}$ are pushouts), stability of $\mathscr{M}$-morphisms under pushouts, the above auxiliary formula (i.e. to demonstrate that $\square_{C_2,N,X_1,O}$ and $\square_{D,C_1,X_0,K}$ are pullbacks) and the property of $\mathscr{M}$-effective unions, we find that $(N \to \overline{C}_2) \in \mathscr{M}$ and $(\overline{C}_2 \to X_1) \in \mathscr{M}$, such that we have in summary exhibited a $\mathscr{M}$-morphism $(N \hookrightarrow X_1) \in \mathscr{M}$, which verifies that $X_1 \not\vDash \mathsf{c}_{\mathscr{N}}$. $\qquad\square$

A heuristic solution for the problem of computing minimal constraint-preserving application conditions has been proposed in [38]. For the case of forbidden graph patterns $\mathscr{N}$ and negative conditions $\mathsf{c}_{\mathscr{N}}$, our result permits to avoid the use of the computationally expensive Shift and Trans constructions and subsequent minimisation [27, 23], yielding an equivalent solution for minimal $\mathsf{c}_{\mathscr{N}}$-preserving constraints via directly taking advantage of the "forbidden relations" in $\mathscr{S}_{\mathscr{N}}$.

4 Implementation and Experimental Evaluation

We provide a Python package called ReSMT (author: N. Behr) as an open-source project via GitLab[4], whose current version `0.0.3` contains both a full online documentation[5] as well as dedicated *GCM 2020 Supplementary Information* materials[6]. The computational core of our implementation is a theory (a set of definitions and constraints) in the Z3 theorem prover[7] [37], which due to its declarative nature allows for a direct encoding of our categorical constructions. Using Z3 through its Python API, we can reflect the structure of the theory in an object-oriented design, where classes represent concepts of the problem domain, such as graphs and graph morphisms, spans, rules, pushouts, etc., and generate assertions in Z3 encoding the defining properties of these categorical structures. However, a major challenge in implementing high-performance algorithms consists in evaluating and choosing between a number of alternative encodings of sets and set operations provided by the Z3 API. We first review these challenges in Section 4.1, followed by a description of the key design choices taken for ReSMT in Section 4.2 and a presentation and evaluation of some experimental results in Section 4.3. Section 4.4 comments on the correctness of the approach.

---

[4]Official ReSMT repository: https://gitlab.com/nicolasbehr/ReSMT

[5]Documentation: https://nicolasbehr.gitlab.io/ReSMT/

[6]GCM 2020 SI materials: https://nicolasbehr.gitlab.io/ReSMT/py_and_ipynb_examples/GCM2020.html

[7]Z3 solver GitHub repository: https://github.com/Z3Prover/z3

4.1   Key Challenge: Implementation of Finite Set Constructs

Albeit finite sets are evidently a well-understood and fundamental theoretical concept, choosing an algorithmically advantageous representation of finite sets within Z3 is a non-trivial matter. As described in detail in the excellent Z3 tutorial [11], the performance of Z3 when computing on sets of assertions encoding a search over set structures hinges on the precise way in which the finiteness of the sets involved is communicated to the solver. In our prototypical early 2020 implementation, we had opted for what appeared to be a natural choice: a finite set such as $S = \{x_1, x_2, x_3\}$ may be encoded by declaring an abstract Z3 sort, instantiating Z3 constants of this sort for each element and asserting that any constant of these sorts must be equal to one of the explicitly instantiated constants:

```
1  import z3                              # load the Z3 API package
2  S = z3.Solver()                        # instantiate a Z3 solver
3  X = z3.DeclareSort('X', S.ctx)         # declare a sort in the context of S
4  x1,  x2,  x3 = z3.Consts('x1 x2 x3', X)  # instantiate one constant of sort X per set element
5  x = z3.Const('x', X)                   # instantiate an auxiliary constant
6  S.add(z3.ForAll(x, z3.Or([x == x1, x == x2, x == x3]))) # add the finiteness assertion to S
```

Unfortunately, while of course logically correct, we found that this particular design choice results in poor performance of our algorithms for curating rule overlaps, both in execution time and memory consumption. As a remedy, our ReSMT package is based upon an alternative encoding of finite sets utilising the *enumeration sorts*[8] provided by Z3. For the aforementioned example, this alternative encoding reads as follows:

```
7  set, els = z3.EnumSort('set', ('x1', 'x2', 'x3'), S.ctx)
```

The above code yields an instantiation of a Z3 enumeration sort set called 'set' as well as a list of Z3 constants els (containing three constants of sort set with names 'x1', 'x2' and 'x3'). After the above call, the enumeration sort set as well as the individual constants may be utilised in a fully analogous fashion to other types of Z3 sorts and constants, yet internally provide additional information to the Z3 solver instance, encoding the finiteness axiom. We refer the reader to a dedicated section[9] of the ReSMT online documentation for in-detail performance experiments and simplified versions of the ReSMT algorithms for finding rule overlaps, demonstrating in particular the feasibility of using enumeration sorts for curating injective partial set overlaps with or without additional cardinality constraints.

4.2   Implementation Strategies for Curating Rule Overlaps

Our current ReSMT implementation (version 0.0.3) consists of approximately 1800 lines of Python 3 source code and is provided with an in-depth API documentation[10]. The core task implemented in the form of three alternative variants of algorithms is the curation of rule overlaps modulo structural constraints, currently supported already for the general data structure of *typed directed multigraphs*. We will present here some of the key aspects of our implementation in the setting of untyped directed multigraphs $G = (V_G, E_G, s_G : E_G \rightarrow V_G, t_G : E_G \rightarrow V_G)$ of relevance to the present paper. Assuming that the sets of vertices and edges of a given graph G have been encoded as enumeration sorts vG and eG (cf. Section 4.1), the source and target functions of G are instantiated as Z3 functions:

```
8  srcG = z3.Function('srcG', eG, vG)
9  trgtG = z3.Function('trgtG', eG, vG)
```

---

[8]See e.g. this tutorial: https://ericpony.github.io/z3py-tutorial/advanced-examples.htm
[9]API experiments: https://nicolasbehr.gitlab.io/ReSMT/py_and_ipynb_examples/ReSMT-API-experiments.html
[10]ReSMT API documentation: https://nicolasbehr.gitlab.io/ReSMT/API-reference/Reference.html

(Injective) graph homomorphisms $\varphi \equiv (\varphi_V, \varphi_E) : G_A \to G_B$ may then be encoded as follows:

```
10   phiV = z3.Function('phiV', vGA, vGB); phiE = z3.Function('phiE', eGA, eGB)
11   v1, v2 = z3.Consts('v1 v2', vGA); e, e1, e2 = z3.Consts('e e1 e2', eGA)
12   S.add(z3.ForAll(eA, z3.And(srcB(phiE(e)) == phiV(srcA(e)), trgtB(phiE(e)) == phiV(trgtA(e)))))
13   S.add(z3.ForAll([v1,v2], z3.Implies(phiV(v1) == phiV(v2), v1 == v2))) # phiV injectivity
14   S.add(z3.ForAll([e1,e2], z3.Implies(phiE(e1) == phiE(e2), e1 == e2))) # phiE injectivity
```

Finally, *injective partial overlaps* of finite multigraphs may be efficiently implemented as *bi-injective Boolean span predicates* $\Phi : G_A \times G_B \to \{\top, \bot\}$, which is particularly advantageous compared to a more direct encoding of monic spans of graphs in our exhaustive overlap-finding algorithms. The span predicate must satisfy the following logical assertions (i.e. bi-injectivity and a variant of a graph homomorphism property):

```
15   PhiV = z3.Function(vGA, vGB, z3.BoolSort); PhiE = z3.Function(eGA, eGB, z3.BoolSort)
16   vA, vA1, vA2 = z3.Consts('vA vA1 vA2', vGA); eA, eA1, eA2 = z3.Consts('eA eA1 eA2', eGA);
17   vB, vB1, vB2 = z3.Consts('vB vB1 vB2', vGB); eB, eB1, eB2 = z3.Consts('eB eB1 eB2', eGB);
18   astsBiInj = [z3.ForAll([vA1, vA2, vB],
19                       z3.Implies(z3.And(PhiV(vA1,vB)==True, PhiV(vA2,vB)==True), vA1==vA2)),
20            z3.ForAll([eA1, eA2, eB],
21                       z3.Implies(z3.And(PhiE(eA1,eB)==True, PhiE(eA2, eB)==True), eA1==eA2)),
22            z3.ForAll([vA, vB1, vB2],
23                       z3.Implies(z3.And(PhiV(vA,vB1)==True, PhiV(vA, vB2)==True), vB1==vB2)),
24            z3.ForAll([eA,eB1,eB2],
25                       z3.Implies(z3.And(PhiE(eA,eB1)==True, PhiE(eA, eB2)==True), eB1==eB2))]
26   astsPhiHom = [z3.ForAll([eA, eB],
27                       z3.Implies(PhiE(eA,eB)==True,
28                               z3.And(PhiV(srcA(eA),srcB(eB))==True,
29                                       PhiV(trgtA(eA),trgtB(eB))==True)))
30   S.add(astsBiInj + astsPhiHom)
```

In order to encode the necessary constructs for our three different strategies, we require a number of additional data structures and logical assertion generation methods, all of which are documented in full detail both in the ReSMT API reference as well as in several sets of tutorial examples on the documentation page of the package. For brevity, suffice it here to highlight the key conceptual difference between the *direct* aka "generate-and-test" strategy `resmt.datatypes.generateTDGoverlapsDirectstrategy` and its *implicit* strategy variant `resmt.datatypes.generateTDGoverlapsImplicitstrategy`. The *direct* strategy consists of a phase wherein all possible injective partial overlaps are determined via searching with Z3 for all possible monic Boolean span predicates as described above, followed by a second phase of forming a pushout object for each individual overlap and testing the pushouts for compliance with the structural constraints. In the *implicit* strategy, both tasks are combined into a single set of logical assertions by instantiating *ordinary sorts* (i.e. *not* enumeration sorts) for the vertex and edge sets of the tentative pushout graphs G_P, and in addition asserting the existence of injective graph homomorphisms $\alpha : G_A \hookrightarrow G_P$ and $\beta : G_B \hookrightarrow G_P$ as well as the properties that α and β should be *jointly surjective* as well es compatible with the Boolean span predicate Φ (in the sense that $\Phi(a,b) = \top \Rightarrow \alpha(a) = \beta(b)$). What at first looks like a rather baroque tautology for the structure of the equivalent *direct* strategy definition has in fact dramatic consequences for the performance of the overlap-finding algorithm, since the implicit encoding of the pushout object permits the simultaneous formulation of the forbidden pattern non-embedding constraints, such that in the *implicit* strategy no non-consistent overlaps are ever constructed.

Table 1: Experimental results of the three different strategies for the examples **P1** to **P4**. For each experiment, the *average total run-time* (over five complete runs) as well as the *maximum RAM consumption* are provided. Here, a "complete run" consists in finding *all* admissible overlaps. Note that for the case of the DPE strategy and example **P3**, we report here on exception only on the result of a single run (due to the prohibitive run-time). All experiments were performed via our ReSMT Python package (version `0.0.3`) executed in `Python 3.8.2` and with `Z3-4.8.8.0` on a computer running `macOS Catalina 10.15.6` with a *2.30GHz Intel(R) Core(TM) i7-3615QM CPU* and *16 GB* of RAM. The entry n/a signals experiments that were not possible to complete under the experimental conditions in an order of hours per run. Note that for experiment **P4**, the number of candidate overlaps could only be estimated (cf. main text).

Experiment name (G_A, G_B)	**P1** $(I_{\text{d-e}}, O_{\text{c-e}})$		**P2** $(I_{\text{b-}c_2}, O_{\text{c-}c_2})$		**P3** $(I_{\text{b-}c_4}, O_{\text{c-}c_4})$		**P4** $(I_{\text{b-}c_7}, O_{\text{c-}c_7})$	
# candidate overlaps	8		49		2426		$1.17 \cdot 10^9 \ldots 3.02 \cdot 10^{22}$	
# correct overlaps	5		4		6		9	
direct strategy	0.388 s	(37.6 MB)	2.420 s	(146.5 MB)	134.642 s	(6466.2 MB)	n/a	(n/a)
DPE strategy	0.083 s	(18.8 MB)	20.790 s	(269.8 MB)	8205.913 s	(5022.1 MB)	n/a	(n/a)
implicit strategy	0.093 s	(16.8 MB)	0.242 s	(19.5 MB)	1.079 s	(30.1 MB)	5.390 s	(57.8 MB)

4.3   Experiments and Evaluation

We present in Table 1 the experimental results for the computation of the full set of admissible overlaps of rigid graphs for four different pairs of input graphs (G_A, G_B) (chosen from the in- and output interfaces of the rules of Example 3), and for the three different strategies. We invite the readers to consult the *GCM 2020 Supplementary Information* section of the ReSMT online documentation for the precise specification of each of the computational experiments. In particular, we provide precise instructions on how to replicate these experiments by executing `Jupyter` notebook `GCM2020.ipynb` provided as part of the ReSMT GitLab repository. The examples **P1** to **P4** were chosen to illustrate the fundamental effect that graph constraints may have on the computational complexity determining injective partial graph overlaps, and with the data type of rigid graphs as introduced in Example 2 providing a prototypical example of a graph-like structure defined via negative constraints. It may be easily verified that the pushout of an injective partial overlap of a "chain" of edges π_n of length n and a "loop" of edges λ_{n+1} (with both notations as in Example 2) can only be a rigid graph if either the overlap was empty, or if the "chain" fully embeds into the "loop" (in one of $n+1$ possible ways). This precisely explains the number of "correct" overlaps for examples **P2** ($n=2$), **P3** ($n=4$) and **P4** ($n=7$), with a similar argument resulting in 5 overlaps for example **P1**. However, for the *direct* strategy also the numbers of all injective partial overlaps *without* taking into account the constraints is highly relevant, which as indicated in Table 1 exhibits an extreme growth with growing size of the graphs. Notably, while for examples **P1**, **P2** and **P3** it was possible to explicitly use our code for the case without constraints to experimentally find the precise numbers of constraint-less overlaps, in example **P4** this number could only be very roughly estimated (with a lower and upper bound provided by the number of vertex-only overlaps and the number of overlaps of the joint sets of vertices and edges without taking into account the graph homomorphism structure, respectively).

Inspecting the experimental results of Table 1, it becomes immediately evident from a comparison of run-times and maximal memory usages that, while the *DPE* strategy poses a theoretical advantage over the *direct* strategy in that it avoids computing the many overlap candidates inconsistent with the

constraints (and for small examples such as **P1** indeed shows slightly better performance), this theoretical advantage is in practice effectively voided by prohibitive memory consumption and a run-time of the order of several hours already for the relatively small example **P3**. Both strategies are incapable of computing all admissible overlaps for example **P4**, where the reason in the case of the *direct* strategy is clearly the prohibitive number of candidate overlaps. On the other hand, the *implicit* strategy appears to provide by far the most performant and robust implementation, with run-times and maximal memory consumptions for this strategy scaling rather favourably with the problem size. We thus posit that it is this type of strategy that will be capable of efficiently tackling the problem of computing rule compositions also in the highly relevant application fields of bio- and organo-chemistry, where one encounters graph constraints for the specification of the respective data types akin to the prototypical model of rigid graphs, and which we have thus identified as a viable avenue for future work and software developments.

4.4 Correctness

We discuss the correctness of the strategies and their implementations. The correctness of the *direct* and *implicit* strategies is obvious from their definitions because they both encode the original statement of the problem of finding all spans between two given graphs whose pushout object does not contain any of the forbidden patters. The correctness of the *DPE* strategy is established by Theorem 3. The implementation of these strategies has not been formally verified, but their encoding by logical constraints reflects directly the definition of the categorical constructions involved.

The implementation has been tested by a number of small examples where it is easy to calculate the expected results by hand. This provides validation that the basic constructions are implemented correctly. Where all three strategies have produced results we have compared them to establish that they are indeed functionally equivalent. In particular, when applied to the same example, they produced the same number of spans.

5 Related Work

In [31] two of the authors contributed to a review of applications of SAT and SMT solvers in graph transformation. In summary, their use in analysing graph transformation systems has increased over the last years, e.g. for the computation of strongest post-conditions [15, 16], the implementation of graph computation problems and algorithms [33, 24], model checking [32], the verification of graph properties (shapes) as invariants [40, 41], and termination analysis [13]. While these approaches share our general approach, they address very different analysis problems. i.e. none of the above compute rule overlaps.

Apart from other references already discussed, there has been work on the derivation of differential equations from rewrite rules in the context of or inspired by Kappa [18, 2, 17]. We share their motivations and aims in enabling a wealth of mathematical analysis for graph transformation systems. Such approaches are generalised by the rule algebra construction [4, 8, 5, 3, 7, 9], which provides the context for our work. The computation of relations between rules subject to constraints is also an element of critical pair analysis [30]. Recent work in this area [35] has focused on the level of granularity at which such relations are computed and presented. For example, in many applications it is not necessary to compute all relations, but instead to focus only on *essential* ones with minimal conflicting overlap. When computing relations to derive commutators, the only meaningful abstraction is by isomorphism classes, but it may be interesting to explore the use of our implementation in critical pair and dependency analysis where often only selected relations are required.

6   Conclusion

We have presented results of both theoretical and experimental nature on the efficient construction of overlap spans with constraints for the computation of commutators for stochastic rewrite systems. On the theoretical side, we showed how for compositions of SqPO rules with application conditions, the constraint checking step in this construction can be modularised, avoiding the direct generate-and-test strategy (of first composing rules and then validating them) by replacing checks for forbidden graph patterns by checks for forbidden relation patterns, and how the same technique can be used to efficiently derive minimal negative conditions preserving forbidden graph constraints. On the experimental side, we have designed a prototypical implementation of our three alternative strategies in Z3 based on the categorical concepts and constructions of the theory. We evaluated the correctness of our algorithms by testing them against each other and compared their performance through a series of experiments, which permitted us to identify the *implicit* strategy as the most robust and best performing one, which in contrast to the other two strategies scaled very well with increasing problem complexity.

We plan to implement the complete commutator computation, including constructing rules with conditions and counting their isomorphism classes, and to explore applications in rule-based models of biochemistry and adaptive networks to derive their evolution equations.

References

[1] Jakob L. Andersen, Christoph Flamm, Daniel Merkle & Peter F. Stadler (2016): *A Software Package for Chemically Inspired Graph Transformation*. In: *Graph Transformation (ICGT 2016)*, *LNCS* 9761, Springer International Publishing, pp. 73–88, doi:10.1007/978-3-319-40530-8_5.

[2] Mayur Bapodra & Reiko Heckel (2010): *From Graph Transformations to Differential Equations*. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 30, doi:10.14279/tuj.eceasst.30.431.

[3] Nicolas Behr (2019): *Sesqui-Pushout Rewriting: Concurrency, Associativity and Rule Algebra Framework*. In: *Workshop on Graph Computation (GCM 2019)*, *EPTCS* 309, Open Publishing Association, pp. 23–52, doi:10.4204/eptcs.309.2.

[4] Nicolas Behr, Vincent Danos & Ilias Garnier (2016): *Stochastic mechanics of graph rewriting*. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2016)*, ACM Press, doi:10.1145/2933575.2934537.

[5] Nicolas Behr, Vincent Danos & Ilias Garnier (2020): *Combinatorial Conversion and Moment Bisimulation for Stochastic Rewriting Systems*. *Logical Methods in Computer Science* Volume 16, Issue 3. Available at https://lmcs.episciences.org/6628.

[6] Nicolas Behr & Jean Krivine (2019): *Compositionality of Rewriting Rules with Conditions*. Available at arXiv:1904.09322 [cs.LO] .

[7] Nicolas Behr & Jean Krivine (2020): *Rewriting theory for the life sciences: A unifying framework for CTMC semantics*. In: *Graph Transformation (ICGT 2020)*, *LNCS* 12150, Springer International Publishing, pp. 185–202, doi:10.1007/978-3-030-51372-6_11.

[8] Nicolas Behr & Pawel Sobocinski (2018): *Rule Algebras for Adhesive Categories*. In: *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, *LIPIcs* 119, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 11:1–11:21, doi:10.4230/LIPIcs.CSL.2018.11.

[9] Nicolas Behr & Pawel Sobocinski (2020): *Rule Algebras for Adhesive Categories*. *Logical Methods in Computer Science* Volume 16, Issue 3. Available at https://lmcs.episciences.org/6615.

[10] Gil Benkö, Christoph Flamm & Peter F. Stadler (2003): *A Graph-Based Toy Model of Chemistry*. *J. Chem. Inf. Comput. Sci.* 43(4), pp. 1085–1093, doi:10.1021/ci0200570.

[11] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson & Christoph M. Wintersteiger (2019): *Programming Z3*. In: *Engineering Trustworthy Software Systems*, Springer International Publishing, pp. 148–201, doi:10.1007/978-3-030-17601-3_4.

[12] Pierre Boutillier, Mutaamba Maasha, Xing Li, Héctor F Medina-Abarca, Jean Krivine, Jérôme Feret, Ioana Cristescu, Angus G Forbes & Walter Fontana (2018): *The Kappa platform for rule-based modeling*. *Bioinformatics* 34(13), pp. i583–i592, doi:10.1093/bioinformatics/bty272.

[13] Harrie Jan Sander Bruggink (2015): *Grez user manual*. Available at http://www.ti.inf.uni-due.de/research/tools/grez/.

[14] Andrea Corradini, Tobias Heindel, Frank Hermann & Barbara König (2006): *Sesqui-Pushout Rewriting*. In: *Graph Transformations (ICGT 2006)*, LNCS 4178, Springer Berlin Heidelberg, pp. 30–45, doi:10.1007/11841883_4.

[15] Andrea Corradini, Barbara König & Dennis Nolte (2017): *Specifying Graph Languages with Type Graphs*. In: *Graph Transformation (ICGT 2017)*, LNCS 10373, Springer International Publishing, pp. 73–89, doi:10.1007/978-3-319-61470-0_5.

[16] Andrea Corradini, Barbara König & Dennis Nolte (2019): *Specifying graph languages with type graphs*. *J. Log. Algebr. Methods Program.* 104, pp. 176–200, doi:10.1016/j.jlamp.2019.01.005.

[17] Vincent Danos, Jerome Feret, Walter Fontana, Russell Harmer, Jonathan Hayman, Jean Krivine, Chris Thompson-Walsh & Glynn Winskel (2012): *Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models*. In: *Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012)*, LIPIcs 18, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 276–288, doi:10.4230/LIPIcs.FSTTCS.2012.276.

[18] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer & Jean Krivine (2008): *Rule-Based Modelling, Symmetries, Refinements*. In: *Formal Methods in Systems Biology (FMSB 2008)*, LNCS 5054, Springer, pp. 103–122, doi:10.1007/978-3-540-68413-8_8.

[19] Vincent Danos, Reiko Heckel & Pawel Sobocinski (2014): *Transformation and Refinement of Rigid Structures*. In: *Graph Transformation (ICGT 2014)*, LNCS 8571, Springer International Publishing, pp. 146–160, doi:10.1007/978-3-319-09108-2_10.

[20] Vincent Danos & Cosimo Laneve (2004): *Formal molecular biology*. *TCS* 325(1), pp. 69–110, doi:10.1016/j.tcs.2004.03.065.

[21] Vincent Danos & Vincent Schachter, editors (2004): *Computational Methods in Systems Biology (CMSB 2004)*. LNCS 3082, Springer Berlin Heidelberg, doi:10.1007/b107287.

[22] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers & Fernando Orejas (2012): $\mathscr{M}$-*Adhesive Transformation Systems with Nested Application Conditions. Part 2: Embedding, Critical Pairs and Local Confluence*. *Fundam. Inform.* 118(1-2), pp. 35–63, doi:10.3233/FI-2012-705.

[23] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers & Fernando Orejas (2014): $\mathscr{M}$-*adhesive transformation systems with nested application conditions. Part 1: parallelism, concurrency and amalgamation*. *Math. Struct. Comput. Sci.* 24(04), doi:10.1017/s0960129512000357.

[24] Marcus Ermler, Hans-Jörg Kreowski, Sabine Kuske & Caroline von Totth (2011): *From Graph Transformation Units via MiniSat to GrGen.NET*. In: *International Symposium on Applications of Graph Transformations with Industrial Relevance*, LNCS 7233, Springer, Berlin, Heidelberg, pp. 153–168, doi:10.1007/978-3-642-34176-2_14.

[25] Philippe Flajolet & Robert Sedgewick (2009): *Analytic Combinatorics*. Cambridge University Press, doi:10.1017/CBO9780511801655.

[26] Karsten Gabriel, Benjamin Braatz, Hartmut Ehrig & Ulrike Golas (2014): *Finitary $\mathscr{M}$-adhesive categories*. *Math. Struct. Comput. Sci.* 24(04), doi:10.1017/S0960129512000321.

[27] Annegret Habel & Karl-Heinz Pennemann (2009): *Correctness of high-level transformation systems relative to nested conditions*. *Math. Struct. Comput. Sci.* 19(02), pp. 245–296, doi:10.1017/s0960129508007202.

[28] Russ Harmer, Vincent Danos, Jérôme Feret, Jean Krivine & Walter Fontana (2010): *Intrinsic information carriers in combinatorial dynamical systems*. Chaos: *An Interdisciplinary Journal of Nonlinear Science* 20(3), p. 037108, doi:10.1063/1.3491100.

[29] Reiko Heckel, Andrea Corradini, Hartmut Ehrig & Michael Löwe (1996): *Horizontal and Vertical Structuring of Typed Graph Transformation Systems*. *Math. Struct. Comput. Sci.* 6(6), pp. 613–648, doi:10.1017/S0960129500070110.

[30] Reiko Heckel, Jochen Malte Küster & Gabriele Taentzer (2002): *Confluence of Typed Attributed Graph Transformation Systems*. In: *Graph Transformation (ICGT 2002)*, *LNCS* 2505, Springer, pp. 161–176, doi:10.1007/3-540-45832-8_14.

[31] Reiko Heckel, Leen Lambers & Maryam Ghaffari Saadat (2019): *Analysis of Graph Transformation Systems: Native vs Translation-based Techniques*. In: *Workshop on Graph Computation (GCM 2019)*, *EPTCS* 309, Open Publishing Association, pp. 1–22, doi:10.4204/EPTCS.309.1.

[32] Tobias Isenberg, Dominik Steenken & Heike Wehrheim (2013): *Bounded Model Checking of Graph Transformation Systems via SMT Solving*. In: *Formal Techniques for Distributed Systems (FMOODS/FORTE 2013)*, *LNCS* 7892, Springer, Berlin, Heidelberg, pp. 178–192, doi:10.1007/978-3-642-38592-6_13.

[33] Hans-Jörg Kreowski, Sabine Kuske & Robert Wille (2010): *Graph transformation units guided by a SAT solver*. In: *Graph Transformations (ICGT 2010)*, *LNCS* 6372, Springer, Berlin, Heidelberg, pp. 27–42, doi:10.1007/978-3-642-15928-2_3.

[34] Stephen Lack & Paweł Sobociński (2005): *Adhesive and quasiadhesive categories*. *RAIRO - Theoretical Informatics and Applications* 39(3), pp. 511–545, doi:10.1051/ita:2005028.

[35] Leen Lambers, Daniel Strüber, Gabriele Taentzer, Kristopher Born & Jevgenij Huebert (2018): *Multi-granular conflict and dependency analysis in software engineering based on graph transformation*. In: *International Conference on Software Engineering (ICSE 2018)*, ACM, pp. 716–727, doi:10.1145/3180155.3180258.

[36] Tihamer Levendovszky, Ulrike Prange & Hartmut Ehrig (2007): *Termination Criteria for DPO Transformations with Injective Matches*. *Electron. Notes Theor. Comput. Sci.* 175(4), pp. 87–100, doi:10.1016/j.entcs.2007.04.019.

[37] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[38] Nebras Nassar, Jens Kosiol, Thorsten Arendt & Gabriele Taentzer (2019): *Constructing Optimized Validity-Preserving Application Conditions for Graph Transformation Rules*. In: *Graph Transformation (ICGT 2019)*, *LNCS* 11629, Springer, pp. 177–194, doi:10.1007/978-3-030-23611-3_11.

[39] Guilherme Rangel, Leen Lambers, Barbara König, Hartmut Ehrig & Paolo Baldan (2008): *Behavior Preservation in Model Refactoring Using DPO Transformations with Borrowed Contexts*. In: *Graph Transformations (ICGT 2008)*, *LNCS* 5214, Springer, pp. 242–256, doi:10.1007/978-3-540-87405-8_17.

[40] Dominik Steenken (2015): *Verification of infinite-state graph transformation systems via abstraction*. Ph.D. thesis, University of Paderborn. Available at https://digital.ub.uni-paderborn.de/ubpb/urn/urn:nbn:de:hbz:466:2-15768.

[41] Dominik Steenken, Heike Wehrheim & Daniel Wonisch (2011): *Sound and complete abstract graph transformation*. In: *Brazilian Symposium on Formal Methods*, Springer, pp. 92–107, doi:10.1007/978-3-642-25032-3_7.