Efficient Modular Graph Transformation Rule Application

Jakob L. Andersen\(^1\) \& Rolf Fagerberg\(^1\) \& Juri Kolčák\(^1,2\) \& Christophe V.F.P. Laurent\(^1\) \& Daniel Merkle\(^1\) \& Nikolai Nøjgaard\(^1\)

\(^1\)Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark
\(^2\)Department of Systems Biology, Harvard Medical School, Boston, MA, USA

Abstract. Graph transformation formalisms have proven to be suitable tools for the modeling of chemical reactions. They are well established in theoretical studies [15] and increasingly also in practical applications in chemistry [11, 28]. The latter is made feasible via the development of programming frameworks which make the formalisms executable [5, 19, 12].

The application of such frameworks to large networks of chemical reactions poses unique computational challenges due to the nature of the involved graphs, since these often consist of many individual connected components. While the existing methods for implementing graph transformations can be applied to such graphs, the combinatorics of constructing the graph matches involved in the application of graph transformation rules quickly becomes a computational bottleneck when the size of the chemical reaction network grows.

In this contribution, we develop a new method of enumerating graph matches during graph transformation rule application. The method is designed to improve performance in such scenarios and is based on constructing graph matches in an iterative, component-wise fashion which allows redundant applications to be detected early and pruned. We further extend the algorithm with an efficient heuristic based on local symmetries of the graphs, which allow us to detect and discard isomorphic applications early. Finally, we conduct chemical network generation experiments on real-life as well as synthetic data and compare against the state-of-the-art algorithm in the field.

1 Introduction

A reaction network is a set of chemical reactions describing how a collection of molecules interact. Reaction networks are used in many areas of research, for instance in biochemical pathway modeling [22], in drug design [17], and in the study of planetary atmospheres [29]. Known networks are commonly made available through manually curated databases, of which the KEGG [22] collection of metabolic networks is one example. Networks stored in such databases are inherently subject to sampling bias: reactions and molecules are only recorded if the database curator deems them of interest in some known chemistry being examined [11]. If one wishes to explore unknown chemistries, e.g., when designing new enzymatic mechanisms [2], one may instead turn to the generation of reaction networks using rule-based approaches.

This work is supported by the Novo Nordisk Foundation grant NNF19OC0057834 and by the Independent Research Fund Denmark,Natural Sciences, grants DFF-0135-00420B and DFF-7014-00041.

E-mail addresses: jlandersen@imada.sdu.dk (Jakob L. Andersen) rolf@imada.sdu.dk (Rolf Fagerberg) juri.kolcak@gmail.com (Juri Kolčák) christophe@imada.sdu.dk (Christophe V.F.P. Laurent) daniel@imada.sdu.dk (Daniel Merkle) nojgaard@imada.sdu.dk (Nikolai Nøjgaard)
Formal rule-based methods for generating reaction networks were already being investigated as early as the 90's [16]. However, it is only recently that practical application has been made possible via the development of computational frameworks such as Kappa [12], BioNetGen [19], and MØD [5]. Of these, Kappa and BioNetGen employ a unique labeling of connections, suited for an abstract representation of binding sites and entire molecules, useful in modeling biological networks. The unique labeling scheme is, however, ill suited for the modeling of reaction networks, where the complete structure of each molecule at the level of individual bonds and atoms is of interest.

To the best of our knowledge, MØD [5] is the only available computational framework with atom-level modeling of molecules. Indeed, its modeling is well aligned with the standard textbook description of molecules as chemical graphs, i.e., undirected graphs with vertices labeled by atom types and edges labeled by bond types. As argued in [7], this strikes a good balance between chemical expressiveness and computational feasibility when studying reaction networks in a generative manner. We use the term chemical graph and the term molecule interchangeably from now on.

In MØD, the generation of reaction networks is achieved by means of rules for transforming chemical graphs. A rule has a left side chemical graph pattern which specifies the minimum necessary molecular context for the transformation to take place. The rule can be applied to any molecule (or combination of molecules, for patterns with multiple parts) that contains the pattern. The right hand side of the rule then specifies how to transform the part of the molecule matching the pattern. Thus, applying a rule means matching its left side pattern to a molecule and then executing the transformation specified. In simple terms, a rule models a chemical reaction (or a set of chemical reactions with a common core, depending on the extent of the context specified) and applying the rule executes the reaction. Constructing chemical networks is done by repeated application of the rules to any combination of the molecules generated, starting from some initial set of molecules. The formalism underpinning MØD is the so-called double pushout (DPO) approach to algebraic graph transformation. Its use in MØD has been explored in multiple publications [4, 6, 10].

A graph transformation rule application is a computationally intensive task as it consists of monomorphism enumeration between the left side graph pattern of the rule and the host graph (i.e., the educt molecules of the reaction executed by the rule). For larger reaction networks, the sheer number of rule and host graph combinations makes the efficiency of the rule application of paramount importance.

The nature of the challenge lies in identifying combinations of molecules which constitute a host graph allowing a rule to be applied. One way to do this is to compose the host graph and check the rule conditions at the same time. This approach is employed in the state-of-the-art method [5], which iterates over the connected components of the left side graph (i.e., the pattern) of the rule, while using partial rule application (PA) [3, 8]. The partial application of the rule is carried out by replacing a component of the rule left side graph by the host graph molecule they match. The full rule application is then achieved by repeated partial application, each time generating a new version of the rule, until all left side components are replaced.

The number of rules generated by this partial application approach is in the worst case exponential in the number of connected components of the left side graph of the rule. Moreover, if the left side graph of the rule and the host graph are highly symmetric, an exhaustive enumeration will invariably lead to redundant isomorphic applications. Ideally, the symmetries of both the rule left side and host graph should be taken into account when enumerating rule applications. However, since the rule is modified during the partial application, keeping track of such symmetries is difficult. While the rules can be pruned by pairwise isomorphism checking, such an operation is
computationally rather expensive. This is further exacerbated when combined with the partial application approach, where the generated rules grow in size as more molecules are matched onto the left side components. For complex chemical networks with higher connected component counts and larger molecules, e.g., enzyme mechanism networks [2], the current algorithmic approach therefore struggles.

To address the combinatorial issues highlighted above, we in this paper develop a new algorithm for the computation of all possible rule applications in a given set of molecules. Similarly to the partial application approach, our algorithm relies on iterating over the connected components of the rule left side graph. However, instead of generating new versions of the rule, we explicitly construct the match (monomorphism) of the rule into the host graph. The match allows linking the components of the left side graph of the rule and the host molecules without directly modifying them, thus avoiding the need to create copies of the rules. Furthermore, we can also identify local symmetries of both the molecules and the rule and exploit them to prune partial matches known to lead to the same reactions.

We illustrate the speedup obtainable by this direct match construction versus the partial application approach in several experiments including both synthetic and chemical data. The results on synthetic data illustrate the potential for an exponential speedup with respect to the number of rule left side components. A further improvement is then observed as a result of on-the-fly symmetry pruning. The results on chemical data demonstrate that a speedup is achieved across wide variety of examples, even in cases with few rule left hand side components and close to no symmetries.

2 Preliminaries

In this contribution, we consider finite labeled simple graphs $G = (V, E, \lambda)$, with vertex set $V = V(G)$, edge set $E = E(G)$ and labeling function $\lambda : V \cup E \to \mathcal{L}$ over some set $\mathcal{L}$ equipped with an equality relation.

We assume the vertex set $V(G)$ is ordered by an arbitrary total order, represented by an index set $I = \{1, \ldots, |V(G)|\}$. By abuse of notation, we unify each vertex $v_i \in V(G)$ with its corresponding index $i$. This is in line with the practical representation of graphs where vertices explicitly or implicitly have such an index.

Following the usual convention, given two (unlabeled) graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, a graph (homo)morphism is a function $\varphi : G_1 \to G_2$, with domain and codomain $\text{dom}(\varphi) = V_1$ and $\text{codom}(\varphi) \subseteq V_2$, which preserves edges. That is, for all $(u, v) \in E_1$ we have $(\varphi(u), \varphi(v)) \in E_2$. If the two graphs are labeled, i.e., $G_1 = (V_1, E_1, \lambda_1)$, $G_2 = (V_2, E_2, \lambda_2)$, then $\varphi$ must further preserve labels, i.e., for any $v \in V_1$ we have $\lambda_1(v) = \lambda_2(\varphi(v))$.

An injective graph morphism $\varphi$ is called a monomorphism, and if one such exists we say that $G_1$ embeds in $G_2$. A monomorphism $\varphi$ that is also bijective is called an isomorphism, and if one such exists we say that $G_1$ and $G_2$ are isomorphic, written as $G_1 \simeq G_2$. If an isomorphism is from a graph to itself, it is called an automorphism, and represents a symmetry of the graph.

For all of the morphism types we may deal with partial versions of them. Notationally we write them with a harpoon, e.g., $\varphi : G_1 \rightharpoonup G_2$. For such morphisms we restrict the domain and codomain to only include vertices of $G_1$ and $G_2$ that $\varphi$ is defined on.

In later sections, we build graphs by combining connected graphs. For this purpose, we utilize the following construction.

**Definition 2.1 (Union Graph)** A union graph $G$ of dimension $n$ is a vector $(G_1, \ldots, G_n)$ of
connected graphs.

For any graph \( G \), let \( G(G) = \{ i \in \{1, \ldots, n \} \mid G_i \simeq G \} \) be the set of indices of \( G \) which correspond to graphs isomorphic to \( G \).

The union graph \( G \) itself specifies a graph \( \tilde{G} = \bigcup_{i=1}^{n} G_i \) defined as the disjoint union of the component graphs in \( G \). Additionally, each vertex \( v \in V(\tilde{G}) \) of the union graph can be uniquely mapped to a vertex of one of the component graphs of \( G \). We formalize the mapping as a bijection \( \gamma: V(\tilde{G}) \to \bigcup\limits_{i \in \{1, \ldots, n\}} (\{i\} \times V(G_i)) \). By abuse of notation, we write \( v = (i, w) \) for any \( v \in V(\tilde{G}) \) such that \( \gamma: v \mapsto (i, w) \).

**Definition 2.2 (Union Graph Extension)** Given a union graph \( G = (G_1, \ldots, G_n) \) and a graph \( G \), the extension of \( G \) with \( G \) is the union graph:

\[
G \cdot G = (G_1, \ldots, G_n, G)
\]

For the transformation of a graph into another, we use the Double Pushout (DPO) approach [14]. For a detailed overview of the different variations of DPO we refer to [18]. For this paper, we use DPO specifically as defined in [5] which is aimed at using DPO to model chemistry. In this framework, a rule \( p = (L \xleftarrow{l} K \xrightarrow{r} R) \) consists of a left graph \( L \), a context graph \( K \), and a right graph \( R \), as well as monomorphisms \( l \) and \( r \) describing how \( K \) is embedded in \( L \) and \( R \). The transformation captured by a rule consists of replacing the graph \( L \) with the graph \( R \), while preserving the common parts, identified by \( K \), \( l \), and \( r \). Note that \( L \), \( K \), and \( R \) are not necessarily connected graphs.

The application of a rule \( (L \xleftarrow{l} K \xrightarrow{r} R) \) to a graph \( G \) requires the existence of a morphism \( m: L \to G \), which we, as per [5], also require to be a monomorphism. We refer to such a morphism as a match of rule \( p \) into \( G \). However, a match \( m \) alone is not enough to guarantee that the rule can be applied, as the resulting structure has to be a valid graph. We therefore require the usual gluing conditions of DPO transformation [15, Def. 3.9], namely the identification condition and the dangling condition. However, as \( m \) is injective the identification condition is trivially fulfilled. In addition we add a parallel edge condition, as we require the graphs to be simple. We thus arrive at:

**Definition 2.3 (Valid Match)** Let \( p = (L \xleftarrow{l} K \xrightarrow{r} R) \) be a rule, let \( G \) be a graph, and let \( m: L \to G \) be a match of \( p \) into \( G \).

Then the match \( m \) is valid if the following conditions are satisfied:

1. **Dangling Condition**: If a vertex \( u \in V(L) \) is removed, \( u \notin \text{codom}(l) \), all the incident edges must also be removed: For each edge \( (m(u), v') \in E(G) \) there must be an edge \( (u, v) \in E(L) \) with \( m(v) = v' \).

2. **Parallel Edge Condition**: For any pair of vertices \( v, w \in V(K) \) that the rule creates an edge between, \( (l(v), l(w)) \notin E(L) \) and \( (r(v), r(w)) \in E(R) \), the graph \( G \) may not have an edge between them, \( (m(l(v)), m(l(w))) \notin E(G) \).

We refer to the application of a rule \( p \) on a graph \( G \) with a valid match \( m: L \to G \) as a direct derivation and denote it by \( G \xrightarrow{p,m} H \), where \( H \) is again a labeled simple graph [5].

Given a union graph \( G \) such that \( \tilde{G} = G \), and a derivation \( G \xrightarrow{p,m} H \), the match \( m \) is also valid for an arbitrary extension \( G \cdot G' \), giving us the derivation \( G \cdot G' \xrightarrow{p,m} (H) \cdot G' \). However,
the extension graph $G'$ is irrelevant for the actual graph transformation. This leads us to define a notion of a minimal graph into which a match can be embedded. This notion is captured by what is known as a proper derivation [4] and the corresponding proper match. A match is proper if every connected component of the host graph $G$ intersects the codomain of the match $m$. A derivation is then proper if its match is proper. We can express this notion very naturally using union graphs.

**Definition 2.4 (Proper Match)** Let $G = (G_1, \ldots, G_n)$ be a union graph, let $(L \leftarrow K \rightarrow R)$ be a rule, and let $m: L \rightarrow G$ be a match.

Then $m$ is proper if for each $i \in \{1, \ldots, n\}$, there exist $v \in V(L)$ and $w \in V(G_i)$ such that $m(v) = (i, w)$.

Given a valid proper match $m: L \rightarrow G$, the derivation $G_{p,m} \Rightarrow H$ is called proper.

### 3 Enumerating Proper Derivations

As stated in the introduction, the goal of this paper is to present an algorithm to enumerate the possible direct derivations of a rule one can create by combining a set of given graphs. We can formalize this problem as follows.

**Problem 1** Let $\mathcal{G}$ be a finite set of pairwise non-isomorphic connected graphs and $p = (L \leftarrow K \rightarrow R)$ a rule.

Enumerate all proper derivations $\mathcal{G}_{p,m} \Rightarrow H$ for all selections of cardinality $n \in \mathbb{N}$ and union graphs $G \in \mathcal{G}^n$, but only up to reordering of the constituent graphs of each $G$.

That is, we want to enumerate all proper derivations that can be obtained by constructing the host graph as combinations of graphs in $\mathcal{G}$. Clearly, the host graph obtained from a simple reordering of elements in a combination of graphs in $\mathcal{G}$ is uninteresting, which is why we ignore such reordering of the constituent graphs of each $G$. As we are only interested in proper derivations the cardinality $n$ is bounded by the number of connected components of $L$.

In the following we assume the left side graph $L$ of the rule $p$ corresponds to a union graph $L = \bar{L}$. This allows us to approach the problem by iterating over the component graphs $(L_1, \ldots, L_k) = L$ and accordingly extending the host graph $G$.

More precisely, we utilize a notion of a partial match which is simply a partial monomorphism of the rule left side into the host graph. We iteratively extend a partial match with a monomorphism from a connected component of the left graph of a rule, eventually obtaining a total match. We formalize this in the notion of a partial match extension, akin to the union graph extension (Definition 2.2).

**Definition 3.1 (Partial Match Extension)** Let $(\bar{L} \leftarrow K \rightarrow R)$ be a rule with $\bar{L} = (L_1, \ldots, L_k)$, $G = (G_1, \ldots, G_n)$ a union graph, $i \in \{1, \ldots, k\}$ and $x \in \{1, \ldots, n\}$ two indices, and let $m: \bar{L} \rightarrow G$ be a partial match. Finally, let $\varphi: L_i \rightarrow G_x$ be a monomorphism such that $L_i$ is undefined in $m$ and for all $v \in \text{codom}(\varphi)$, $(x, v) \notin \text{codom}(m)$. 

Then the partial match extension is the partial match \( m \cup_{x}^{i} \varphi : \tilde{L} \rightarrow \tilde{G} \) of the rule \( p \) into \( \tilde{G} \) defined as follows:

\[
m \cup_{x}^{i} \varphi = \begin{cases} (i, v) & \mapsto (x, \varphi(v)) \quad \text{for all } v \in V(L_i) \\
 w & \mapsto m(w) \quad \text{for all } w \in \text{dom}(m) 
\end{cases}
\]

To ease notation, we write simply \( m \cup \varphi \) where the indices \( i \) and \( x \) are obvious from the context.

The partial match extension thus lifts the monomorphism \( \varphi \) from one of the component graphs into the union graph \( G \) itself. The resulting partial morphism is always a partial monomorphism since \( m \) is restricted to components other than \( L_i \) and the codomain of \( \varphi \) lifted to the union graph is disjoint with the codomain of \( m \). The match extension does not however guarantee that the resulting match obtained from repeated extensions of a partial match is valid.

To ensure that any extension might potentially lead to a valid match we first extend Definition 2.3 of a valid match to partial matches. This is done by limiting the scope of the definition to the domain of the match itself, indicated by the underlined sections. Condition (1) again corresponds to the dangling edge condition while Condition (2) corresponds to the parallel edge condition.

**Definition 3.2 (Valid Partial Match)** Given a partial match \( m : L \rightarrow G \) for a rule \( p = (L \leftarrow K \rightarrow R) \) with \( L = (L_1, \ldots, L_k) \) on a graph \( G \), we say that \( m \) is valid if it satisfies the following criteria:

1. **Dangling Condition**: For a vertex \( u \in V(L) \) defined in \( m \) that is removed, \( u \notin \text{codom}(l) \), all the incident edges must also be removed: for each edge \((m(u), v') \in E(G)\) there must be an edge \((u, v) \in E(L)\) with \( m(v) = v' \).

2. **Parallel Edge Condition**: For any pair of vertices \( v, w \in V(K) \) that the rule creates an edge between, \((l(v), l(w)) \notin E(L)\) and \((r(v), r(w)) \in E(R)\), and are in the match, \((l(v), l(w)) \in \text{dom}(m)\), the graph \( G \) may not have an edge between them, \((m(l(v)), m(l(w))) \notin E(G)\).

Clearly, any partial match obtained from a valid match is again valid. As a result, we are interested in extensions that lead to valid partial matches.

**Definition 3.3 (Valid Partial Match Extension)** Let \( (L \leftarrow K \rightarrow R) \) be a rule with \( L = (L_1, \ldots, L_k), G = (G_1, \ldots, G_n) \) a union graph, \( i \in \{1, \ldots, k\} \) and \( x \in \{1, \ldots, n\} \) two indices, and let \( m : L \rightarrow G \) be a valid partial match. Finally, let \( \varphi : L_i \rightarrow G_x \) be a monomorphism such that \( L_i \) is undefined in \( m \).

Then the extension \( m' = m \cup_{x}^{i} \varphi \) is valid if it is defined and for any vertex \( v \in V(L_i) \) we have that \((i, v) \in V(\tilde{L})\) satisfy the Dangling Condition of Definition 3.2 in \( m' \) and any pair \( v, w \), where \( w \in V(\tilde{L}) \), satisfy the Parallel Edge Condition of 3.2 in \( m' \).

Definition 3.3 ensures that any valid partial match extension is defined and produces another valid partial match by iterating over the vertices in \( L_i \). The fact that \( m' \) is valid results from the fact that \( m \) is valid, and hence any vertex not satisfying Definition 3.2 must invariably involve vertices of \( L_i \). We note that for any extension to be valid, the corresponding monomorphism must itself be a valid partial match, and it thus suffices to only extend these monomorphisms. However, even if both \( m \) and \( \varphi \) are valid partial matches, their extension might not produce a valid partial match, as is depicted in Figure 1.
Figure 1: A rule application with the union graphs $L = (L_1, L_2)$ and $G = (G_1)$ corresponding to the connected components in $\tilde{L}$ and $\tilde{G}$ respectively. The red arrows represent the two monomorphisms $\varphi_1: L_1 \to G_1$ and $\varphi_2: L_2 \to G_1$, each individually a valid partial match. The match $m = \varphi_1 \cup_1 \varphi_2$ and the corresponding final extension, is however not valid because a parallel edge is created in $H$.

The partial match extension allows us to construct proper matches of a rule $p = (\tilde{L} \leftarrow K \rightarrow R)$ into a graph $\tilde{G}$ iteratively over the component graphs of the left side union graph $L = (L_1, \ldots, L_k)$. Given a match $m: \tilde{L} \to \tilde{G}$, we let $m_i$ be the monomorphism mapping $L_i$ into some component graph of $G$.

Property 3.4 (Match Decomposition) Let $G$ be a set of graphs and let $p = (\tilde{L} \leftarrow K \rightarrow R)$ be a rule with $L = (L_1, \ldots, L_k)$. Then, any proper match $m: \tilde{L} \to \tilde{G}$, where $G \in G^n$ can be rewritten as a sequence of valid partial match extensions $m = m_0 \cup m_1 \cup \cdots \cup m_k$ where $m_0: \tilde{L} \to \tilde{G}$ is the empty match of $\tilde{L}$ into $\tilde{G}$, $\text{dom}(m_0) = \text{codom}(m_0) = \emptyset$.

We can always construct an empty match between any left side graph and any host graph. In what follows we thus assume the existence of the empty match implicitly and do not write it in the decomposition.

Following Property 3.4, we know we can construct any proper match of a rule $(L \leftarrow K \rightarrow R)$ into a collection of graphs from a given set $G$ by iteratively extending a partial match over the connected components of the left side graph $L$. The iterative approach is useful in the context of Problem 1 as it allows us to iteratively construct a host union graph $G$ alongside the match $m$. Such iterative extension of the host graph restricts the possible order of the components of $G$ by the order in which the connected components of the left side graph $L$ are matched. However, the graph $G$ being a disjoint union of the component graphs, any reordering of the components of $G$ amounts only to a particular type of automorphism of $G$.

Property 3.5 (Component Graph Order Independence) Let $G = (G_1, \ldots, G_n)$ and $H = (H_1, \ldots, H_n)$ be two union graphs such that there exists a bijection $i: \{1, \ldots, n\} \to \{1, \ldots, n\}$ with $G_i = H_i(i)$ for all $i \in \{1, \ldots, n\}$. Then, the two graphs $G \simeq H$ are isomorphic.

Property 3.5 formalizes our earlier claim that we are only interested in one of the possible orderings of the components graphs in a union graph. We can then solve Problem 1 by Algorithm 1. The algorithm enumerates valid matches recursively, interleaving match extensions with the host union graph $G$ extensions. Since each host graph extension is followed by a match extension into the new component, the constructed matches are guaranteed to be proper as long as each graph $G \in G$ is connected.
The match extensions are always conducted in the order given by the component graphs \( L_1, \ldots, L_k \) of the left side graph of the rule. The partial matches can then be arranged in a tree structure with the empty partial match \( m_0 : \emptyset \rightarrow \) in the root. Then, any node on the \( i \)-th level of the tree represents a partial match \( m : L_i \rightarrow G \) for some \( G \in \mathcal{G}^n \) defined on \( L_1, \ldots, L_i \), and any child of the node represents a possible extension \( m \cup \varphi \), where \( \varphi : L_{i+1} \rightarrow G \) for some \( G \in \mathcal{G} \). The algorithm recursively navigates the above tree structure in a depth-first approach. This is done by only keeping a single partial match and host union graph in memory, which we extend and shorten as necessary.

More precisely, the algorithm first computes all monomorphisms, that are also valid partial matches, for each connected component \( L_i \in L \) (Line 4). The procedure \textsc{EnumeratePartial} is then called recursively with the parameter \( i \). At each call, we check if \( m \) is total, and hence a match, in which case we yield the corresponding direct derivation (Line 11). Otherwise, we enumerate all computed monomorphisms for \( L_i \), and for each \( \varphi : L_i \rightarrow G \) we enumerate all possible positions of \( G \) within \( G \) in which we try and extend \( m \) (Line 15). In addition, we also enumerate the case where \( \varphi \) is extended into a graph not yet in \( G \) (Line 20). Finally, in all cases, we remove \( \varphi \) from \( m \) when returning from the recursively called function, as well as removing \( G \) from the end of \( G \) if it was extended.

**Theorem 3.6** Let \( \mathcal{G} \) be a finite set of pairwise non-isomorphic graphs and \( p = (L \xleftarrow{L} K \xrightarrow{R} R) \) a rule. Then Algorithm 1 enumerates all proper derivations as defined in Problem 1.

**Proof:** The algorithm constructs all possible combinations of the monomorphisms \( \varphi : L_i \rightarrow G \) for some \( i \in \{1, \ldots, k\} \) and \( G \in \mathcal{G} \), filtering them for validity. As we are only interested in proper derivations, there is no need to extend the host graphs beyond the component graphs required by the monomorphisms themselves. It is therefore easy to see that all proper matches of interest and the resulting derivations are enumerated at least once.

What remains to be shown is that each such match, respectively derivation, is only enumerated for a single host graph. The order of the component graphs of the host graphs constructed by Algorithm 1 is uniquely determined by the rule left side graph components and the match itself. The order of the rule left side components being fixed, change in the order of the host graph components necessitates a change of the match itself.

The set \( \mathcal{G} \) containing pairwise non-isomorphic graphs, two host graphs \( \tilde{G} \simeq \tilde{G}' \in \mathcal{G}^n \) being isomorphic necessitates they only differ on the order of their component graphs, as the component graphs themselves have to be identical.

\[ \square \]

4 Enumerating Non-isomorphic Derivations

Algorithm 1 introduced in Section 3 enumerates the possible direct derivations one can create given a rule and a set of non-isomorphic graphs. Depending on the problem domain, however, some of the enumerated derivations might be redundant. As an example, assume we are given a rule \( p \), two direct derivations \( G_1 \xrightarrow{m_1} H_1, G_2 \xrightarrow{m_2} H_2 \) enumerated by Algorithm 1, and an isomorphism \( \varphi \) between \( G_1 \) and \( G_2 \) such that \( \varphi \circ m_1 = m_2 \). Then, clearly, the host graphs \( H_1 \simeq H_2 \) must also be isomorphic. Only one of the two derivations is therefore relevant in multiple application scenarios, such as generation of reaction networks [5].

To identify such redundant derivations, we utilize the concept of derivation isomorphism. Such isomorphism necessarily depends on the building blocks of the derivation itself, that is the host
Algorithm 1 Enumerate($G, p = (\bar{L} = (L_1, \ldots, L_k) \leftarrow K \rightarrow R))$

1: $G \leftarrow$ the empty union graph  
2: $m \leftarrow$ the empty partial match $\bar{L} \rightarrow \bar{G}$
3: for each $i \in \{1, \ldots, k\}$ do
4: \hspace{1em} $L_i \leftarrow \{ \varphi: L_i \rightarrow G \mid G \in G \land \varphi \text{ is a valid partial match}\}$
5: \hspace{1em} if $|L_i| = 0$ then
6: \hspace{2em} return
7: \hspace{1em} EnumeratePartial(1)
8: procedure EnumeratePartial($i$)
9: \hspace{1em} if $i = k + 1$ then \hspace{6em} $\triangleright$ $m$ is total
10: \hspace{2em} yield $\bar{G} \xrightarrow{p,m} H$
11: \hspace{1em} return
12: \hspace{1em} for each $\varphi: L_i \rightarrow G \in L_i$ do
13: \hspace{2em} for each $x \in \{y \in \{1, \ldots, n\} \mid G_y = G\}$ do
14: \hspace{3em} if $m \cup x \varphi$ is valid then \hspace{6em} $\triangleright$ try $\varphi$ on existing graphs
15: \hspace{4em} $m \leftarrow m \cup x \varphi$
16: \hspace{4em} EnumeratePartial($i + 1$)
17: \hspace{4em} remove $\varphi$ from $m$.
18: \hspace{2em} $G \leftarrow G \cdot G$
19: \hspace{1em} if $m \cup |G| \varphi$ is valid then \hspace{6em} $\triangleright$ try $\varphi$ on a new graph
20: \hspace{2em} $m \leftarrow m \cup |G| \varphi$
21: \hspace{2em} EnumeratePartial($i + 1$)
22: \hspace{2em} remove $\varphi$ from $m$
23: \hspace{1em} remove $G$ from $G$

graph, the rule and the match. A simple graph isomorphism suffices for the host graph, we thus begin with isomorphism of the graph transformation rules.

**Definition 4.1 (Rule Isomorphism)** Let \( p_1 = (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1) \) and \( p_2 = (L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2) \) be two rules. We say that a triplet of graph isomorphisms \( \varphi = (\varphi_L: L_1 \rightarrow L_2, \varphi_K: K_1 \rightarrow K_2, \varphi_R: R_1 \rightarrow R_2) \) is a rule isomorphism between \( p_1 \) and \( p_2 \) if the following diagram commutes:

\[
\begin{array}{ccc}
L_1 & \xleftarrow{l_1} & K_1 & \xrightarrow{r_1} & R_1 \\
\uparrow \varphi_L & & \uparrow \varphi_K & & \uparrow \varphi_R \\
L_2 & \xleftarrow{l_2} & K_2 & \xrightarrow{r_2} & R_2
\end{array}
\]

If \( p_1 = p_2 \) we say \( \varphi \) is a rule automorphism of \( p_1 \). The internal graph isomorphisms \( \varphi_L, \varphi_K, \varphi_R \) of a rule automorphism \( \varphi \) are necessarily also graph automorphisms.

A rule automorphism perfectly preserves the transformation prescribed within the rule itself. Thus, given a pair of valid matches \( m \) and \( m' \) such that \( m' = m \circ \alpha_L \) where \( \alpha_L \) is the first component of a rule automorphism \( \alpha \), the derivations \( G \xrightarrow{m \cdot p} H_1 \) and \( G \xrightarrow{m' \cdot p} H_2 \) necessarily produce isomorphic results \( H_1 \simeq H_2 \). Here, \( m \) and \( m' \) are an example of what we refer to as isomorphic matches. In general, isomorphic partial matches are defined as follows.

**Definition 4.2 (Isomorphic Partial Matches)** Let \( G_1, G_2 \) be two graphs, \( p = (L \xleftarrow{l} K \xrightarrow{r} R) \) a rule, and \( m_1: L \rightarrow G_1, m_2: L \rightarrow G_2 \) two valid partial matches of \( L \) into \( G_1 \) and \( G_2 \) respectively.

We say the partial matches \( m_1 \) and \( m_2 \) are isomorphic, \( m_1 \simeq m_2 \), if there exists an automorphism \( (\alpha_L, \alpha_K, \alpha_R) \) of the rule \( p \) and a graph isomorphism \( \varphi_G: G_1 \rightarrow G_2 \) such that the following diagram commutes:

\[
\begin{array}{ccc}
L & \xleftarrow{\alpha_L} & L \\
\downarrow m_2 & & \downarrow m_1 \\
G_2 & \xleftarrow{\varphi_G} & G_1
\end{array}
\]

Building on rule isomorphisms, match isomorphisms also preserve the structure of the graph transformation, guaranteeing isomorphism of the target graphs. Hence, we can extend this notion of structure preservation to direct derivations.

**Definition 4.3 (Isomorphic Derivations)** Let \( p = (L \xleftarrow{l} K \xrightarrow{r} R) \) be a rule, and \( G_1 \xrightarrow{p \cdot m_1} H_1 \) and \( G_2 \xrightarrow{p \cdot m_2} H_2 \) two proper derivations. Then the two derivations are isomorphic if \( m_1 \simeq m_2 \).

By only considering non-isomorphic derivations Problem 1 can be restated as:

**Problem 2** Let \( G \) be a finite set of pairwise non-isomorphic connected graphs and rule \( p = (L \xleftarrow{l} K \xrightarrow{r} R) \) a rule.
Then we want to enumerate all pairwise non-isomorphic proper derivations

\[ \overline{G} \xrightarrow{p,m} H \]

where \( G \in \mathcal{G}^n \) for any \( n \in \mathbb{N} \).

Isomorphic matches, as per Definition 4.2, defines an equivalence relation on the set of matches enumerated by Algorithm 1. The solution to Problem 2 thus entails enumerating exactly one representative match for each of the classes of said equivalence relation. Efficiently constructing such representatives in an iterative manner, however, turns out to be nontrivial due to the complex interplay between rule and host graph automorphisms.

In the following, we propose a heuristic for pruning partial matches, which can be determined to only extend into derivations isomorphic to ones already enumerated. The heuristic aims to identify canonical representatives for each equivalence class based on a total order on matches. The total order itself is composed of two subparts. The first part imposes an order on the way we extend by monomorphisms of isomorphic component graphs of the left side of the rule. The second part prunes matches based on the automorphisms in a single connected component either of the left side of the rule or of the host graph.

The heuristic does not guarantee that we do not enumerate any isomorphic matches. The details of such cases are described towards the end of this section. Nonetheless, the results in Section 6 show the heuristic provides a significant speedup to the algorithm when one is only interested in enumerating non-isomorphic derivations.

### 4.1 Order-Preserving Match Extensions

Consider a rule \( p = (\overline{L} \xrightarrow{\ell} K \xrightarrow{r} R) \) where \( L = (L_1, \ldots, L_k) \) and let \( L_i, L_j \in L \) such that \( L_i \simeq L_j \). Even though \( L_i \) and \( L_j \) are isomorphic, their semantics might differ in \( p \), i.e., the corresponding isomorphism does not exist in \( R \). We say that the graphs \( L_i \) and \( L_j \) are isomorphic with respect to \( p \), denoted \( L_i \simeq_p L_j \), if there exists an automorphism \( (\alpha_L, \alpha_K, \alpha_R) \) of \( p \) such that \( \alpha_L(\overline{L}_i) = \overline{L}_j \).

Given such components \( L_i \) and \( L_j \) and a match \( m \), we can always obtain a new match \( m' \simeq m \) by swapping the monomorphisms used to map \( L_i \) and \( L_j \). Formally, to be able to exchange the monomorphisms, we have to modulate them with the automorphism \( \alpha_L \), respectively its inverse. For simplicity of the notation, we instead assume, without loss of generality, that any components \( L_i \simeq_p L_j \) are representationally equivalent, \( L_i = L_j \) for the remainder of the section.

The existence of two components \( L_i \simeq_p L_j \) automorphic with respect to the rule \( p \) can thus force Algorithm 1 to enumerate isomorphic matches. Several examples of such isomorphic matches are given in Figure 2. Since the monomorphisms mapping the components \( L_i \) and \( L_j \) are exchangeable, it is the order of said monomorphisms that differentiates between the isomorphic matches. In this section, we define a total order on the monomorphisms, which can be used to identify a canonical representative among the isomorphic matches produced by monomorphism swapping between rule automorphic components \( L_i \simeq_p L_j \).

We construct the order in three steps. First, we compare on the host graphs of the monomorphisms. In case of equality, we compare on the indices of the target host graph components within the union graph. And finally, if both monomorphisms map into the same component graph, we compare on the monomorphisms themselves, represented as vectors.

The first step requires us to fix a total order on the set of graphs \( \mathcal{G} \). The set being finite means we can always find such an order, let thus \( \preceq \subseteq \mathcal{G} \times \mathcal{G} \) be an arbitrary total order. We can proceed with the definition of monomorphism order.
(a) An identity rule with three connected components in $L$. Each component contains a single vertex, either a filled • type or a hollow ○ type.

(b) Three graphs used for the examples below. As example we consider the following total order for these graphs: $g \prec h$.

(c) Two matches onto the graph of $G = (g, h, g)$. The red match it is mapped to a copy of $h$ whereas $L_3$ is mapped to a copy of $g$. As $g \prec h$ this red match violates Condition (1) of Definition 4.4.

(d) Two matches onto the graph of $G = (g, g)$. Both $L_2$ and $L_3$ are mapped to copies of $g$, but in the red match $L_2$ is mapped to a copy with a higher index in $G$ than $L_3$ is, and the red match thus violates Condition (2) of Definition 4.4.

(e) Two matches onto the graph of $G = (g, x)$. The graphs $L_2$ and $L_3$ are mapped to the same graph in $G$, but in the red match the vertex of $L_3$ is mapped to a vertex with a lower index in $x$ than the vertex of $L_2$ is, and the red match thus violates Condition (3) of Definition 4.4.

Figure 2: Examples of order-preserving (blue) and non-order-preserving (red) matches for the rule (a) and graphs in (b). The connected components $L_2$ and $L_3$ of $\bar{L}$ are isomorphic with respect to the rule $p$, and each of Figure (c), (d), and (e) show different examples of how a match can be non-order-preserving, corresponding to the conditions of Definition 4.4.
Definition 4.4 (Monomorphism Order) Let $p = (\tilde{L} \xrightarrow{K} \tilde{L} \xrightarrow{R})$ be a rule with $L = (L_1, \ldots, L_k)$ and $G = (G_1, \ldots, G_n)$ a union graph. Let further $\varphi: L_i \rightarrow G_x$, $\varphi': L_j \rightarrow G_y$ be two monomorphisms mapping two isomorphic rule left side component graphs $L_i \simeq_p L_j$ into component graphs of $G$. We say $\varphi < \varphi'$ if they satisfy one of the following conditions:

1. $G_x \prec G_y$;
2. $G_x = G_y$ and $x < y$;
3. $G_x = G_y$, $x = y$ and there exists $v \in \{1, \ldots, |V(L_i)|\}$, $\varphi(v) < \varphi'(v)$ and for any $u < v$, $\varphi(u) = \varphi'(u)$, under the assumption of representation equivalence, $L_i = L_j$.

The monomorphism order defines a total order on monomorphisms into components of a given union graph. The order is determined by the host component graphs of the monomorphisms (Conditions (1) and (2)), or as a lexicographical vector order on the images of their ordered domains in case the monomorphisms share a common host component graph (Condition (3)).

The monomorphism order naturally extends into a total order $\ll$ on valid partial matches $m, m': L \rightarrow G$ sharing a common domain, $\text{dom}(m) = \text{dom}(m')$. We say the match $m$ is smaller than $m'$, $m \ll m'$ if the vector of monomorphisms for each component of $m'$ is lexicographically smaller than the same vector for $m'$, formally $m \ll m'$ if there exists $i \in \{1, \ldots, |L_i|\}$, such that $m_i < m'_i$ and $m_j = m'_j$ for all $j < i$.

We say that a match extension is order-preserving if the monomorphism mapping the new component is larger than the monomorphisms mapping any preceding rule automorphic rule left side components.

Definition 4.5 (Order-Preserving Match Extension) Let $m: \tilde{L} \rightarrow \tilde{G}$ be a valid partial match of a rule $p = (\tilde{L} \xrightarrow{K} \tilde{L} \xrightarrow{R})$ with $L = (L_1, \ldots, L_k)$ into the graph $\tilde{G}$ with $G = (G_1, \ldots, G_n)$. Let further $\varphi: L_i \rightarrow G_y$ be a monomorphism for some $i \in \{1, \ldots, k\}$, $y \in \{1, \ldots, n\}$ such that the extension $m \cup_y \varphi$ is valid.

We say that the extension $m \cup_y \varphi$ is order-preserving, if for each $j \in \{1, \ldots, i-1\}$ with $L_j \simeq_p L_i$, $m_j < \varphi$.

In simple terms, a partial match extension is order-preserving if the order on the matches of rule isomorphic components $L_i \simeq_p L_j$ of the rule left side graph coincides with their order in $L$.

Note that Definition 4.5 only checks monomorphism order on component graphs $L_j \simeq_p L_i$ in the prefix of the match, $j < i$. This is in line with enumeration of partial matches by Algorithm 1, which already follows the order of the component graphs of $L$. All partial matches enumerated by Algorithm 1 are thus defined on a prefix of $L$. This constraint is reflected in the definition of an order-preserving partial match.

Definition 4.6 (Order-Preserving Partial Match) Let $m: \tilde{L} \rightarrow \tilde{G}$ be a valid partial match of a rule $p = (\tilde{L} \xrightarrow{K} \tilde{L} \xrightarrow{R})$ with $L = (L_1, \ldots, L_k)$ into the graph $\tilde{G}$ such that $m$ is defined exactly on $L_1, \ldots, L_i$ for some $i \in \{0, \ldots, k\}$.

Then $m$ is order-preserving if it can be decomposed into a series of order-preserving extensions $m = \varphi_1 \cup \cdots \cup \varphi_i$.

Examples of order-preserving and non-order-preserving matches are depicted in Figure 2.
Lemma 4.7 Let \( m : \mathbf{L} \to \mathbf{G} \) be a valid partial match of a rule \( p = (\mathbf{L} \xleftarrow{L} K \xrightarrow{R} \mathbf{G}) \) with \( \mathbf{L} = (L_1, \ldots, L_k) \) into the graph \( \mathbf{G} \) with \( \mathbf{G} = (G_1, \ldots, G_n) \).

Then there exists an order-preserving valid partial match \( m' : \mathbf{L} \to \mathbf{G} \) such that \( m \simeq m' \).

Proof: Assuming \( m \) is not already order-preserving, there must exist two indices \( j < i \in \{1, \ldots, k\} \) such that \( L_i \simeq_p L_j \) and \( m_j > m_i \). By definition, \( L_i \simeq_p L_j \) guarantees that there exists a rule automorphism \( \alpha_p = (\alpha_L, \alpha_K, \alpha_R) \) such that \( \alpha_L(L_i) = L_j \), \( \alpha_L(L_j) = L_i \) and \( \alpha_L \) limited to any other component of \( \mathbf{L} \) is an identity. We can use the automorphism to obtain the partial match \( m' = m \circ \alpha_L \) isomorphic to \( m \). As \( \alpha_L \) effectively swaps the monomorphisms mapping \( L_i \) and \( L_j \) we know \( m_j' = m_i \) and \( m_i' = m_j \). For any index \( l \) other than \( i \) and \( j \) we have \( m_l = m'_l \). In particular this holds for any \( l < j \) and thus \( m' \ll m \).

If \( m' \) is order-preserving, we are done. Otherwise, we repeat the procedure for \( m' \), the termination guaranteed by \( \ll \) and the number of valid partial matches of \( \mathbf{L} \) into \( \mathbf{G} \) being finite. \( \Box \)

Following Lemma 4.7, it suffices to enumerate only order-preserving matches to solve Problem 2. Assuming that rule left side component graphs \( L_i \simeq_p L_j \) related by rule automorphism are established in a preprocessing step, checking for order-preserving partial matches reduces to a simple check of the three conditions given in Definition 4.4.

4.2 Minimal Match Extensions

In the previous section, we discussed how order-preserving matches allow us to deal with isomorphisms across multiple rule left side graph components. Recall that by Property 3.5, the way that Algorithm 1 enumerates derivations we are ensured that matches are enumerated only up to the reordering of the host graph. As a result, by only enumerating order-preserving partial matches, we are enumerating derivations up to the reordering of the host graph as well as the reordering of isomorphic connected components of the left graph of a rule.

Order-preserving matches do not, however, consider isomorphic matches obtained from automorphisms that permutes single component graphs of rule nor host graphs.

As an example, assume we are given a valid match \( m : \mathbf{L} \to \mathbf{G} \), where \( \mathbf{L} = (L_1) \) and \( \mathbf{G} = (G_1) \), and an automorphism \( \alpha_{G_1} : G_1 \to G_1 \). As \( m \) consists of a single connected component, clearly it is also order-preserving. However, the match \( m' = \alpha_{G_1} \circ m \) is isomorphic to \( m \).

Given a rule automorphism \( (\alpha_L, \alpha_K, \alpha_R) \), we let \( \alpha_{L,i} \subseteq \alpha_L \) refer to all the automorphisms of \( \alpha_L \) that permutes vertices of the \( i \)th connected component of \( L \).

We refer to such automorphisms \( \varphi_{L,i} \) as local rule automorphisms. For a local rule automorphism \( \varphi_{L,i} \), the match \( m' = m \circ \varphi_{L,i} \) is by definition valid and isomorphic to \( m \), \( m \simeq m' \).

The automorphisms of the component graphs can themselves be exploited to avoid enumeration of such isomorphic (partial) matches. As we are dealing with automorphisms of single component graphs, we can limit ourselves to the context of the monomorphisms \( m_i : L_i \to G_x \) mapping the individual component graphs of \( \mathbf{L} \). We let two monomorphisms be isomorphic if their obvious corresponding partial matches are isomorphic, and re-utilize the monomorphism order from Definition 4.4. As relevant monomorphisms will always map into the same component graph of the host graph, only Condition (3) of Definition 4.4 is applicable. This order corresponds to the lexicographic order on the vectors obtained by applying each monomorphism on their ordered domain. A total order on monomorphisms uniquely identifies the minimal and maximal elements for each subset, in particular all sets of pairwise isomorphic monomorphisms. As we only need one, we choose the minimal element as the unique representative.
Definition 4.8 (Minimal Monomorphism) Let \( p = (\overrightarrow{L} \xrightarrow{K} \overrightarrow{R}) \) be a rule with \( L = (L_1, \ldots, L_k) \) an index and \( G \in \mathcal{G} \) a graph. Let further \( \mathcal{A}_{L,i} \) be a set of local rule automorphisms on \( L_i \), and \( \mathcal{A}_G \) a set of graph automorphisms of \( G \).

Then we say a monomorphism \( \varphi : L_i \rightarrow G \) is minimal with respect to \( \mathcal{A}_{L,i} \) and \( \mathcal{A}_G \) if for each \( \varphi' \in \{ \alpha_G \circ \varphi \circ \alpha_{L,i} \mid \alpha_{L,i} \in \mathcal{A}_{L,i} \text{ and } \alpha_G \in \mathcal{A}_G \} \), \( \varphi \leq \varphi' \).

Definition 4.8 identifies a unique representative for any set of pairwise isomorphic monomorphisms delimited by the local automorphism set \( \mathcal{A}_{L,i} \) and graph automorphism set \( \mathcal{A}_G \). In the general case, when all relevant automorphisms are included, the obtained minimal monomorphisms are pairwise non-isomorphic. While such a configuration might appear to be the obvious choice, in the context of a match, several rule left side component graphs might be mapped into the same host component graph. Consider two indices \( i < j \in \{1, \ldots, k\} \), with \( m_i : L_i \rightarrow G_x \) for some \( x \in \{1, \ldots, n\} \) and a monomorphism \( \varphi : L_j \rightarrow G_x \) into the same host component graph. Then, for the extension \( m \cup \varphi \), we are only interested in such isomorphisms of the monomorphism \( \varphi \) which do not impact \( \text{codom}(m) \). This can be achieved by taking \( \mathcal{A}_{G,x} \) to be the set of all automorphisms \( \alpha_{G,x} : G_x \rightarrow G_x \) such that for each union graph vertex \((x, v) \in \text{codom}(m)\), \( \alpha_{G,x}(v) = v \). We denote such sets of automorphisms preserving \( m \) as \( \mathcal{A}_{G,x}[m] = \{ \alpha_{G,x} : G_x \rightarrow G_x \mid \alpha_{G,x} \text{ is an automorphism and for all } (x, v) \in \text{codom}(m), \alpha_{G,x}(v) = v \} \). No restriction on the local rule automorphisms being necessary.

Definition 4.9 (Minimal Match Extension) Let \( m : \overrightarrow{L} \rightarrow \overrightarrow{G} \) be a valid partial match of a rule \( p = (\overrightarrow{L} \xrightarrow{K} \overrightarrow{R}) \) with \( L = (L_1, \ldots, L_k) \) into a graph \( \overrightarrow{G} \) with \( G = (G_1, \ldots, G_n) \). Let further \( \varphi : L_i \rightarrow G_x \) be a monomorphism for some \( i \in \{1, \ldots, k\} \), \( x \in \{1, \ldots, n\} \) such that the extension \( m \cup \varphi \) is valid.

We say that the extension \( m \cup x \varphi \) is minimal, if \( \varphi \) is minimal with respect to all local rule automorphisms of \( L_i \) and \( \mathcal{A}_{G,x}[m] \).

As an extension of Definition 4.9, we say a valid partial match \( m : \overrightarrow{L} \rightarrow \overrightarrow{G} \) defined on \( L_1, \ldots, L_j \) for some \( j \in \{0, \ldots, k\} \), is minimal if it can be decomposed into a series of extensions \( m = \varphi_1 \cup \cdots \cup \varphi_j \) such that for each \( 1 \leq i \leq j \), the extension \((\varphi_1 \cup \cdots \cup \varphi_{i-1}) \cup \varphi_i \) is minimal.

Lemma 4.10 Let \( m : \overrightarrow{L} \rightarrow \overrightarrow{G} \) be a valid partial match of a rule \( p = (\overrightarrow{L} \xrightarrow{K} \overrightarrow{R}) \) with \( L = (L_1, \ldots, L_k) \) into the graph \( \overrightarrow{G} \) with \( G = (G_1, \ldots, G_n) \).

Then there exists a minimal partial match \( m' : \overrightarrow{L} \rightarrow \overrightarrow{G} \) such that \( m \simeq m' \).

Proof: Assuming \( m \) is not already minimal, there must exist an index \( i \in \{1, \ldots, k\} \) such that the extension \((m_1 \cup \cdots \cup m_{i-1}) \cup m_i \) is not minimal. By definition, there exist a rule automorphism \((\alpha_L, \alpha_K, \alpha_R)\) of \( p \) such that \( \alpha_L \in \mathcal{A}_{L,i} \) is an automorphism of \( L_i \), and a host graph automorphism \( \alpha_G \in \mathcal{A}_{G,x}[m_1 \cup \cdots \cup m_{i-1}] \), which allow us to construct \( \varphi = \alpha_G \circ m_i \circ \alpha_L \) such that \( \varphi < m_i \). We can use the automorphisms to obtain the partial match \( m' = \alpha_G \circ m \circ \alpha_L \) isomorphic to \( m \). \( \alpha_L \) only interacts with the rule left side graph component \( L_i \) and \( \alpha_G \in \mathcal{A}_{G,x}[m_1 \cup \cdots \cup m_{i-1}] \) guarantees that for all \( j < i \), \( m'_j = m_j \). Moreover, \( \varphi = m'_i < m_i \) and thus \( m' \ll m \).

If \( m' \) is minimal, we are done. Otherwise, we repeat the procedure for \( m' \), the termination guaranteed by \( \ll \) and the number of valid partial matches of \( \overrightarrow{L} \) into \( \overrightarrow{G} \) being finite. \( \Box \)

Lemma 4.10 guarantees that for any enumerated valid match we also enumerate at least one isomorphic match which is also minimal. The solution to Problem 2 can thus be obtained by only enumerating minimal matches. To be able to use Lemma 4.10 in conjunction with Lemma 4.7...
stating that order-preserving partial matches are sufficient, we still have to prove that for any valid partial match there exists an isomorphic match that is both order-preserving and minimal.

**Lemma 4.11** Let \( m : \tilde{L} \rightarrow \tilde{G} \) be a valid partial match of a rule \( p = (L \leftarrow K \rightarrow R) \) with \( L = (L_1, \ldots, L_k) \) into the graph \( \tilde{G} \) with \( G = (G_1, \ldots, G_n) \). Then there exists a minimal and order-preserving partial match \( m' : \tilde{L} \rightarrow \tilde{G} \) such that \( m \simeq m' \).

**Proof:** Let us first assume \( m \) is not order-preserving. By Lemma 4.7, there exists an isomorphic order-preserving match \( m' \simeq m \) which is additionally smaller than \( m \) according to the lexicographic order on monomorphism vectors, \( m' \ll m \). Similarly, if \( m \) is not minimal, by Lemma 4.10, there exists an isomorphic minimal match \( m'' \simeq m \) which is also smaller, \( m'' \ll m \).

As \( \ll \) is a total order and there are finitely many partial matches from \( \tilde{L} \) into \( \tilde{G} \), the consecutive application of Lemma 4.7 to obtain an order-preserving match and Lemma 4.10 to obtain a minimal match must necessarily terminate in a fixed-point which is both order-preserving and minimal. \( \square \)

Lemma 4.11 allows us to obtain a solution to Problem 2 by only enumerating order-preserving minimal partial matches. It should be noted, however, that two order-preserving minimal matches can still be isomorphic, as illustrated in Figure 3, making the method introduced in this section a heuristic. This is due to the limitation to host component graph automorphisms which fix the partial match \( m, \mathcal{A}_{G,x} | m \), in Definition 4.9.

In practice, more automorphisms could be considered to obtain less minimal matches, if they commute with a rule automorphism on the match \( m \). In particular, consider a graph automorphism \( \alpha_{G,x} : G_x \rightarrow G_x \) which does not satisfy the condition \( \alpha_{G,x}(v) = v \) for each union graph vertex \( (x, v) \in \text{dom}(m) \), but for which there exists a rule automorphism with \( \alpha_L : \tilde{L} \rightarrow \tilde{L} \) such that for each \( u \in \text{dom}(m) \) such that \( m(u) = (x, v) \) for some \( v \in V(G_x) \), \( m(u) = \alpha_{G,x} \circ m \circ \alpha_L(u) \) as shown in Figure 3. The left side of the commuting rule automorphism \( \alpha_L \) allows us to “reverse” the change introduced to \( m \) by \( \alpha_{G,x} \) thus preserving the isomorphism of the results, \( m \cup_x (\alpha_{G,x} \circ \varphi) = \alpha_{G,x} \circ (m \cup_x \varphi) \circ \alpha_p \).

Detection of automorphisms \( \alpha_{G,x} : G_x \rightarrow G_x \) with a commuting rule automorphism is nontrivial, as it in the general case requires enumeration over all rule automorphisms, including non-local ones. In spirit of avoiding explicit enumeration of all rule automorphisms, whose number can be up to exponentially larger than the number of local rule automorphisms, we limit ourselves to automorphisms that fix the match \( m \). While an efficient discovery of automorphisms \( \alpha_{G,x} \) for which there exists a commuting rule automorphisms might be possible, as our results in Section 6 showcase, using a limited amount of symmetries is sufficient to obtain significant speedup. We thus consider such exploration to be beyond the scope of this contribution.

Finally, the implementation of the symmetry pruning can be done by a minor modification of Algorithm 1. In particular, it suffices to check the minimality and order-preservence of the extension on top of the validity check on lines 15 and 20. Additionally, since the minimality of a monomorphism is always determined with respect to all local rule automorphisms, we can pre-filter the monomorphisms computed on line 4 to only include ones that are minimal with respect of all local rule automorphisms and the identity automorphism of the host graph, \( \mathcal{A}_G = \{ \text{id}_G \} \).

5 Implementation

The algorithm described in the above sections, Efficient Derivation Enumeration (EDE), is implemented in C++ as part of the software framework MØD [5], available online: https://github.
(a) Example with no edges in $\tilde{L}$, and all vertices having the same type. The three connected components of $\tilde{L}$ are pairwise isomorphic.

(b) Example with a single edge in $\tilde{L}$, and two different vertex types. The connected components of $\tilde{L}$ are not isomorphic, but $L_1$ has a symmetry.

Figure 3: Examples where two isomorphic matches are both minimal and order-preserving. The vertices of $\tilde{L}$ are annotated with graph index and vertex index, while the vertices of $G$ are annotated with just their vertex index. In each example one match is $m$ (blue) and the other $m'$ (red). In (a), let $\varphi_1$ and $\varphi_2$ represent the corresponding monomorphisms mapping $L_1$ and $L_2$ into $G_1$, and let $\varphi_3$ (resp. $\varphi'_3$) be the constituent monomorphism for $m$ (resp. $m'$) mapping $L_3$ into $G_1$. Then, we can write each match as a sequence of extensions, $m = \varphi_1 \cup \varphi_2 \cup \varphi_3$ and $m' = \varphi_1 \cup \varphi_2 \cup \varphi'_3$. Clearly, each extension is order-preserving and hence both matches are order-preserving. To see why both are minimal matches, observe that $G_1$ contains the local automorphism $\alpha_{G_1} = \{1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 4, 4 \mapsto 3\}$. However, $\alpha_{G_1} \not\in A_{G_1}[\varphi_1 \cup \varphi_2]$, and both $\varphi_3$ and $\varphi'_3$ are minimal match extensions. Only when the automorphism $\alpha_{G_1}$ is used in conjunction with the automorphism that swaps $L_1$ and $L_2$ one can detect that $m$ and $m'$ are in fact isomorphic. In (b) the same issue occurs, though here the symmetry of $\tilde{L}$ is within a single connected component.
com/jakobandersen/mod/tree/archive/rule-application-21. The MOD integration allows
the algorithm to be easily employed for efficient reaction network construction. Moreover, the
algorithm relies on the solutions of several well-known problems, many already implemented in
MOD.

First, EDE constructs a database of monomorphisms between connected components in the
rule and the given connected input graphs. To enumerate these monomorphisms we employ the
VF2 algorithm [13] as implemented in the Boost Graph library [26]. The reaction network is
constructed by including all discovered, pairwise non-isomorphic derivations. To check for iso-
morphisms between derivations, we use the graph canonicalization framework [9] as implemented
in the GraphCanon library [1], allowing for efficient comparison between two derivations. This
framework is based on the individualization-refinement approach, used by many other the tools,
e.g. nauty [23], Bliss [20, 21] or Traces [24].

The EDE algorithm allows for multiple configurations. We use EDE for the simple version
corresponding to Algorithm 1. The version with symmetry pruning, denoted EDE-S (EDE with
Symmetry pruning), only considers order-preserving and minimal extensions, as discussed in Sec-
tion 4. Conveniently, the individualization-refinement approach constructs the generators of the
automorphism group of the graph being canonicalized as a by-product, allowing EDE-S to use the
generators in match minimality checking with no extra computation cost. Similarly, canonicaliza-
tion can be used to obtain automorphism group generators of rule left side connected components.

To construct the automorphism group itself we employ the Schreier-Sims algorithm [27]. An
implementation of this algorithm is provided by the GraphCanon library [9, 1]. In practice, the
construction of automorphism groups using the Schreier-Sims algorithm can be fairly expensive
with limited gain, especially in the case of graphs with few automorphisms. The algorithm as
presented in Section 4, however, does not require every automorphism of the group to be repre-
sented. While using only a subset of automorphisms may lead to enumeration of some isomorphic
derivations, simple symmetries can be pruned based on the derived generators themselves. Many
of the symmetries commonly encountered in molecule graphs are indeed simple, e.g. hydrogen
atoms of a carbon atom. The abundance of simple symmetries often justifies skipping the costly
automorphism group construction altogether, motivating us to consider one more configuration,
EDE-SS (EDE with Simple Symmetry pruning), that only considers order-preserving and minimal
matches, where minimality is determined using only the generators of the automorphism groups
rather than the automorphism groups themselves.

6 Experiments

Our work is heavily motivated by the need for efficient generation of chemical reaction networks. We
therefore employ reaction network generation as the testbed for comparing EDE against the Partial
Application (PA) approach described in the introduction. For the purposes of the experiments,
we understand PA in the limited scope of the rule application enumeration algorithm published
in [5], which can be directly employed to solve Problem 2. Both EDE and PA are implemented
within the MOD software framework, relying on the same infrastructure for the construction of
the reaction network itself, thus ensuring a fair comparison.

We conduct four different experiments, two using artificial data and two using chemical data. Each experiment consists of the iterative construction of a reaction network $N$ from an initial set of graphs $G$ and a given set of graph transformation rules $R$. In each iteration, EDE or PA are used to enumerate all non-isomorphic derivations for each of the rules in $R$ (Problem 2) using the molecules of the network $N$ for the set $G$. The obtained derivations are then included in $N$, in the
form of reactions linking the components of the host and result graphs, with any newly discovered graphs as new molecules included in $G$ in the next iteration.

To demonstrate the differences in isomorphic match pruning performance, we employ EDE in all three available settings (EDE, EDE-S and EDE-SS). These variations allow us to showcase the effect of early symmetry pruning on different examples.

Similarly, the existing PA implementation also allows for different configurations with respect to isomorphism checking. In particular, the implementation allows checking for isomorphisms between rules obtained by partial application. Thus, if two different partial applications result in isomorphic rules, only one is preserved, effectively pruning isomorphic partial matches. We consider PA in both configurations, as plain PA with no isomorphism checking (analogous to simple EDE), and as PA-I (PA with Isomorphism checking) for the version where isomorphic rules are pruned after each partial application.

All results were obtained using a laptop computer equipped with a 9th generation Intel core i7 processor and 32 GB of RAM. The code has been compiled with the g++ compiler version 8, with optimization level 03.

6.1 Binary Strings

The first experiment is designed to compare performance in a setting with no rule or host graph symmetries. The experiment itself consists of connecting small molecules with a common “backbone” into chains, mimicking the arrangement of amino acids into polypeptide chains. For the sake of simplicity, the experiment follows a minimalistic design. The common backbone of the molecules consists of a single C-O bond, leading to the chains being an interleaving of carbon and oxygen atoms. The use of different atom types allows us to easily distinguish the ends of the chains (C-end and O-end) ensuring there are no symmetries in the constructed molecules.

We consider two initial graphs, shown in Figure 4a and Figure 4b, distinguished by their abstract side chain A or B. From a formal language perspective, arranging these molecules into chains is equivalent to the construction of strings over the binary alphabet $\Sigma = \{A, B\}$. We further consider the experiment in several variations based on the number $k$ of connected components in the rule left side graph, for $k \in \{2, 3, 4\}$. For each $k$ a different rule, $\text{chain}(k)$, is used to build chains by connecting $k$ molecules at a time. An example for $k = 3$ is shown in Figure 4c. For each $k$, the rule contains exactly one component that can match chains of more than one molecule. This allows us to restrict the growth of the chains to only allow extensions of the O-end, and ensuring that there are no symmetries in the rules. In terms of binary strings, each variation is building molecules corresponding to the regular expression $\{A, B\}^n(k-1)+1$ where $n \in \mathbb{N}$ is the number of rule applications.

To curtail the constructed reaction networks, which are theoretically infinite, we limit all variations to molecules of up to 22 atoms, effectively restricting the binary strings to length 7. This restriction explains the drop in the size of the constructed network for increasing $k$ of $\text{chain}(k)$ (Figure 5a) as it limits the number of network construction iterations. Figure 5b then depicts the total number of derivations reported by each algorithm (the number of yields in Algorithm 1) across all iterations. As there are no symmetries, all variations of EDE compute the same number of derivations, corresponding to the size of the reaction network. PA on the other hand, computes multiple copies of the same derivation, owing to the partial application not considering any order on the connected components of the rule left side graphs. We can observe that the isomorphism checking for PA-I manages to eliminate some of the duplicate derivations—however, it does not achieve a higher efficiency as seen by the time taken in Figure 5c. This follows from the number
of partially applied rules growing exponentially with the number of connected components in the rule left side graph, making the pairwise isomorphism checking intractable.

Figure 5c further showcases that the speedup EDE achieves over PA seems consistent with the expected exponential in $k$. Observe also that in spite of having no symmetries to exploit, EDE-S and EDE-SS variations perform on par with EDE. This seems to indicate that the overhead introduced by order-preserving and minimality checks, including the construction of the symmetry groups, is negligible if no symmetries are present.

### 6.2 Binary Trees

Complementing the first example, which focused on asymmetric rules and graphs, our second synthetic data experiment focuses on rules and graphs exhibiting various symmetries instead. The experiment again consists of arranging the initial molecules into large structures. However, instead of simply growing in a linear fashion into a chain, we grow in all directions from a symmetric centerpiece with a four-cycle, shown in Figure 6b. Each carbon atom of the four-cycle then acts as a root of a binary tree of carbon atoms. The tree is constructed by the rule depicted in Figure 6c, consisting of two symmetric components representing methane molecules (Figure 6a) being attached to a leaf of a tree. To prevent merging two trees, the methane molecules are fully specified in the rule. Similarly, to prevent merging three methane molecules, the component representing the tree specifies an extra carbon atom. The constraints imposed by this additional structure in the rule do not only allow us to control the growth of the binary trees, but in the case of the methane molecules also introduce symmetries into the graph components of the left side of the rule.

We grow the reaction network for up to 10 iterations with a strict five minute timeout. The results are summarized in the three graphs in Figure 7, which show the size of the reaction network
Figure 5: Summary results for Binary Strings (Section 6.1). (a) The number of reactions (|E(N)|) and molecules (|V(N)|) in the resulting reaction network. (b) The number of yielded direct derivations during construction of each network. (c) Runtime of each reaction network construction.
after each iteration, the number of derivations computed in each iteration, and the time taken to compute each iteration. The benefit of pruning isomorphic matches, respectively isomorphic partially applied rules, is made clear by the number of computed derivations shown in Figure 7b. The high number of isomorphic derivations enumerated by both EDE and PA is also reflected in the running time, leading to EDE and PA timing out during the 7th, respectively 6th, iteration.

While both EDE-S and EDE-SS successfully compute all 10 iterations, PA-I timed out during the 9th derivation. One can observe that the time taken by PA-I grows more sharply than the number of computed derivations, hinting again at the inefficiency of the pairwise isomorphism checks between the partially applied rules. Another interesting observation is the discrepancy between run times and numbers of computed derivations between EDE-S and EDE-SS. Due to the presence of complex symmetries, which are not captured by the generators, EDE-SS computes significantly more derivations in the 10th iteration \((3 \times 10^4)\) compared to EDE-S \((7 \times 10^3)\). EDE-SS still achieves better runtime, however, as EDE-S suffers from the overhead of constructing the full symmetry group for each graph.

### 6.3 The Formose Chemistry

For our first chemical experiment we select the well-known formose reaction describing the formation of sugars from formaldehyde. The formose reaction was first modeled by graph transformation rules in [4]. Roughly speaking, the formation of sugars consists of polymerisation of formaldehyde molecules. This is a two-step process where a sugar molecule first undergoes keto-enol isomerization (Figure 8c) thereby preparing itself for the aldol addition (Figure 8d). The aldol addition then binds a formaldehyde or another compatible molecule to the enolized sugar. The inverse of both of the rules are also considered.

The performance comparison on the formose reaction gives an interesting baseline, as it is a chemical example which does not play into the strengths of our approach. Indeed, only one out of
Figure 7: Summary results for Binary Trees (Section 6.2). (a) The number of reactions and molecules in the resulting reaction network. (b) The number of yielded direct derivations during construction of each network. (c) Construction time for each reaction network.
the four rules has more than a single connected component in the left side graph.

Similarly to the binary string example (Section 6.1), we want to limit the reaction network size. We therefore consider the formose reaction in multiple variations, with \( n \in \{1, \ldots, 13\} \) being the maximal number of carbon atoms in a single molecule. The results for each \( n \) are presented in Figure 9.

Due to the low number of connected components in the left side graphs of the rules, as well as the relatively small amount of symmetries, the number of computed derivations closely reflects the size of the reaction network and barely differs between all the variations of PA and EDE. This is also reflected in the runtime of the algorithms, Figure 9c. One can observe, however, that despite the similarity in the number of computed derivations, all variations of EDE outperform both PA variations, with the EDE-S trailing behind EDE and EDE-SS due to the overhead from symmetry group construction.

### 6.4 Enzymatic Mechanisms

For the final experiment we examine enzymatic mechanisms modeled by graph transformation rules. In simple terms, a reaction catalyzed by an enzyme can be viewed as a multi-step process, often known as a mechanism, which consists of elementary reaction steps converting the educts to products via a series of intermediate states, possibly temporarily modifying the enzyme itself in the process. This view is adopted by a hand-curated database of enzymatic mechanisms, the Mechanism and Catalytic Site Atlas (M-CSA) [25]. The M-CSA database has served [2] as a foun-
Figure 9: Summary results for the formose chemistry (Section 6.3). The horizontal axis refers to the number of C atoms allowed in a connected component of the product graph. Derivations that produced molecules with more than \( n \) carbon atoms were not considered. (a) The number of reactions and molecules in the resulting reaction network. (b) The number of yielded direct derivations during construction of each network. (c) Construction time for each reaction network.
dation for representing the reaction steps of enzymatic mechanisms by graph transformation rules, making the knowledge in M-CSA executable. As a result, a total of 368 different enzymatic mechanisms have been fully represented by graph transformation rules in [2]. We use these enzymatic mechanism rules in our experiment.

In [2], rules from all mechanisms are pooled together and used to propose new mechanisms for a given reaction. In our experiments, to curtail the combinatorial explosion, we consider each of the 368 mechanisms in isolation. As such, each mechanism is only allowed to use the rules that have been derived from the mechanism itself. To further control the reaction networks growth, we only allow molecules of up to 100 atoms to appear on the product side and impose a runtime limit of 30 minutes per mechanism. Under these conditions there are eight instances in which PA cannot finish, twice due to the time limit and six times due to running out of available memory, limited to 32 GB by the hardware of the machine used for the experiments. Among all variations of EDE, three instances fail to finish, two due to time and one due to memory. In the following, we focus on the 360 enzymatic mechanisms for which all configurations of both algorithms finish within the allocated resource limits.

Table 1 lists runtime statistics of all tested algorithms on all enzymatic mechanisms as a relative percentage of the running time of PA. In the vast majority of cases, the produced reaction network is relatively small, and all algorithms finish the computation within a second. The run time of PA exceeds one second in only 20 instances. Irrespective of the total run time, all three variations of EDE outperform both PA and PA-I on almost all enzymatic mechanisms considered. The exception being EDE-S, which performs slower than PA on a few examples. All such instances are on the small reaction networks, the difference between PA and EDE-S being in the order of hundredths of a second.

On the other hand, Table 2 lists running times as a relative percentage of the running time of PA on the 20 mechanisms that take more than a second to compute. This comparison makes it clear that all variations of EDE perform on average four to almost eight times faster than PA, EDE-SS finishing consistently at least twice as fast for reaction networks of sufficient complexity.

Table 2 suggest EDE-SS as the fastest alternative. This result is not surprising as many of the symmetries found in molecules are relatively simple, such as symmetries between hydrogen atoms bonded to the same carbon atom, and can therefore be captured by a single generator. More complex symmetries resulting from a composition of generators are far more rare, making the time saved by not constructing the symmetry groups overshadow the time saved by avoiding enumeration of all matches isomorphic by means of complex symmetries.

Finally, we highlight the 20 enzymatic mechanisms that take the longest time to compute for PA. The performance of all algorithms as a relative percentage of the PA run time is given individually for each of the 20 mechanisms in Figure 10a. To avoid the issue of scale, the same data for only variations of EDE is given in Figure 10b. One can remark that the performance varies greatly from one mechanism to the other. An interesting case is the mechanism 131_1, which contains several rules with at least seven connected components in the left side graph. As intended, EDE has little trouble with applications of such rules, while the pairwise isomorphism checking in PA-I suffers from the exponential explosion in the number of partially applied rules.

7 Conclusion

We design a new efficient algorithm for enumerating the matches of graph transformation rules in the application area of chemical reaction networks. The proposed algorithm is designed to improve processor time when rules have multiple connected components on the left side. It also allows
Table 1: Average, minimum, and maximum recorded runtime across all enzymatic mechanisms for which all versions of EDE and PA finished. All values are given as a percentage of the plain PA runtime.

|     | mean | min | max   |
|-----|------|-----|-------|
| EDE | 27.39| 0.44| 90.55 |
| EDE-S| 24.35| 0.05| 106.12|
| EDE-SS| 17.89| 0.04| 86.55 |
| PA  | 100.00| 100.00| 100.00|
| PA-I | 96.13| 13.55| 1501.56|

Table 2: The same comparison as in Table 1, however, only for the 20 mechanisms with non-trivial runtime (more than one second for the plain PA). All values are given as a percentage of the plain PA runtime.

|     | mean | min | max   |
|-----|------|-----|-------|
| EDE | 25.06| 0.44| 65.87 |
| EDE-S| 15.60| 0.05| 58.90 |
| EDE-SS| 12.94| 0.04| 49.90 |
| PA  | 100.00| 100.00| 100.00|
| PA-I | 233.37| 18.56| 1501.56|
Figure 10: Summary results for enzymatic mechanisms (Section 6.4). The horizontal axes refer to the corresponding mechanism entry and proposal in the M-CSA database. (a) The relative runtime of all variants, scaled to the plain PA variant. (b) The relative runtime of EDE variants scaled to the plain PA variant.
for trading off time for space by means of a pre-computation of the monomorphisms between rule components and potential host graphs. While the algorithm is motivated specifically by rules with multiple connected components on the left side, our results show that the algorithm remains highly competitive even when rules with few connected components, e.g., the formose reaction in Section 6.3, are utilized.

We further propose an alternative version of the algorithm which in addition to the validity of the matches also checks their uniqueness with respect to the resulting derivation isomorphisms. This is achieved by the introduction of a total order on the partial matches based on the symmetry groups of the host molecules as well as the rules themselves in order to determine if a smaller partial match is available. As a result, a partial match that does not lead to any non-isomorphic derivations can be discarded early, already during the enumeration, thereby saving resources. We not only demonstrate that the new algorithm vastly outperforms existing methods, which suffer from the combinatorial explosion in the number of derivations when checking for isomorphisms, but also that not all symmetries are necessary to obtain a significant speedup. This allows us to avoid the expensive operation of computing the automorphism groups of the graphs involved, using only generators of said groups instead.

References

[1] Jakob L. Andersen. GraphCanon repository on GitHub, 2021. http://github.com/jakobandersen/graph_canon.

[2] Jakob L Andersen, Rolf Fagerberg, Christoph Flamm, Walter Fontana, Juraj Kolčák, Christophe V F P Laurent, Daniel Merkle, and Nikolai Nøjgaard. Graph transformation for enzymatic mechanisms. Bioinformatics, 37(Supplement_1):i392–i400, 07 2021.

[3] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. 50 shades of rule composition. In François Fages and Carla Piazza, editors, Formal Methods in Macro-Biology, volume 8738 of Lecture Notes in Computer Science, pages 117–135. Springer International Publishing, 2014.

[4] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. Generic strategies for chemical space exploration. International Journal of Computational Biology and Drug Design, 7(2/3):225 – 258, 2014.

[5] Jakob L Andersen, Christoph Flamm, Daniel Merkle, and Peter F Stadler. A software package for chemically inspired graph transformation. In International Conference on Graph Transformation, pages 73–88. Springer, 2016.

[6] 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, 16(2):510–523, 2017.

[7] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. An intermediate level of abstraction for computational systems chemistry. Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 375(2109), 2017.
[8] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. Rule composition in graph transformation models of chemical reactions. *MATCH, Communications in Mathematical and in Computer Chemistry*, 80(3):661–704, 2018.

[9] Jakob L. Andersen and Daniel Merkle. A generic framework for engineering graph canonization algorithms. *J. Exp. Algorithmics*, 25(1), April 2020.

[10] Nicolas Behr and Jean Krivine. Compositionality of Rewriting Rules with Conditions. *Compositionality*, 3, April 2021.

[11] Gil Benkő, Christoph Flamm, and Peter F Stadler. A graph-based toy model of chemistry. *Journal of Chemical Information and Computer Sciences*, 43(4):1085–1093, 2003.

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

[13] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[14] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation–part i: Basic concepts and double pushout approach. In *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations*, pages 163–245. World Scientific, 1997.

[15] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taenthzer. Fundamentals of algebraic graph transformation, 2006.

[16] Walter Fontana. Algorithmic chemistry. Technical report, Los Alamos National Lab., NM (USA), 1990.

[17] M Grom, G Stavber, P Drnovšek, and B Likozar. Modelling chemical kinetics of a complex reaction network of active pharmaceutical ingredient (api) synthesis with process optimization for benzazepine heterocyclic compound. *Chemical Engineering Journal*, 283:703–716, 2016.

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

[19] 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, 07 2016.

[20] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 135–149. SIAM, 2007.

[21] Tommi Junttila and Petteri Kaski. *Conflict Propagation and Component Recursion for Canonical Labeling*, pages 151–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[22] Minoru Kanehisa, Susumu Goto, Yoko Sato, Miho Furumichi, and Mao Tanabe. Kegg for integration and interpretation of large-scale molecular data sets. *Nucleic acids research*, 40(D1):D109–D114, 2012.
[23] Brendan D. McKay. Practical graph isomorphism. In Congressus Numerantium, volume 30, pages 45–97. Utilitas Mathematica Pub. Incorporated, 1981.

[24] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism II. Journal of Symbolic Computation, 60:94–112, 2014.

[25] António J M Ribeiro, Gemma L Holliday, Nicholas Furnham, Jonathan D Tyzack, Katherine Ferris, and Janet M Thornton. Mechanism and catalytic site atlas (m-csa): a database of enzyme reaction mechanisms and active sites. Nucleic acids research, 46(D1):D618–D623, 2018.

[26] Boris Schäling. The boost C++ libraries. Boris Schäling, 2011.

[27] Charles C. Sims. Computation with permutation groups. In Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation, SYMSAC ’71, page 23–28, New York, NY, USA, 1971. Association for Computing Machinery.

[28] Maneesh K Yadav, Brian P Kelley, and Steven M Silverman. The potential of a chemical graph transformation system. In International Conference on Graph Transformation, pages 83–95. Springer, 2004.

[29] Yuk L Yung and William B DeMore. Photochemistry of planetary atmospheres. Oxford University Press, 1998.