Knowledge representation and curation in hierarchies of graphs

Ievgeniia Oshurko

Numéro National de Thèse : 2020LYSEN024

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée par
l'Ecole Normale Supérieure de Lyon

Ecole Doctorale N° 512
École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 06/07/2020, par:

Ievgeniia OSHURKO

Knowledge representation and curation in hierarchies of graphs

Représentation et curation des connaissances dans les hiérarchies de graphes

---

Devant le jury composé de :

Fages, François	Directeur de Recherche, INRIA Saclay	Rapporteur
Winskel, Glynn	Professeur, University of Cambridge	Rapporteur
König, Barbara	Professeure, Universität Duisburg-Essen	Examinatrice
Champin, Pierre-Antoine	Maître de conférences HDR, IUT Lyon1	Examinateur
Harmer, Russ	Chargé de Recherche CNRS, ENS de Lyon	Directeur de thèse

Abstract

The task of automatically extracting insights or building computational models from knowledge on complex systems greatly relies on the choice of appropriate representation. This work makes an effort towards building a framework suitable for representation of fragmented knowledge on complex systems and its semi-automated curation—continuous collation, integration, annotation and revision.

We propose a knowledge representation system based on hierarchies of graphs related with graph homomorphisms. Individual graphs situated in such hierarchies represent distinct fragments of knowledge and the homomorphisms allow relating these fragments. Their graphical structure can be used efficiently to express entities and their relations. We focus on the design of mathematical mechanisms, based on algebraic approaches to graph rewriting, for transformation of individual graphs in hierarchies that maintain consistent relations between them. Such mechanisms provide a transparent audit trail, as well as an infrastructure for maintaining multiple versions of knowledge.

We describe how the developed theory can be used for building schema-aware graph databases that provide schema-data co-evolution capabilities. The proposed knowledge representation framework is used to build the KAMI (Knowledge Aggregation and Model Instantiation) framework for curation of cellular signalling knowledge. The framework allows for semi-automated aggregation of individual facts on protein-protein interactions into knowledge corpora, reuse of this knowledge for instantiation of signalling models in different cellular contexts and generation of executable rule-based models.

Résumé

L'extraction automatique des intuitions et la construction de modèles computationnels à partir de connaissances sur des systèmes complexes repose largement sur le choix d'une représentation appropriée. Ce travail s'efforce de construire un cadre adapté pour la représentation de connaissances fragmentées sur des systèmes complexes et sa curation semi-automatisé.

Un système de représentation des connaissances basé sur des hiérarchies de graphes liés à l'aide d'homomorphismes est proposé. Les graphes individuels représentent des fragments de connaissances distincts et les homomorphismes permettent de relier ces fragments. Nous nous concentrons sur la conception de mécanismes mathématiques, basés sur des approches algébriques de la réécriture de graphes, pour la transformation de graphes individuels dans des hiérarchies qui maintient des relations cohérentes entre eux. De tels mécanismes fournissent une piste d'audit transparente, ainsi qu'une infrastructure pour maintenir plusieurs versions des connaissances.

La théorie développée est appliquée à la conception des schémas pour les bases de données orientée graphe qui fournissent des capacités de co-évolution schémas-données. Ensuite, cette théorie est utilisée dans la construction du cadre KAMI, qui permet la curation des connaissances sur la signalisation dans les cellules. KAMI propose des mécanismes pour une agrégation semi-automatisée de faits individuels sur les interactions protéine-protéine en corpus de connaissances, la réutilisation de ces connaissances pour l'instanciation de modèles de signalisation dans différents contextes cellulaires et la génération de modèles exécutables basés sur des règles.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, Russ Harmer, for his guidance and support throughout the thesis. I thank him for his rigorous eye, good humor, patience and open-mindedness, for being attentive and respectful to my opinions during our numerous discussions that made this thesis possible.

I would also like to thank Angela Bonifati for our fruitful collaboration, for bringing new challenges and helping us in building a bridge between our communities.

I am grateful to my collaborators, Sébastien Légaré and Yves-Stan Le Cornec for our interesting discussions, their help and valuable advice. I thank the students who worked with me, Florien, Tristan, Redwan and Thomas.

I would like to thank Marie Narducci for being so kind and always helping me with a radiant smile, and Daniel Hirschkoff for his *umorismo affascinante*.

I want to thank my family and especially my mom for being there and for being the woman who empowers me throughout my life.

I am grateful to Raimon for giving me amazing support in the best and the worst times, for always being so kind and also reminding me to be kind.

Finally, I am grateful to my friends Sasha, Lorenza, Farid, Sam and Jacob.

Contents

Chapter 1

Introduction

Modern technologies allow us to generate and store virtually unlimited amounts of data containing knowledge of different provenance, from records of human activity to high-throughput experimental results. Extracting implicitly present knowledge from this data and reasoning about it requires advanced processing and analysis techniques that go beyond the capacities of a human brain and are required to be automatized. To understand the emergent dynamics of complex systems, such as social or biological systems, extracting implicit knowledge often means building computational models. In domains such as biology building accurate and reliable computational models is of crucial importance to the understanding of diseases or the development of new drugs. In this abundance, however, even explicitly present knowledge is often represented in a form not suitable for automated processing and analysis, e.g. natural language, unstructured or semi-structured documents. Moreover, this knowledge is often fragmentary, dispersed over multiple sources and inconsistent, therefore, needs to be assembled and curated.

This thesis makes an effort to build a knowledge representation framework particularly suitable for representation of knowledge on complex systems and its semi-automated curation. It is built upon three pivots: the first one consists in development of the mathematical theory for knowledge representation and update using hierarchies of graphs, the second one focuses on the application of the theory for building schema-aware graph databases, and, finally, the third one treats the problem of curation of biological knowledge on cellular signalling. The rest of this chapter provides a brief introduction to all three topics elaborating on their motivation and context.

1.1 Knowledge representation

The field of *knowledge representation* (KR) studies the approaches to representation of knowledge that facilitate building of intelligent systems [6, 30]. Such systems are often referred to as *knowledge-based systems*. Typically, the represented knowledge consists in some domain-specific description of the world and the intelligent systems help to perform inference of some not explicitly present knowledge. In the dicussion of what is a KR, Randall et al. [30] articulate five roles played by a KR, which give a flavour of what we actually mean by a representation:

- *Surrogate:* a representation is always a substitute for the represented thing itself. The correspondence between the surrogate and what it refers to gives the semantics for the

representation. This implies that the representation is always an approximation and, therefore, is inaccurate and may contain some representation artifacts.

- *Set of ontological commitments:* choosing a particular representation we choose a set of ontological commitments that define the aspects of reality that we include and the ones we ignore in our representation. This focuses our attention on aspects that we believe to be relevant. Due to its overwhelming complexity, to be able to efficiently reason about the world, we are obliged to make this choice.

- *Fragmentary theory of intelligent reasoning:* a representation implicitly provides a theory of intelligent reasoning, it defines the set of inferences that the representation allows. Therefore, the choice of a representation intrinsically involves the choice of the nature of intelligent reasoning.

- *Medium for efficient computation:* reasoning using machines is a computational process, therefore the representation of knowledge should offer computational efficiency of reasoning.

- *Medium of human expression:* a representation provides means for expressing and communicating knowledge about the world to the machine (or to other humans).

Application domains of KR include software engineering, natural language processing, database management, etc. Basic KR tools include logic, frames, semantic networks, rules and so on. Logic-based KR systems exploit the idea that the predicate calculus can be used to capture knowledge about the world and verification of logical consequences—to reason using this knowledge. Non-logic based systems (frames, semantic networks and rules) are often graphical, they represent knowledge with some intuitive data structures and provide some specially tailored ad hoc reasoning mechanisms [6].

In this work we focus on building a KR system that allows for representation of knowledge about *entities* and their *relationships* on different abstraction levels. To elaborate, our KR system should provide means to:

- express entities and their relationships (where entities could represent some kinds, objects, agents, events);

- equip entities and relationships with key/value attributes (that can express, for example, states, properties, qualities of entities and relations);

- divide a knowledge corpus into distinct fragments;

- relate entities (and potentially relationships) from different fragments of a knowledge corpus (for example, to express relations 'is the same as' or 'is an instance of').

Therefore, the developed KR system, presented in Section 2.2, is structured as a *hierarchy of graphs* related to each other with graph homomorphisms. In this work we focus mainly on the update aspects of KR and almost do not treat the question of reasoning using the knowledge represented with hierarchies of graphs. This is partially due to the fact that conceivable modes of reasoning over such knowledge are highly application- and domain-specific. The structure of the system makes it well-adapted for reasoning on hierarchical relations of entities (such as subsumption) and reasoning based on graph theory techniques (such as search for specific

graphical patterns and motifs, analysis of topological properties of the underlying graphs). For example, the kind of 'analytics' that can be applied to such knowledge is tangential to the querying capabilities of graph databases discussed in more details in Subsection 1.2.1.

1.1.1 Related work on knowledge representaion

In this subsection, to give the reader an idea of classical approaches developed in the KR domain, we discuss some related representation models, try to compare and place our system in their context when possible.

Semantic networks (nets). Semantic nets use network-like structures to represent knowledge. Nodes and links usually represent concepts and relations among them [6, 74]. Nodes of the same semantic net can be used to represent both classes of objects and individual objects themselves. Moreover, nodes can be equipped with attributes describing properties of concepts (for example the concept 'bird' can have properties 'can fly' and 'has feathers'). Concepts can be related with special 'IS-A' edges defining subclass-superclass relations, and properties of superclasses are by default inherited by subclasses (for example, the concept 'duck' is connected with an 'IS-A' edge to 'bird', therefore, 'duck' inherits the properties 'can fly' and 'has feathers', and it can have its own properties, for example 'is brown'). Both semantic nets and our KR system use graphs as the main underlying data structure, however, the notions of hierarchy and inheritance of properties differ substantially. First of all, the most reasonable translation of 'IS-A' edges between concepts to our KR system would correspond to separation of concepts on different abstraction levels into separate graphs providing homomorphisms between them (which is not always possible). Then properties of concepts could be translated to attributes of nodes in our graphs (for example, a property 'can fly' to an attribute 'can fly: True'). Such a translation would, however, still break down as in our system attributes of a superclass node define a set of *allowed* attributes of a subclass node (a person can have a name, Bob is a person, he can have a name), while properties in semantic nets define *necessary* properties (every person has a name, Bob is a person, he has a name). This renders the correspondence of inheritance in two systems invalid. Attributes of superclass nodes in our KR system provide a specification of their instances by giving all possible attribute keys/values that the latter are allowed to have.

Frames. Developed in the seventies and greatly inspired by psychology and linguistics, frame-based KR systems are based on the idea that whenever we find ourselves in a new situation we retrieve a prototype situation from our memory and adapt it to fit the newly occurred one. The main data structure of this KR system is called *frame*, represents a stereotypical situation and contains some information on how to use the frame. A frame is presented by two levels: the first one represents things that are always true in the respective situation and is fixed, the second one contains empty 'slots' to be instantiated with new knowledge. Frames are related into *frame-systems* that can be used to represent taxonomies, actions, cause-effect relations or changes in the modellers view-point. Structural properties of these frame systems provide means for various inferences [42, 69].

Description logic (DL). DL is a language that allows to represent knowledge about concepts, individual objects, roles and their relationships. A DL knowledge base consists of two kinds of statements: *TBox* and *ABox*. *TBox* statements provide a terminology and contain declarations of properties of concepts. *TBox* statements allow performing *subsumption* reasoning,

i.e. checking whether a subsumer concept is more general than a subsumee. *ABox* statements define assertions about individuals and can be used to express membership and role assertions. The main reasoning task in *ABox* is *instance checking*, i.e. whether an individual object is an instance of some concept [6]. Similarly, to 'IS-A' edges of semantic nets, *TBox* statements can be encoded into our KR system by separating concepts on different abstraction levels into separate graphs and providing homomorphisms between them.

Ontologies. The general term *ontology* in computer science refers to a formalized representation of a set of entities, processes, attributes and relations that constitute a particular domain-specific world-view [82]. Ontologies usually include a vocabulary of terms and definitions of these terms. Their main goal is to facilitate communication and inter-operability of knowledge between different agents (both people and software systems). There exist a number of ontology definition languages, among which the triad of *Resource Description Framework (RDF)*, *RDF Schema (RDF Schema)* [31] and *Web Ontology Language (OWL)* [67] are the most well-known. RDF is a data model allowing representation of facts about entities and their relations in the form of triples 'subject-predicate-object'. RDF Schema defines a vocabulary for describing properties and classes of entities represented with RDF triples as well as class hierarchies. OWL allows to define a layer of semantics over RDF and RDFS, it allows to represent different relations between classes, cardinality, equality, etc. For example, OWL allows to express statements such as 'A is the same as B', 'the class A is disjoint from the class B', 'if A is a friend of B, then B is a friend of A'. Our KR system can be used to define an ontology and at the current stage of its development it does not provide a formalized OWL-like semantics (however, some of its fragments can be still adapted, such as 'A is the same as B'), therefore can be directly compared only to the couple RDF and RDFS. Similarly, to the previous KR approaches 'IS-A' relations between a subclass entity and a superclass entity can be encoded with homomorphisms between different graphs of our system.

Other related models. Predominantly used in relational database design the *entity-relashionship model (ER)* [16] provides a unified view on data as a set of entities and relationships. Similarly *Unified Modeling Language (UML)* [76], widely used for object-oriented design of software systems, represents various system components (activities, classes of objects, interfaces) with nodes and their relations (interaction of components, their composition, inheritance) with edges. These models are not usually considered as tools for KR, but rather specialized modelling tools. However, implicitly they do represent expert knowledge on the design and architecture of underlying databases and software systems. With respect to our KR system, the 'instance-of' relations in ER or UML intuitively constitute homomorphisms between different graphs (they represent relations that cross from one abstraction level to another).

1.1.2 Knowledge curation

To formulate the principal motivation behind this thesis, we need to define the notion of *knowledge curation* that will be central to all the following parts of the thesis. By curation we mean the dynamical process of continuous collation, integration, annotation and revision of some domain-specific knowledge.

Therefore, the main motivation for our work is to design a curation framework for knowledge represented with hierarchies of graphs. Namely, we would like to design rigorous mechanisms for the update of individual knowledge fragments that would preserve consistent relations between

them. Accommodated knowledge is expected to be frequently updated by potentially different curators, whose updates are not necessarily consistent. Therefore, our curation framework should provide a traceable history of updates—an *audit trail*, as well as an infrastructure for maintaining multiple versions of a knowledge corpus. A transparent audit trail would not only provide an insight on the history of knowledge updates, but would also allow the curator to rollback to an arbitrary point in this history, which is extremely useful, when, for example, trying to fix an erroneous update or their sequence.

The update of a knowledge fragment corresponds to a transformation of the corresponding graph in the hierarchy. The transformations are based on *sesqui-pushout (SqPO) graph rewriting*, an algebraic rewriting approach based on category theory [20]. Applied to a specific graph in a hierarchy some transformations are required to be *propagated* to other graphs to preserve the consistency of the hierarchy. The system is described in detail and formalized in Section 2.2.

We design an audit trail for updates in hierarchies of graphs based on the ideas underlying modern version control (VC) systems. We exploit the *reversibility* and *composition* of SqPO rewriting to adapt the main notions of VC to graph-like structures. Thus, the developed audit trail allows to efficiently represent the history of updates for individual graphs and graph hierarchies, revert the changes, maintain and semi-automatically merge diverged versions of knowledge. Further discussion of the audit tail system designed as a part of our KR framework can be found in Sections 2.1.4 and 2.2.10.

The Python library `ReGraph`[1] developed in the frame of this thesis allows the user to build arbitrary representations based on hierarchies of simple graphs with attributes, perform knowledge updates and maintain an audit trail. A more detailed discussion on `ReGraph` is presented in Section 2.3.

1.1.3 Related work on graph hierarchies and rewriting

In this subsection we provide a brief overview of the work related to the construction of graph hierarchies, as well as the rewriting techniques used for their update and audit.

Slice categories and typed graphs. Slice categories (see the definition in Appendix A.2.6) are often used to formulate the notion of *typed graphs*, i.e. graph objects typed by other graphs via homomorphisms. Rewriting in such slice categories is typically formulated in terms of transformations that respect the fixed typing object (called type graph), i.e. whose result is guaranteed to be an object of the underlying slice category [19, 38]. For instance, a transformation $G \rightsquigarrow G'$ of a typed graph $G \to T$ results in $G' \to T$.

Our work generalizes such typing to a hierarchy, where every object is typed by all its descendant objects and the typing by different descendants is required to be consistent (more details on this account will follow in Section 2.2). Moreover, we formulate the rewriting in hierarchies which does not only guarantee that the resulting object stays well-typed by its descendants, but also allows for their dynamical transformation. For example, a transformation $G \rightsquigarrow G'$ of a typed graph $G \to T$ may result in $G' \to T'$, where T' represents the result of such a dynamical transformation of the type graph T (or propagation to T). This approach is related to the change-of-base functor familiar from algebraic topology and to its right adjoint whose existence characterizes pullback complements [33].

[1] `https://github.com/Kappa-Dev/ReGraph`

Graph rewriting. Algebraic approaches to graph transformation provide a mathematical framework allowing for simple proofs, not specifically tailored for graph structures, but working for any objects satisfying some structural requirements. In addition, they provide some interesting results on concurrency, parallelism and distribution analysis [21]. The most well-studied algebraic graph rewriting approaches are the double-pushout (DPO, [37]), the single-pushout (SPO, [64]) and the sesqui-pushout (SqPO, [20]) approaches. As is suggested by their names, the three approaches are defined in terms of different categorical constructions (such as pushouts or pullback complements, formally defined in Appendix A). These constructions provide us with tools for propagation of rewriting in hierarchies and construction of transparent audit trails.

The choice of the SqPO approach as the main graph transformation technique is motivated by the following two factors. First of all, unlike the SPO or the SqPO approach, DPO rewriting does not allow for 'deletion in unknown context', i.e. it allows to delete a node only if all of its incident edges are explicitly removed. The latter condition is often called the 'dangling condition'. Formally, a DPO rewrite consists in gluing two pushouts (therefore, the name double-pushout), and the 'dangling condition', roughly speaking, guarantees that one of this pushouts can be constructed. The SPO approach, on the other hand, allows to perform 'deletion in unknown context'. However, unlike the SqPO approach, it does not allow to clone graph elements, while, in the applications of our interest, we would like to be able to perform both 'deletion in unknown context' and cloning of graph elements.

Propagation of transformation. As we have previously discussed, in the classical theory of typed graph rewriting, the result of transformation stays well-typed by the fixed typing object. In our work we propose an approach that allows for dynamical transformation of the typing object. Moreover, our propagation framework allows to rewrite the typing object itself and dynamically transform the typed object. For instance, for a typed graph $G \to T$, a transformation $T \rightsquigarrow T'$ may result into $G' \to T'$, where G' represents a dynamical transformation of the typed object (or propagation to G). A coupled rewriting of both typed and type graphs was proposed in [66] for the DPO approach. The propagation techniques proposed in this work are canonical, i.e. are given by some categorical constructions having universal properties, such as a pushout or a pullback. We generalize this approach to the SqPO rewriting and provide an entire spectrum of possible coupled transformations: from strict (where the fixed type or typed object are respected) to canonical (where the results of the coupled transformation has a universal property). Subsections 2.2.2 and 2.2.3 describe the proposed propagation techniques in more detail.

Audit trail. The designed audit trail system for individual graphs and graph hierarchies provides features similar to VC systems used in software development. Such systems typically provide a control over different versions of software's source code distinguishing sets of updates into atomic operations—commits. These commits are stored in the revision graph representing the temporal (partial) order of commits. Storing a state of a software project at the time of every commit can be extremely heavy and resource consuming. Therefore, a typical VC system stores only the current state of a project, while the history of commits stores its states at different times using delta compression, i.e. storing only the difference from a time step to the next one—a delta [73]. We adapt the notion of a delta to the graphical structures by representing graph transformations with rewriting rules and their matches (see more details in Section 2.1). For this representation to be sound, the performed transformations are required

to be reversible, i.e. the underlying rewriting rules and their matches should explicitly encode these transformations. Reversibility of the SqPO rewriting was studied in [29] and is discussed in more details in Subsection 2.1.2. We extend this work by formulating the notion of a rule hierarchy that expresses reversible rewriting and propagation in a hierarchy of graphs.

In addition, the proposed audit trail allows to efficiently represent multiple diverged versions of the underlying object. Such versions are represented using deltas, i.e. by maintaining only the current version of the object, while representing the other versions with deltas from this object. In order to express the divergence of versions (the transformations of the underlying objects), we need to provide means for composing deltas. This is done by defining the composition of consecutively applied rewriting rules. The construction of such a composition for two consecutive rule applications (often called concurrent synthesis) is closely related to the question of concurrency of SqPO rewriting [21]. There exists an extensive general theory of concurrency for different graph rewriting approaches [36, 21], including SqPO rewriting. However, in this thesis we focus exclusively on its aspects related to the concrete question of composition of two consecutive applications of SqPO rewriting rules. This question has been studied in [7] for linear SqPO rules (rules that do not clone or merge elements). In this work we extend this approach to any SqPO rules, where the application of the first rule is reversible (see the discussion presented in Subsection 2.1.3). Moreover, we generalize the composition of rewriting to transformations operating on hierarchies of graphs.

Finally, the developed audit trail allows merging two diverged versions of the object. Such merging is performed by gluing the versions by their 'common denominator', either provided by some canonical constrution or user-defined. The questions of versioning, merging and detection of such 'common denominator' are closely related to the works presented in [80] and [34], which study versioning based on the DPO approach.

1.2 Graph databases

In the previous section we have briefly discussed what is a knowledge representation (KR). While designing and building a knowledge-based intelligent system it is not enough to just select an appropriate KR, it is also necessary to choose a technology that would enable us to store the *knowledge base (KB)*. Usually such a technology is a database management system (DBMS) that allows storing persistently and querying our knowledge.

Remark 1.2.1. What is the difference between a database and a knowledge base? Some databases accommodate knowledge bases, but not all of them. At the same time, some knowledge bases are stored in databases, but not all of them (for example, some of them are stored in collections of unstructured documents). Databases usually refer to organized collections of data, which can be modified and accessed by some DBMS, while knowledge bases often refer to collections of highly interconnected data (usually representing knowledge of a higher level), they are often equipped with some inference engine and are associated to some intelligent systems.

The *property graph data model* underlies most of the modern graph database technologies [9]. Using this model has been initially envisaged for the development of knowledge bases built upon our KR system. Interestingly, we have discovered that not only this data model can be adequately used in our framework, but that the model itself can be enriched by the concepts and techniques provided in our system. In this section we will briefly introduce the property graph data model and discuss some languages designed to query graph databases. We will

also introduce the notions of schemas for property graphs, schema validation and data/schema co-evolution as an interesting application of our KR framework.

1.2.1 Property graphs

Property graphs (PGs) allow to represent data with sets of nodes and relationships. Relationships are directed edges connecting at most two nodes, both nodes and relationships can be equipped with key/value properties. Nodes and relationships can be assigned with sets of labels (that can be used, for example, to group them into sets). Modern graph database technologies (such as Neo4j, Oracle PGX, SAP HANAGraph, Redis Graph, etc.) are predominantly based on the PG data model.

The querying functionality of graph databases exploits the ability to represent complex relations between entities. The most common queries include finding direct and indirect connections between entities, finding various subgraph patterns and so on [9]. One of the most powerful graph query languages is called Cypher (originally implemented as part of the Neo4j graph database) and its open-source counterpart openCypher [43]. Cypher has an intuitive ASCII-art syntax that allows to visually specify graph patterns for querying and modifying data. Figure 1.1 shows a small example of a Cypher query.

```
MATCH (p:Post)-[:HAS_CREATOR]->(u:Person)<-[:HAS_CREATOR]-(c:Comment)
WHERE (c)-[:REPLY_OF]->(p)
RETURN u, p, c
```

Figure 1.1: Example of a Cypher query. This pattern matching query finds comments on posts whose author is the same person.

A graph database is equipped with an engine that performs planning and execution of queries. Query execution is decomposed into different elementary operators which together form a tree structure called an *execution plan*[2]. Evaluation of an execution plan starts from the leaves of the corresponding tree. Such leaves do not have input nodes and typically implement some operations of direct access to the data storage (for example, scanning of nodes by a label).

1.2.2 Building a hierarchical knowledge base with PGs

In the previous section we have briefly introduced the main idea behind our KR system. Its main representation units are graphs whose nodes and edges can be equipped with key/value attributes. The system allows to fragment knowledge into multiple graphs and relate them with graph homomorphisms creating a hierarchical structure. We would like to be able to 'encode' our system using the PG data model.

To find such an encoding we need to address a couple of challenges. The first challenge consists in encoding the structure of graphs with attributes using PGs, which is fairly easy to solve by establishing a one-to-one correspondence between nodes and nodes, edges and relationships, attributes and properties of respectively our graphs and PGs.

The next challenge consists in finding the representation of a hierarchical structure of a knowledge base (i.e. how to separate knowledge into multiple graphs and relate them with homomorphisms). Most of the modern graph database technologies do not allow to store multiple PGs at the same time. Therefore, we need to find a way to encode the entire knowledge base in

[2] `https://neo4j.com/docs/cypher-manual/current/execution-plans/`

a single PG. To do this we use the capability of PGs' node labels to separate nodes into different sets. We can assign a unique identifier to every individual graph in our knowledge base and label its nodes with this identifier.

We now need to find a way to encode graph homomorphisms using components of PGs. Here two separate approaches should be applied for the cases when the graphs in our KR system are *simple* (at most one edge from the same source node to the same target node is allowed) or *non-simple* (multiple edges from the same source to the same target are allowed). The two cases result in different mathematical definitions of graph homomorphisms, i.e. in the first case a homomorphism is fully defined by a map of nodes, while in the second—by a pair of maps, one for nodes and one for edges (see formal definitions in Appendix A.1). In the first case we can simply use PG relationships (for example, of some reserved relationship type) to connect nodes from different graphs representing maps of nodes. The second case, on the other hand, is less trivial as the standard definition of the PG data model does not allow to have relationships between relationships. Therefore, it requires the revision of the initial encoding of our graphs with attributes: edges of graphs should be represented by, for example, nodes connected to their source and target with special types of relationships.

Finally, the last two challenges to be tackled are related to the update of a knowledge base. First of all, we need to be able to translate graph transformation approaches used in our framework to PG update queries. Secondly, some of the transformations are applied across different graphs. To make the transformations across different levels of hierarchy efficient we incorporate an extra structure into our knowledge base encoding—a hierarchy skeleton whose nodes represent graphs and whose edges represent homomorphisms. A more detailed description and discussion of this encoding in given in Section 2.3.

1.2.3 Schemas for property graphs

The PG data model has been originally conceived as a *schema-free* model. However, with constantly increasing popularity of graph databases, the requirement for developing schema-aware PGs has matured. Some graph database technologies, such as Neo4j, provide first rudimentary means for specifying schema-like constraints [10]. These constraints include the requirements on the uniqueness and the existence of properties.

Working on the encoding of our knowledge base using PGs, we have realized that some of the features of our framework provide sound and natural tools of designing schema-aware PG data model. A PG schema can be seen as a PG itself whose nodes define types of data nodes and whose relationships define types of allowed relationships between data nodes of different types, while properties of schema nodes and relationships define sets of allowed properties (and their values) for corresponding data elements. Seeing data and schema as a two graphs related with a homomorphism, allows us to reduce the task of schema validation to checking whether this homomorphism is mathematically valid (total, edge preserving, etc.)

Moreover, the approach to graph transformation that propagates across multiple graphs in the hierarchy (discussed in detail in Section 2.2) can be used to tackle the problem of data/schema co-evolution. It allows to perform updates according to the following two scenarios:

- *Prescriptive updates:* updates to the schema (such as removal of node and relationship types, splitting of types) that propagates automatically to the underlying data.

- *Descriptive updates:* updates to the data (such as addition of new graph elements, merge of nodes and relationships) that propagated automatically to the schema.

1.2.4 Related work

In this subsection we make a brief overview of the related work regarding two different problems. We discuss RDF and RDFS in the context of choice of encoding for our graphical knowledge base. After we provide some ideas on the existing work that tackles the problem of schema validation and co-evolution for graph databases.

Resource description framework and its schema (RDF and RDFS). We have previously discussed RDF and RDFS as standard representation tools for building ontologies. However, in the graph database community they provide a data model alternative to PGs. We have mentioned previously that RDF allows representing data with statements consisting of 'subject-predicate-object' triples. Statements and predicates are first-class citizens that can be used as a subject or an object in other triples [31]. Interpreted graphically this would mean that RDF enables us to add edges between edges, which could be extremely useful when encoding our graph hierarchy consisting of non-simple graphs (recall that to represent homomorphisms of non-simple graphs we need to represent maps of edges). However, this 'first-class citizenship' would apply for encoding of node/edge attributes as well. Notes and relations (predicates) in RDF do not have an internal structure, e.g. if we would like to represent the attribute 'name: Bob' of a node in RDF we would create a new node 'Bob' and relate it to this node with a predicate 'hasName'. This (on the first sight small) difference is one of the crucial arguments behind the choice of the PG data model over RDF, as it provides more efficient querying and storage capabilities.

Schemas for PGs. The problem of schema evolution is a classical and well-studied topic in data management [75]. The research on this topic concerns mostly relational databases [8]. There exists an extensive work on schema validation and co-evolution for XML and RDF [54, 68]. On the other hand, the problem of schemas for graph databases stays under-studied. While some recent attempts were made in PRISM [23], InVerDa [56], MoDEF [81], they mostly treat a PG schema as prescriptive (fixed, prescribing sets of allowed node/relationship types, etc.). Our approach, on the other hand, allows to achieve a spectrum of co-evolution capabilities that range from totally prescriptive (from schema to data) to totally descriptive (from data to schema), which enables great flexibility when modelling data on different stages of application's development.

1.3 Bio-curation for cellular signalling

In this section we address the challenges related to modelling biological systems and curating biological knowledge, which constitutes the main use-case for our KR system. We give a brief insight on what constitutes cellular signalling and how it can be modelled. We focus, in particular, on the rule-based modelling approach. We present the bio-curation framework KAMI—a novel approach that allows to decouple knowledge curation from model building. Finally, we present some related work that attempts to place KAMI in the context of existing curation and modelling tools.

1.3.1 Modelling cellular signalling

Cellular signalling emerges from thousands of individual interactions between proteins. Signalling allows a cell to adapt to a changing environment or to communicate with other cells in, for example, a multicellular organism. The final goal of signalling is a change in a cell's behaviour, e.g. an alternation of protein functions, their synthesis, etc. Traditionally cellular signalling is divided into intracellular (inside a single cell) and intercellular (between multiple cells) [2]. In this thesis we mainly focus on modelling intracellular signalling. It underlies many fundamental processes of living cells, from cell proliferation to their apoptosis (death). Its abnormalities are responsible for common and serious diseases such as cancer and diabetes.

Actors of signalling—receptor proteins, GTP-binding proteins, protein kinases—are large and complex molecules that can possess a number of 'on/off' states usually actualized by post-translational modifications (PTMs, e.g. phoshorylation, methylation, etc.). Every combination of such states alters a function of a protein, its catalytic activity, its ability to bind other proteins and so on. Moreover, the surface of some proteins can contain multiple binding sites that can be occupied by particular combinations of binding partners (ligands) at the same time. Every such combination of ligands may alter protein function as well.

Signalling actors form a functional network in which proteins play different roles: some relay the signal further in the network, some create conditions for interactions between other proteins (scaffold proteins), some can amplify the original signal, modulate the activity of other proteins [2], etc. This network is a highly redundant and enormously complex "spaghetti" of chemical reactions, disentanglement of which is a great challenge.

One of the common ways to disentangle signalling networks consists in finding *signalling pathways*—structures of interaction events leading to some event of interest (a change in a cell's behaviour, for example). Such signalling pathways do not exist physically, but are artifacts emerging from concurrent interaction events underlying the dynamics of signalling systems [14]. Knowledge of pathways does not only give an insight on biological mechanisms behind changes in a cell behaviour, it also allows to develop mechanisms for intervention into these changes (some of these pathways, for example, are involved in various diseases such as cancer).

This immense complexity of cellular signalling systems makes them extremely hard to model and analyse. Traditional approaches for modelling dynamical systems of reacting molecules (such as modelling with ODEs, stochastic chemical kinetics) require explicit listing of interacting agents in the system. In this contexts, every combination of protein states and bindings induces a species of an interaction agent [27]. The number of agent species of a typical signalling system becomes astronomical fairly quickly. Therefore, it becomes not only computationally hard to simulate and analyse such systems, even writing down their models is in itself a highly non-trivial problem.

Moreover, most of the classical and well-studied signalling pathways (e.g. MAPK/ERK pathway [79], Wnt signalling pathway [59]) were discovered and built by manually examining a set of what seemed to biologists relevant PPIs. Due to the complexity of signalling systems this way of modelling introduces inevitable *modelling bias* which may result in misleading or inaccurate models. Therefore, there is a need for tools allowing automated discovery and analysis of signalling pathways from 'models of nothing', i.e. executable mechanistic models produced from the aggregation of large knowledge corpora, collected without preconceived ideas on their relevance.

The nature of knowledge that underlies signalling models raises another challenge that makes our modelling task even more intricate. Knowledge on protein-protein interactions (PPIs) and

their PTMs is often fragmentary, incomplete and even incoherent; it can result from experimental data, an inference or a hypothesis. All these factors require from a feasible modelling approach to be incremental and to provide transparent and reasonable mechanisms for updating underlying knowledge on agents and their interactions (which is not the case when modelling with ODEs, for example).

The rule-based modelling approach (proposed by Kappa [26] and BioNetGen [40]) allows to overcome the described challenges. It solves the problem of explosion in the number of agent species and allows incremental model building. Moreover, this approach provides a set of tools for automated discovery of potential signalling pathways by constructing compressed causal past of simulations. This automated pathway discovery proposes a promising solution for the problem of modelling bias inherent to manual pathway building. On the other hand, the KAMI framework, developed as a part of this thesis, proposes a system for representation and curation of signalling knowledge that provides means for automated and reusable aggregation of individual PPI knowledge, and allows for automated generation of executable rule-based models. The following subsections present the rule-based modelling approach and the KAMI framework in more details.

1.3.2 Rule-based modelling

Rule-based modelling approaches allow to build mechanistic models of biochemical interactions in terms of agents equipped with states and sites (defining agent interfaces) and rules of their interactions. Rules specify how local patterns of states and sites change as the result of interactions between agents [26]. These local patterns represent necessary conditions for interactions to appear, this is often referred to as the 'do not care, do not write' approach.

A system's state is represented with a large graph structured in a particular way. Its evolution (i.e. the result of biochemical events) is then mimicked using graph transformation techniques. This frees the modeller from the necessity to enumerate all the molecular species arising in the course of system's evolution (allowing discovery of species as an artefact of system's dynamics) and provides means for building scalable and easily updatable models. Moreover, it allows to employ a range of techniques for causal analysis that can be used to *discover signalling pathways* [25].

In this subsection (for historical reasons) we will focus on the Kappa language and give a brief overview of the set of tools constituting the Kappa platform—a set of software components that allow to perform simulation and analysis of the dynamical systems [14].

Representation of knowledge in Kappa. The basic modelling units of Kappa are called *agents* (in our context they represent proteins) possessing *sites* (used to represent interaction sites and post-translational modifications). Using sites agents connect to each other into *site graphs*, which implicitly represent molecular species. As it was previously mentioned, the system's state is represented with a *mixture* graph (comprising of multiple site graphs). The *rules* consist of two site graphs L and R that can be seen as patterns representing necessary conditions for the interaction to happen and its result respectively. A rule is applied to a mixture by embedding L (finding a part of a site graph matching the necessary conditions described by L) and replacing it with R. A Kappa *model* is then given by a set of rules together with an initial mixture.

Kappa simulator (KaSim). Simulation of the system dynamics is performed by the Kappa simulator KaSim. The simulation techniques used are stochastic and are based on continuous time Markov chains (more details can be found in [28, 44]). Simulations are optimized for particular data structures used by the simulator with clever techniques from graph rewriting [13]. Moreover, every simulation is interactive, i.e. it can be paused and system perturbations can be introduced on the fly. During simulation KaSim is able to produce the dynamic influence network (DIN) whose nodes represent rules and whose edges represent the influence of rules on each other (such as activation and inhibition). Analysis of this network can produce useful insights on the system dynamics, e. g. rule-fluxes over time, clusters of highly interrelated rules (that may suggest potential emergence of pathways). The results of simulations can be uploaded and DINs can be visualized on DIN-Viz server [14]. KaSim allows to convert rule-based models into different representations amoung which are ODEs (when feasible) and SBML (a standard model exchange language [58] widely used in the systems biology community).

Kappa static analyzer (KaSA). Some kinds of analysis of the model described with a rule set can be performed without running simulations. Such analysis is therefore called *static* and can be performed using the Kappa static analyzer KaSA [12, 41]. Among the questions that can be answered statically are the following: reachability of a molecular species, identification of 'dead' rules (whose necessary conditions cannot be satisfied in the current rule-set), detection of invariants, static rule influence (i.e. which rules activate/inhibit each other 'in principle'), detection of unbounded polymers. Analysis provided in KaSA does not only give some useful insights on the model, but also helps to debug it by detecting potential anomalies.

Kappa story extractor (KaSTOR). KaSim produces a sequence of events (basically applications of rules) that have occured during a simulation. This sequence is called a *trace*, and can be arbitrary long. The main goal of the Kappa story extractor is, given a trace, to find a relevant sequence of events that lead to some event of interest (EOI) specified by the user (events that explain *how* the EOI happened) [25, 14]. These sequences are called *stories* and the idea behind them corresponds to what biologists call *pathways*. First of all, KaSTOR reconstructs a directed acyclic graph representing the precedence relations between events of the trace (note that two events happening one after another does not imply the precedence relation, i.e. some events are concurrent). However, such graphs can be quite large and they do not directly represent stories. To obtain stories this graph should be reduced to contain a minimal set of events necessary to produce the EOI. The process producing stories from precedence graphs in KaSTOR is called *causal compression* and is based on techniques employed from the graph transformation community [25].

1.3.3 Bio-curation for cellular signalling with KAMI

As we have previously discussed, rule-based approaches allow to build incremental models and overcome the problem of combinatorial explosion in the number of molecular species. However, building and maintaining large signalling models using directly these approaches stays cumbersome and cognitively heavy. Moreover, they are not particularly suitable for *bio-curation*, i.e. for collation and maintenance of ever-growing corpora of signalling knowledge.

The first reason for that is the *'update problem'* which originates from the fact that different rules expressing PPIs are not necessarily independent and may represent instances of the same *interaction mechanisms*. A large proportion of PPIs involved in cellular signalling are instances

of generic interaction mechanisms of conserved protein domains (for example protein kinase, phosphatase, SH2, PTB domains, etc.). These interaction mechanisms are generally well-studied and appear in highly specific conditions (for example, presence of particular sequence motifs). An update of knowledge on a particular interaction mechanism may require identification and update of all the rules that express this mechanism, which is extremely difficult and error-prone to perform manually.

Let us consider a simple example of the 'update problem' that can arise in the course of modelling. Figure 1.2 presents a set of Kappa rules expressing phosphorylation of different substrates by the protein kinase enzyme SRC. The three interactions correspond to the same interaction mechanism, i.e. the phosphorylation mechanism of protein kinase region of SRC.

```
SRC, SHC1(p) -> SRC, SHC1(p{1}) // SRC phosphorylates SHC1
SRC, STAT3(p) -> SRC, STAT3(p{1}) // SRC phosphorylates STAT3
SRC, JAK2(p) -> SRC, JAK2(p{1}) // SRC phosphorylates JAK2
```

Figure 1.2: Example of Kappa rules expressing the same interaction mechanism of SRC (KaSim 4.0 compatible syntax). Rules are given in a monospaced font, comments—in italic.

Now, assume that the modeller would like to revise the knowledge expressed with the rules in Figure 1.2. For example, she found out that SRC is required to be activated to acquire its enzymatic activity (the ability to phosphorylate). To perform this revision all the rules expressing the instances of the phosphorylation interaction of SRC should be identified and updated (Figure 1.3 presents an example of updated rules). While performing such an update in this simple example may seem trivial, in real-world signalling models consisting of thousands of rules (that may involve other interaction mechanisms of SRC as well) it is often a bottleneck.

```
SRC(activity{1}), SHC1(p) -> SRC(activity{1}), SHC1(p{1})
                                                  // active SRC phosphorylates SHC1
SRC(activity{1}), STAT3(p) -> SRC(activity{1}), STAT3(p{1})
                                                 // active SRC phosphorylates STAT3
SRC(activity{1}), JAK2(p) -> SRC(activity{1}), JAK2(p{1})
                                                  // active SRC phosphorylates JAK2
```

Figure 1.3: Example of updated Kappa rules from the example in Figure 1.2.

The second issue that arises when using rule-based modelling languages for collation of mechanistics knowledge about signalling-related PPIs is called the *instantiation* problem, i.e. the re-use of knowledge in different contexts (such as different cell types, mutants). Depending on the context, and more precisely on the anatomy of participating proteins, the same set of PPIs can give a rise to different models (for example, some PPIs that depend on particular functional domains of proteins do not take place in the contexts where these proteins lose the necessary domains).

Consider another small example of Kappa rules presented in Figure 1.4. The rules represent bindings of GRB2 to SHC1 and EGFR through its SH2 domain and to GAB2 through one of its SH3 domains. For the first two binding interactions GRB2 is required to have the SH2 domain (a biochemically active 'sticky' surface of the SH2 domain allows GRB2 to bind to its SHC1 and EGFR). The third interaction, on the other hand, does not depend on the presence of SH2 (it requires another domain, one of GRB2's SH3 domains to be present). Note that

the illustrated rules do not make these requirements explicit (here the names `SH2` and `SH3` are purely incidental). GRB2 has a natural isoform GRB3-3 (a splice variant) with the SH2 domain removed. Imagine now that the modeller would like to include in the model the GRB3-3 variant of GRB2 and to reuse already present PPI knowledge to infer a set of interactions for GRB3-3. Clearly, GRB3-3 would inherit the binding interaction with GAB2, but not the ones with SHC1 and EGFR.

```
GRB2(SH2[.]), SHC1(p1, pY_site[.])  -> GRB2(SH2[1]) SHC1(p1, pY_site[1])
                                          // GRB2 binds SHC1 through SH2 domain
GRB2(SH2[.]), EGFR(p1, pY_site[.])  -> GRB2(SH2[1]) EGFR(p1, pY_site[1])
                                          // GRB2 binds EGFR through SH2 domain
GRB2(SH3[.]), GAB2(grb2_site[.])  -> GRB2(SH3[1]), GAB2(grb2_site[1])
                                          // GRB2 binds GAB2 through SH3 domain
```

Figure 1.4: Example of Kappa rules expressing binding of GRB2 to different ligands through its SH2 domain and SH3 domain.

The two presented issues illustrate that due to the complexity of signalling models and the nature of knowledge there exists a need to separate the process of model building from collation, analysis and update of underlying biological knowledge—*bio-curation*.

KAMI tackles the 'update problem' by conceptualizing the notion of interaction mechanism and providing their explicit representation (which allows to identify the instances of interaction mechanisms immediately). The instantiation problem is solved by de-contextualizing protein-protein interactions. KAMI enables the collation of knowledge about *potential* individual PPIs and their necessary conditions into a knowledge *corpus*. It allows then to instantiate this knowledge into *models* consisting of *concrete* PPIs. Agents of potential interactions are called *protoforms* and represent neighbourhoods in the sequence spaces of different genes. By associating regions, residues and states to a specific protoform the modeller represents a feasible neighbourhood of its variants. KAMI allows the re-use of the knowledge on potential interactions for the automatic generation of models in different systems, by specifying which agents are present in these systems, that determines which interaction mechanisms are realizable.

The bio-curation framework proposed by KAMI enables the collation of knowledge about individual PPIs and the semi-automatic aggregation of this knowledge into a coherent corpus identifying interaction agents and mechanisms according to some body of grounding knowledge. It then allows to instantiate this knowledge into various signalling models and to automatically generate executable models such as Kappa scripts. KAMI's pipeline is summarized in Figure 1.5. The conceptual framework of KAMI is implemented in the Python library KAMI[3] and a graphical environment KAMIStudio[4]. More detailed discussion of KAMI and the related software tools can be found in Chapter 4.

1.3.4 Sources of signalling knowledge

Before we can conclude this introductory section we need to discuss another important question. This question concerns the sources of knowledge for building signalling models. Even the most elaborated knowledge curation tool is virtually useless without relevant and reliable knowledge.

[3] `https://github.com/Kappa-Dev/KAMI`
[4] `https://github.com/Kappa-Dev/KAMIStudio`

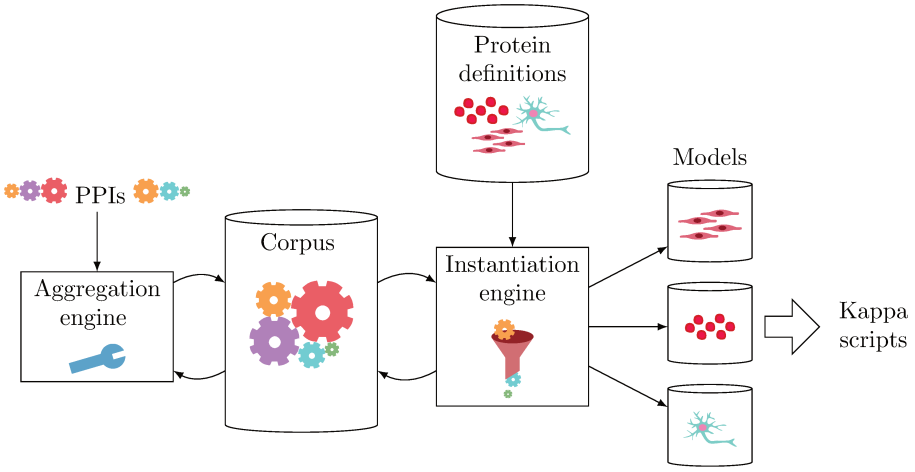


Figure 1.5: KAMI bio-curation pipeline. Individual PPIs are fed to KAMI's aggregation engine, which assembles them into a coherent knowledge corpus. Using definitions of protein variants in different contexts, this corpus can be further reused and instantiated into different signalling models, which subsequently can be used to generate executable Kappa scripts.

In this subsection we discuss the kinds of knowledge necessary to model cellular signalling and the typical sources of this knowledge.

To be able to build accurate signalling models we need to aggregate detailed mechanistic knowledge on necessary conditions of PPIs (PTMs, presence or absence of other bindings, etc.). This knowledge is scarce and is often biased by 'hot topics' of research, such as cancer research. A lot of potential PPIs are not targeted for discovery when not preconceived to be involved in biological processes of interest. Mass-spectrometry combined with recent high-throughput techniques [1, 57, 77, 84] makes the discovery of PTMs and PPIs much cheaper and faster, therefore, allowing to target much larger range of proteins and to reduce the discovery bias. Although high-throughput protein interaction data often lacks accuracy and is often limited to motif-domain and domain-domain interaction [77], it gives a promising direction for unveiling a genome-scale PPI network.

As we have previously mentioned, good rule-based models require detailed mechanistic knowledge about PPIs. The success of automated discovery of signalling pathways that such models promise relies upon the quality and reliability of knowledge underying them. The level of mechanistic detail, however, varies greatly from one knowledge source to another. For, example some sources often contain phenomenological descriptions of interactions and functional states rather than mechanistic details and structural changes respectively. Literature and interaction databases often contain statements of a sort *"active A activates B"*. To build an accurate rule-based model containing this interaction, we needs to detail and de-contextualize this phenomenological knowledge. We need to understand what is the context of this interaction, i.e. what do *"active A"* and *"activation of B"* mean. Another similar problem arises when we encounter a pair of statements like *"A binds B"* and *"A binds C"*. In this case, to be able to understand whether the two bindings are conflicting or not we need to identify the exact mechanism of both bindings. If the two interactions are conflicting, A cannot bind B and C at the same time, so our system can contain only A-B and A-C complexes. In the second case our system can potentially contain A-B, A-C and B-A-C complexes (here, a thorough modelling would require to investigate, for example, whether the complex C-A-B is possible and is different

from B-A-C, whether the binding rate of A to B changes when A is already bound to C, etc.).

In the rest of this subsection we discuss the common formats in which different sources represent signalling knowledge. Such formats range from standard machine-readable representation formats to natural language and manual input.

BioPAX. BioPAX (Biological Pathway Exchange) is a standard language for the representation of biological pathway data such as metabolic and signalling pathways, molecular and genetic interactions and gene regulation networks [32]. Its main goal is to facilitate representation and exchange of pathway data from different sources. BioPAX uses syntax based on OWL and provides a set of tools for validation and querying BioPAX documents.

Our project (at least in its current state) focuses exclusively on modelling of cellular signalling, therefore, it is clear that BioPAX targets the representation of a much broader spectrum of knowledge. It includes mechanistic details of protein-DNA/RNA, protein-protein interactions as well as interactions between small molecules and proteins. Moreover, BioPAX allows to represent knowledge that in our context (the context of KAMI) we consider as phenomenological. It can directly represent activation and inhibition of enzyme molecules (with the possibility to specify additional knowledge on whether their are competitive, allosteric, irreversable etc.). It also allows for direct accommodation of knowledge on metabolic and signalling pathways, which according to the rule-based modelling framework is exactly the knowledge to be *discovered*.

Another interesting point of dissimilarity to elaborate concerns the status of protein complexes in the representation of rule-based models and BioPAX. A large number of PPI involve protein complexes as their actors, i.e. large molecular aggregates consisting of multiple proteins (i.e. homo- or heteropolymers). BioPAX provides means for representation of interactions whose actors are such protein complexes. Some modification and binding interactions performed by a specific protein are possible only if this protein is a part of a larger complex (for example, allosteric control), while some of them are actually performed by multiple molecules bound together in a complex (for example, functional sites formed by surfaces of two adjacent proteins). These subtleties can have a significant influence on the dynamics of the modelled signalling system, while the knowledge that would allow to discriminate between these two cases cannot be represented in BioPAX. Moreover, the event of complex assembly is usually a multi-step procedure involving a non-trivial temporal partial order. In BioPAX, complex assemblies (as well as all other biochemical reactions) are represented with a left-hand side providing reactants and a right-hand side providing reaction products. This kind of representation, viewed through the lens of rule-based modelling, misses some mechanistic details (i.e. the exact binary interactions involved in the complex assembly).

Concerning the status of BioPAX as a candidate knowledge input format to KAMI, the main conceptual differences between the knowledge representation in KAMI and BioPAX can be summarized and extended in the following points:

- BioPAX can represent broader range of knowledge, currently not supported by KAMI, e.g. generic, metabolic interactions.
- BioPAX allows to accommodate knowledge on different abstraction levels, including phenomenological interactions such as activation or inhibition as well as direct representation of pathways.
- The representation level of BioPAX corresponds to the one of rule-based modelling and does not solve the rule update problem, i.e. there is no notion of interaction mechanisms.

However, at least a substantial subset of BioPAX can be used as an input format for knowledge fed to KAMI's aggregation engine. This would permit to automatically import from the resources that use BioPAX for knowledge export (such as Reactome[5], Biomodels[6], Signaling Gateway Molecule Pages[7]). For this purpose, we have implemented a prototype BioPAX importer for KAMI that is able to import a signalling-related subset of BioPAX (discussed further in Subsection 4.5.1).

PSI-MI. Similarly to BioPAX, the PSI-MI (Proteomics Standards Initiative Molecular Interaction) XML format [70, 60, 78] aims to provide a standard for exchange of data on protein interactions. The basic atoms of PSI-MI are called entries and can group one or more interactions together with annotations indicating the experimental methods used to determine interactions and their agents. It relies on controlled vocabularies (rather than free text attributes) that can be defined externally that allow to specify interaction/agent detection methods, interaction types, agent features, etc. Among the databases allowing export in PSI-MI formats are IntAct[8] [55], DIP[9], MINT[10]. The implemented bio-cuation tool KAMI (discussed in Section 4.5) provides an importer for PSI-MI 3.0 format.

Natural language. A wast amount of relevant knowledge is not present in machine-readable formats, but expressed with natural language in scientific articles. Therefore, a question of automated reading and understanding of natural language is being actively investigated (e.g. automated readers TRIPS/DRUM, REACH, R3 [45, 15]). The language of biomedical articles is technical, written by professional scientists and intended to be understood by such. It greatly facilitates the task of natural language processing (NLP), however, does not completely solve it. No formalized writing conventions whatsoever exist and the language of biomedical literature stays 'natural'. Challenges in understanding this natural language include [15, 45]:

- Entity recognition: for example, in text 'RAS' can refer to gene, protein or complex.
- Mismatch of abstraction levels of different texts (e.g. *"A phosphorylates B"* vs *"A binds B, A bound to B phosphorylates B, A unbinds B"*). How to assemble such knowledge into a single coherent model, how to relate different abstraction levels?
- Logically related knowledge is spread across the sentences and paragraphs. Obtaining a complete description of a mechanism requires collecting and assembling pieces of knowledge from different parts of the text, which is a non-trivial task.
- Some background knowledge is assumed to be known by the reader and therefore omitted in the text.

Manual input. Finally, it worth mentioning that there is no more valuable input than an input from a human expert. An expert is able to analyze, de-contextualize, disambiguate, investigate and synthesize mechanistic details from different sources. Therefore, a curation tool

[5] `https://reactome.org/`
[6] `http://www.ebi.ac.uk/biomodels/`
[7] `http://www.signaling-gateway.org/`
[8] `https://www.ebi.ac.uk/intact/`
[9] `https://dip.doe-mbi.ucla.edu/dip/Main.cgi`
[10] `https://mint.bio.uniroma2.it/`

should provide an intuitive and flexible format for manual input of signalling knowledge at an arbitrary level of mechanistic detail. Moreover, this tool should provide means for validation and verification of this manual knowledge (however beneficial this manual input is, it stays prone to human error).

A graphical bio-curation environment KAMIStudio, developed as a part of this thesis, provides flexible intuitive forms for input of individual PPIs. It allows to represent various kinds of modification and binding interactions, specify interaction proxies (such as functional domains and sites of proteins), structural requirements (such as the presence of particular domains, sites, key residues) and PTMs. Further discussion of this topic can be found in Section 4.6.

1.3.5 Related work

Originally KAMI was developed as a part of DARPA's Big Mechanism Project [17]. The main goal of this project was to develop tools for automatic machine reading of the biochemical literature and assembly of the extracted knowledge into large causal models with special focus on cancer signaling pathways. Such causal models could be then used for reasoning and explanation of fundamental signalling phenomena underlying the living cell, development of new drugs and automatic design of experimental protocols.

In this section we would like to discuss a set of tools related to KAMI (some of them being a part of the Big Mechanism Project as well). Functionalities of these tools facilitate modelling of cellular signalling systems and are either complementary or have some intersection with KAMI. We conclude by connecting KAMI and the related tools in the same picture (see Figure 1.6) that hopefully can give some broader context to our project.

MetaKappa. Presented in [27, 47], MetaKappa provides means for specifying a space of concrete models derived from a set of rules and an associated hierarchy of agents. Such a hierarchy of agents can be used to represent splice variants, mutated forms of the same genes, families of proteins, etc. Arbitrary entities of this hierarchy can be used as agents in the rule set, e.g. a protein family can be used to specify a generic rule for all proteins in this family. Historically, MetaKappa can be considered as a conceptual predecessor of KAMI; it makes a first step towards the de-contextualization of knowledge encoded in Kappa rules. However, there exist some crucial conceptual differences between MetaKappa and KAMI:

- MetaKappa cannot be used to specify gain-of-function mutants, i.e. variants produced from mutations that enable particular PPIs.
- Derived from Kappa, MetaKappa cannot represent fine-grained agent components subjected to splicing and mutations, i.e. protein domains and residues.
- Mechanisms are implicit, therefore the problem of identification and update of non-independent rules persists.

Essentially, MetaKappa facilitates writing already conceptualized Kappa rules in a more "laconic" way, while KAMI, on the other hand allows discovery of these rules by contextualizing mechanisms [49].

INDRA. INDRA (Integrated Network and Dynamical Reasoning Assembler) represents another closely related tool. Similarly to KAMI, the goal of INDRA is to decouple the curation of

knowledge from model implementation [45]. INDRA is integrated with various NLP tools (e.g. REACH [83], DRUM [3], Sparser [15]) that allow to extract relevant knowledge directly from texts of biochemical literature. The model-building framework proposed by INDRA consists of three main steps: (1) text-to-model conversion, (2) generation of INDRA statements and (3) assembly of statements into a model.

First of all, an input text is processed to some machine-interpretable intermediate representation which includes grounding of identities of proteins and genes in reference databases. Then, this representation is converted to INDRA statements. Statements constitute the intermediate knowledge representation used in INDRA. They can express different kinds of interactions such as protein modifications as well as some phenomenological interactions such as complex assembly, activity and amount regulation. This representation is further used to assembly models (e.g. networks of differential equations, rule-based models or PPIs networks). As will be discussed in more detail in Subsection 4.2.1, KAMI provides a set of classes similar to INDRA statements called KAMI interactions that are used for the representation of PPIs. However, they are used by KAMI as an intermediary representation that facilitates user input (through KAMI programmatic API or interactive forms of KAMIStudio). They are further used to generate "their true representation" used in KAMI, i.e. a collection of nugget graphs that are aggregated into a corpus. The main differences in knowledge representation of KAMI and INDRA can be summarized as follows:

- INDRA does not allow to represent protein regions and sites and use them as actors of interactions.
- INDRA allows to represent phenomenological knowledge, e.g. activation, complex assembly, regulation of amounts, assumed by KAMI as knowledge to be *discovered*.
- Agents of interactions expressed with INDRA statements are specific gene products and, therefore, mutants are treated as distinct agents.
- INDRA does not conceptualize interaction mechanisms, therefore, does not solve the problem of non-independent rule update either.

Overall, KAMI and INDRA are aiming to solve (at least superficially) similar goals with significantly different approaches. Similarly to KAMI, INDRA aims to separate curation of mechanistic knowledge from model building [45]. However, KAMI provides a semantically rigorous framework for curation of de-contextualized knowledge about generic mechanisms of PPIs at the last step of which resides the generation of concrete models. On the other hand, INDRA allows extracting (contextualized) knowledge about concrete PPIs into a pool of independent statements and employing various techniques (both systematic and ad hoc) to automatically assemble these statements into models. The greatest asset of INDRA is its ability to extract knowledge directly from biomedical texts. Therefore, combining INDRA and KAMI is certainly of interest, and KAMI provides a small importer of INDRA statements into KAMI interactions. Note, however, that due to some discontinuity between the knowledge representation of the two tools some knowledge that can be extracted from the literature is lost when represented with INDRA statements (e.g. functional sites and conserved protein domains). Moreover, it would be very interesting to combine KAMI directly with NLP tools not only to prevent the loss of relevant knowledge, but also to implement context-dependent "reading with a model" using KAMI corpora as such models [15].

Other related tools. A number of languages for modelling with pathway diagrams have been developed together with tools for their visualization and analysis (e.g. SBG [24] and eEPN [71]). However, these tools rely on explicit representation of molecular species (that may lead to combinatorial explosion), do not conceptualize interaction mechanisms (therefore, do not solve the 'update problem'). Most importantly, they are designed to be used by biologists for manual building of pathways, while one of our main claims (and motivations behind designing KAMI) is that signalling pathways are not objects of modelling, but rather phenomena emerging from the dynamics of modelled systems and that ought to be discovered.

The Python framework PySB [63] allows to build models of biochemical systems using simple and intuitive domain-specific language and generate BioNetGen, Kappa and systems of ODEs. LBS-κ represents an extension of Kappa-language [72] allowing the modeller to define parameterized modules representing sets of generic PPIs (for example, phosphorylations) and to instantiate these modules with particular agents and their sites. It also provides a solution for representing compartmentalization of reactions (i.e. reactions that happen in particular cellular locations and transport reactions). Although these tools provide some additional abstraction level over rule-based models that aicilitates and simplifies writing of models, they do not solve 'the update problem' and, importantly, do not separate knowledge curation process from model building.

All this makes KAMI a unique bio-curation tool that occupies its niche in the ensemble of existing knowledge curation and modelling tools and is able to connect them into a coherent pipeline (see Figure 1.6), at the beginning of which is automated reading of biomedical literature and at the end of which are insights on the dynamics of signalling systems of a living cell.

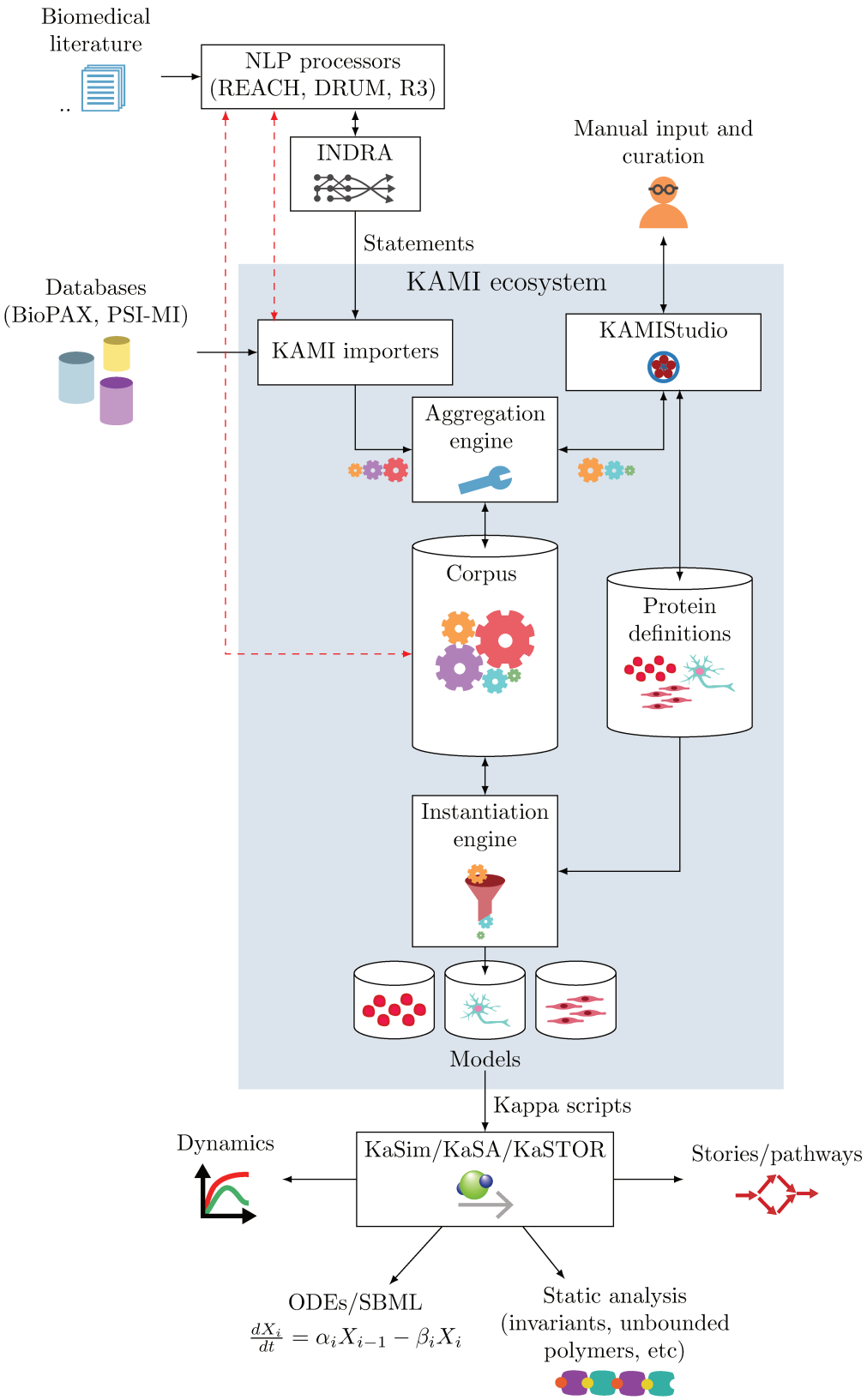


Figure 1.6: The KAMI ecosystem and its context

Chapter 2

Rewriting in graph hierarchies

Graphical structures and their transformations are widely used in various domains of computer science, such as software engineering, databases and distributed systems. In this thesis we use graphs predominantly for KR purposes and graph transformations for knowledge update and audit. We start this chapter by Section 2.1, where we present the main building blocks of our KR system, graphs and graph transformations.

We build a hierarchic structure consisting of individual graphs related by homomorphisms—a *hierachy of graphs*. This structure can be used to represent fragmented knowledge of a graphical nature and relate the fragments using homomorphisms. Using the designed structure, various relations between knowledge fragments can be expressed, such as relation between knowledge on different abstraction levels, typing, identification, etc. We further study the ways to transform individual graphs inside such a structure that allow consistent co-evolution of its different layers. In general, the mathematical theory provided in Section 2.2 can be used to build hierarchical structures of any objects in categories satisfying certain structural requirements.

Recall that one of the main motivations behind the design and the implementation of our KR system is to provide a flexible and mathematically rigorous framework for knowledge curation, i.e. collation, organization and maintenance of knowledge. In this context, we design a system for an audit of updates in graphs and graph hierarchies, called an *audit trail*. Audit trails allow us to store and navigate the history of updates, maintain multiple versions of our objects (therefore, multiple versions of knowledge) and rollback to specific object versions in the update history. They are presented and discussed in Subsections 2.1.4 and 2.2.10.

As a part of this thesis, a software tool implementing the presented mathematical theory—the Python library `ReGraph`—has been designed and implemented. Section 2.3 gives some details on its architecture, specifications and example use-cases.

The contributions on hierarchies of graphs and their rewriting were communicated as a conference paper in [52] and its journal version in [53]. The work on rule hierarchies, reversibility and composition of rewriting in hierarchies was submitted to the 13th International Conference on Graph Transformation (ICGT 2020) and its pre-print version can be found in [].

2.1 Graph rewriting

Graphs constitute natural mathematical objects for representation of sets of entities related between each other in some particular ways. Sets of entities are modelled with sets of *nodes* (often called vertices in the graph theory), while their relations are modelled with sets of *edges*.

Edges can be undirected or directed representing symmetric or asymmetric relations between entities. Nodes and edges of graphs can be equipped with some internal structure providing, for example, a description of the entities and relations they represent. In this thesis, we will refer to such internal data as *attributes*. Additionally, we will distinguish between simple and non-simple graphs, i.e. graphs where at most one edge is allowed between any given pair of nodes and, on the other hand, where multiple edges are allowed. This distinction is important when defining graph homomorphisms. For simple graphs we will define a homomorphism of one graph to another as a function that associates nodes of the first graph with the nodes of the second is such a way that edges stay preserved. While in the case of non-simple graphs our homomorphisms will consist of two maps, one for nodes and one for edges, satisfying some properties. Formal definitions of graph objects, attributes, graph homomorphisms and categories they form can be found in Appendix A.1. The objects of our interest are both simple and non-simple directed graphs with dictionaries of attributes attached to their nodes and edges. Their categorical formulation is necessary to apply graph transformation techniques of interest, namely algebraic approaches to graph rewriting.

Algebraic approaches to graph rewriting define a transformation of a graph G with a graph rewriting rule $r : L \rightsquigarrow R$, where L and R are often referred to as the left- and the right-hand side of the rule respectively. Rule application is performed given a match $l : L \to G$ defining an occurrence of rule's left-hand side in G. This application performs a transformation $G \rightsquigarrow G'$ by finding a match of the left-hand side and 'replacing' it with the right-hand side of the rule. In the rest of this section we study the rewriting approach of our interest—the sesqui-pushout (SqPO) approach—in more detail and show how it can be used as a powerful tool for the update and audit of knowledge represented with graphs.

2.1.1 Sesqui-pushout rewriting

SqPO rewriting is an approach for abstract deterministic rewriting in any category with pushouts and pullback complements over monomorphisms [20] (definition of monomorphisms, pushouts, pullback complemens can be found in Appendix A). In the context of graphs with attributes, SqPO allows to perform the operations of addition, deletion, cloning and merging of graph elements, such as nodes, edges and attributes. Rewriting of a graph G is defined by a rule $r : L \leftarrow r^- - P - r^+ \rightarrow R$, its matching is given by a monomorphism $m : L \rightarrowtail G$ (injective function where at most one node/edge of L maps to a node/edge of G). The graphs L, P, R are referred to as the left-hand side, the interface and the right-hand side respectively.

Application of the rule r is performed in two phases: (1) a graph G^- is constructed as the final pullback complement from $P - r^- \rightarrow L \rightarrowtail m \rightarrow G$ (see the formal definition in Appendix A.5), it represents the intermediary rewriting result obtained after removing and cloning graph elements; (2) the final result of rewriting G^+ is constructed as the pushout from $G^- \leftarrow m^- \rightarrowtail P - r^+ \rightarrow R$, it corresponds to addition and merge of the graph elements specified by the rule. We often refer to the first rewriting phase as the *restrictive* phase and to the second one as the *expansive* phase. Diagram 2.1 illustrates the categorical constructions of the rule application corresponding to the two phases, while Figure 2.1 gives a small example of such rewriting.

$$\begin{array}{ccccc} L & \xleftarrow{r^-} & P & \xrightarrow{r^+} & R \\ \downarrow{\scriptstyle m} & (1) & \downarrow{\scriptstyle m^-} & (2) & \downarrow{\scriptstyle m^+} \\ G & \xleftarrow[g^-]{} & G^- & \xrightarrow[g^+]{} & G^+ \end{array} \tag{2.1}$$

The categorical constructions for concrete graph categories of interest are formulated in Appendix A (in particular, A.4 for pushouts and A.5 for final pullback complements). While

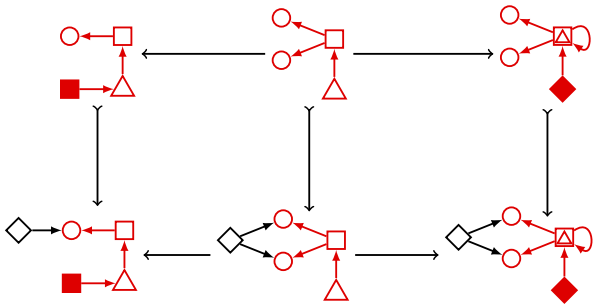


Figure 2.1: Example of SqPO rewriting. The span on the top represents a rewriting rule. In the first phase this rule removes the node represented with the filled square and clones the circle, in the second phase it merges the triangle with the square and adds the filled diamond together with an edge to the merged node. The bottom span illustrates how this rule is applied inside the input graph (the matched patterns corresponding to different phases of rewriting are highlighted with red).

the notion of a pushout, used to apply SqPO rules, can be considered as a classical and widely-used categorical notion, the final pullback complement is slightly less common. We collect a set of useful lemmas and properties of final pullback complements in Appendix A.5. Note that final pullback complements are unique up to unique isomorphism, which makes SqPO rewriting deterministic [20].

To give the reader some intuition on how SqPO rewriting works for graphs with attributes, informally, application of a rule $r : L \leftarrow r^- - P - r^+ \rightarrow R$ can be reformulated as the following sequence of steps (in the canonical order):

1. *Node deletion:* for every node $v \in V_L$ of the left-hand side of the rule that is not in the image of r^- delete its match $m(v)$ together with all its incident edges. The removal of edges incident to the removed node is considered to be a *side-effect* of SqPO rewriting.

2. *Node cloning:* for every node $v \in V_L$ of the left-hand side of the rule that has more than one pre-image in P, i.e. such that $|(r^-)^{-1}(v)| \geq 1$, clone it $|(r^-)^{-1}(v)| - 1$ times.

3. *Edge deletion:* for every edge $e \in E_L$ of the left-hand side of the rule that is not in the image of r^- delete its match $m(e)$. Recall that in the category of simple graphs (formally defined in Appendix A.1) our edge e corresponds to some pair of nodes $(u, v) \in V_L \times V_L$ and its matching edge corresponds to the edge $(m(u), m(v)) \in E_G$.

4. *Attribute deletion:* for every graph element $a \in V_P \cup E_P$ of the interface, let $attrs(a)$ be the attribute dictionary of this element and $attrs(r^-(a))$ be the attribute dictionary of the element it maps to in the left-hand side of the rule. Remove the dictionary defined by $attrs(r^-(a)) \setminus attrs(a)$ from the attributes of the element matched by a in the target graph (the operation of dictionary difference is defined in Appendix A.1).

5. *Node merging:* for every node $v \in V_R$ of the right-hand side that has more than one pre-image in P, i.e. such that $|(r^+)^{-1}(v)| \geq 1$, merge all nodes from $(r^+)^{-1}(v)$ into a single node.

6. *Node addition:* for every node $v \in V_R$ of the right-hand side that is not in the image of r^+ add a new node. If v contains some attributes, add these attributes to the newly added node.

7. *Edge addition:* for every edge $e \in E_R$ of the right-hand side of the rule that is not in the image of r^+, add an edge between the corresponding nodes in the input graph.

8. *Attribute addition:* for every graph element $a \in V_P \cup E_P$ of the interface, let $attrs(a)$ be the attribute dictionary of this element and $attrs(r^+(a))$ be the attribute dictionary of the element it maps to in the right-hand side of the rule. Add the dictionary defined by $attrs(r^+(a)) \setminus attrs(a)$ to the attributes of the element matched by a in the target graph (addition in this context corresponds to the operation of dictionary union defined in Appendix A.1).

In this thesis we will often consider rules as simply arrows of a form $r : L \leftarrow L^-$ or $r : L \rightarrow L^+$. We then refer to the first kind of rules as *restrictive rules*, call a mono $m : L \rightarrowtail G$ a *restrictive instance* and apply them by constructing the pullback complement $L^- \rightarrowtail^{m^-} G^- \rightarrow^{g^-} G^-$ (as in Diagram 2.2). The second kind of rules are referred to as *expansive rules*, a mono $m : L \rightarrowtail G$ is called an *expansive instance* and their application is performed by constructing the pushout $G \rightarrow^{g^+} G^+ \leftarrow^{m^+} L^+$ (as in Diagram 2.3). An arbitrary SqPO rule $r : L \leftarrow^{r^-} P \rightarrow^{r^-} R$ can be seen as a composition of a restrictive rule given by $r^- : L \leftarrow P$ and an expansive rule given by $r^+ : P \rightarrow R$.

$$\begin{array}{ccc} L & \xleftarrow{r} & L^- \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle m^-} \\ G & \xleftarrow[g^-]{} & G^- \end{array} \qquad (2.2) \qquad\qquad \begin{array}{ccc} L & \xrightarrow{r} & L^+ \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle m^+} \\ G & \xrightarrow[g^+]{} & G^+ \end{array} \qquad (2.3)$$

2.1.2 Reversible sesqui-pushout rewriting

An SqPO rewrite of a graph may introduce some *side-effects*, i.e. graph transformations not explicitly specified by the underlying rewriting rule. The nature of these side-effects depends on the category in which we are working. For example, in both simple and non-simple graphs some edges not matched by the left-hand side of the rule can be removed as a side-effect of a node removal. For the category of simple graphs, for example, merging of nodes may cause some edges incident to the merged nodes to be merged (simply because multiple edges between the same nodes are not allowed by definition). These side-effects make graph transformations irreversible. Having applied a rewriting rule and transformed the original object, we can no longer restore this original object by simply looking at the applied rule. Here we formulate the notion of the *reversibility* of SqPO rewriting, i.e. the absence of side-effects. The reversibility of transformations is a desirable property in a lot of practical applications. For instance, if we would like to record a history of transformations in an audit trail and be able to 'undo' these transformations, rewriting side-effects become an important issue. Additionally, in the next subsection we will see how the reversibility allows us to compose SqPO rules.

First introduced in [29], reversible SqPO rewriting formalizes the notion of side-effect free rewriting and can be defined as follows:

Definition 2.1.1. An SqPO rewriting corresponding to the application of $r : L \leftarrow r^- - P - r^+ \rightarrow R$ through a matching $m : L \rightarrowtail G$ is *reversible*, if the left square in the following diagram is also a pushout and $P \rightarrowtail m^- \rightarrow G^- - g^+ \rightarrow G^+$ is the pullback complement of $P - r^+ \rightarrow R \rightarrowtail m^+ \rightarrow G^+$. We call $r^{-1} : R \leftarrow r^+ - P - r^- \rightarrow L$ the *reverse* of r.

$$\begin{array}{ccccc} L & \xleftarrow{r^-} & P & \xrightarrow{r^+} & R \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m^-} & & \downarrow{\scriptstyle m^+} \\ G & \xleftarrow[g^-]{} & G^- & \xrightarrow[g^+]{} & G^+ \end{array} \tag{2.4}$$

Remark 2.1.2. Given a rewrite of an object $G \rightsquigarrow G^+$ as in Diagram 2.5, the rewrite $G^+ \rightsquigarrow \bar{G}$ from diagram 2.6 produces an object $\bar{G}$ isomorphic to G.

$$\begin{array}{ccccc} L & \xleftarrow{r^-} & P & \xrightarrow{r^+} & R \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m^-} & & \downarrow{\scriptstyle m^+} \\ G & \xleftarrow[g^-]{} & G^- & \xrightarrow[g^+]{} & G^+ \end{array} \tag{2.5}$$

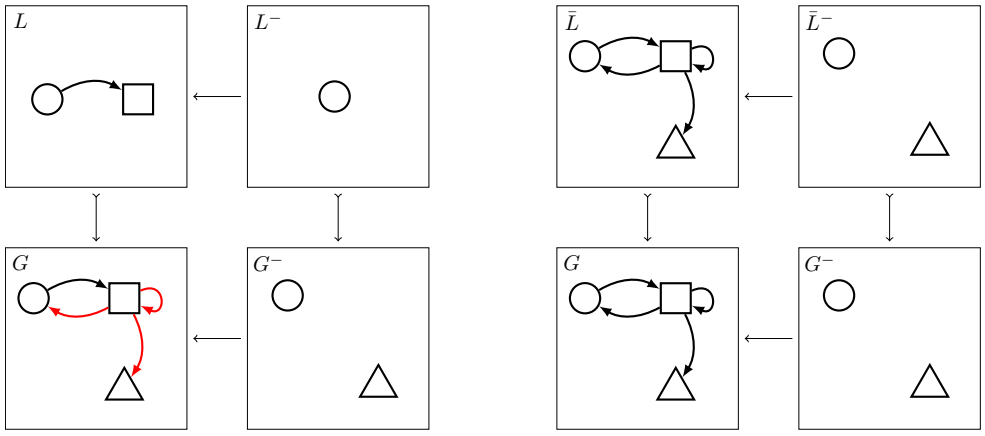
$$\begin{array}{ccccc} R & \xleftarrow{r^-} & P & \xrightarrow{r^+} & L \\ {\scriptstyle m^+}\downarrow & & \downarrow{\scriptstyle m^-} & & \downarrow{\scriptstyle m} \\ G^+ & \xleftarrow[\bar{g}^+]{} & \bar{G}^- & \xrightarrow[\bar{g}^-]{} & \bar{G} \end{array} \tag{2.6}$$

Constructing reversible rules in SimpGrph

For our practical applications we would like to develop a constructive procedure that, given an arbitrary SqPO rewriting rule and a matching, allows to compute its reversible version. We refer to the resulting rule as the *reversible rule refinement*. In the following subsection we consider an example of such a procedure for the category of simple graphs, where the operations of node removal and merging potentially introduce side-effects. This procedure can be easily extended to non-simple graphs and graphs with attributes. This subsection contains some tedious constructions used to obtain reversible rule refinements (in particular, implemented in the `ReGraph` library, discussed in 2.3). It also provides concrete proofs manipulating directly with graph structures and not categorical notions. Therefore, the reader not particularly interested in the details of such constructions is invited to skip this subsection.

To formulate our procedure we will view rules as simply arrows $r^- : L \leftarrow L^-$ and $r^+ : L \rightarrow L^+$ defining respectively restrictive and expansive rules. Then, formally, the goal of our procedure can be formulated separately for restrictive and expansive rules.

Reversible rule refinement for restrictive rules. Recall that applications of restrictive rules in the category of simple graphs correspond to the operations of node/edge removal and node cloning. As we have previously mentioned, comparing the SqPO and the DPO rewriting approaches, the SqPO rewriting allows to perform 'deletion in unknown context'. Such deletion is precisely the cause of side-effects occurring when applying restrictive rules. Upon a node removal, the rewriting implicitly removes all the edges incident to the removed node (by constructing the final pullback complement), which effectively cleans up our graph of dangling edges. Consider Figure 2.2a illustrating an example of such a side-effect. We are interested in refining our rules in a way that would allow for all the graph transformations to be captured explicitly by these rules. More concretely, in the category of simple graphs we want our rules to define the removal of edges incident to removed nodes explicitly (see example in Figure 2.2b).

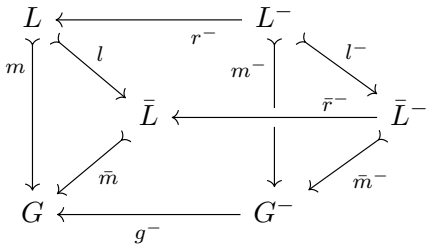


(a) Original rule *(b) Refined rule with no side-effects*

Figure 2.2: Example of node removal side-effects and a rule refinement that removes these side-effects. Graph elements affected with side-effects of rewriting are highlighted with red. Removing a node results into the removal of all its incident edges.

More formally, for restrictive rules we are interested in finding arrows $\bar{r}^- : \bar{L} \leftarrow \bar{L}^-$, $\bar{m} : \bar{L} \rightarrowtail G$ and $l : L \rightarrowtail \bar{L}$ such that

- $m = \bar{m} \circ l$,
- arrows $\bar{m}^-$ and g^- define the final pullback complement of $\bar{r}^-$ and $\bar{m}$,
- and arrows $\bar{m}$ and g^- define the pushout from $\bar{r}^-$ and $\bar{m}^-$ (see Diagram 2.7).



(2.7)

First of all, let as define the set of nodes removed by the rule as

$$V^- = \{v \in V_G \mid v \in img(m) \text{ and } \nexists w \in V_{L^-} : r^-(w) = m^{-1}(v)\}.$$

Note that the homomorphism m is a mono, therefore for every $v \in V_G$ such that $v \in img(m)$ $m^{-1}(v)$ corresponds to a single node in L. The set of nodes to be added to the refined left-hand side of the rule is then defined as

$$V^+ = \{v \in V_G \mid v \notin im(m) \text{ and } \exists w \in V^- : (v, w) \in E_G \text{ or } (w, v) \in E_G\}.$$

This set corresponds to all the nodes, not matched by the original rule, that are adjacent to the removed nodes. Then, the set of nodes of the refined left-hand side $\bar{L}$ is given by $V_{\bar{L}} := V_L \cup V^+$. We can then compute the set of edges of $\bar{L}$ as $E_{\bar{L}} := E_L \cup E^+$, where

$$E^+ = \{(u, v) \in V_{\bar{L}} \times V_{\bar{L}} \mid (u, v) \notin E_L, (\bar{m}(u), \bar{m}(v)) \in E_G \text{ and } (u \in V^- \text{ or } v \in V^-)\}.$$

The set E^+ captures all the edges incident to the removed nodes that were not present in the original left-hand side. The homomorphism $\bar{m}$ is defined as $\bar{m}(v) = v$ for all $v \in V^+$ and as

$\bar{m}(v) = m(v)$ for all $v \in V^L$, while l is simply defined as $l(v) = v$ for all $v \in V^L$. It is easy to verify that, by construction, these homomorphisms satisfy $m = \bar{m} \circ l$.

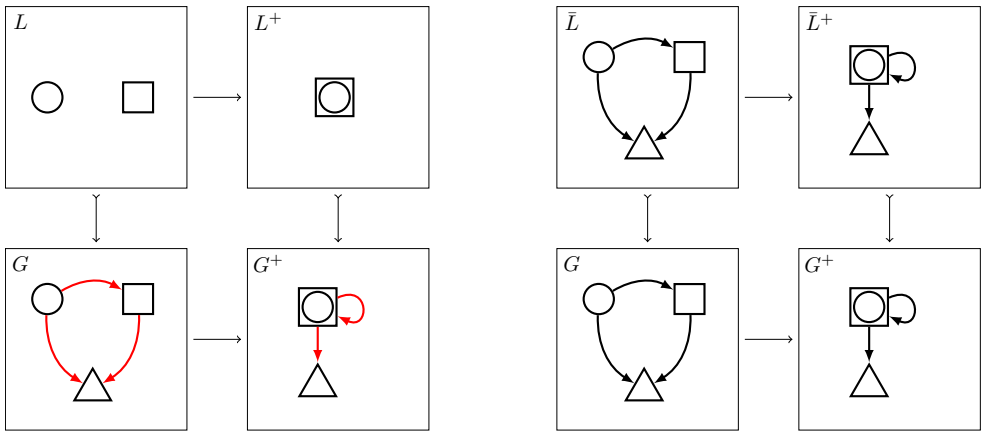
To construct $\bar{L}^-$ and $\bar{r}^- : \bar{L}^- \to \bar{L}$ we will now find the final pullback complement from r^- and l corresponding to the top square in Diagram 2.8. The application of $\bar{r}^-$ through $\bar{m}$, i.e. the final pullback complement of these arrows corresponding to the bottom square in the diagram, produces the graph G^- by the vertical pasting lemma (see Lemma A.5.4).

$$\begin{array}{ccc} L & \xleftarrow{r^-} & L^- \\ {\scriptstyle l}\downarrow & & \downarrow{\scriptstyle l^-} \\ \bar{L} & \xleftarrow{\bar{r}^-} & \bar{L}^- \\ {\scriptstyle \bar{m}}\downarrow & & \downarrow{\scriptstyle \bar{m}^-} \\ G & \xleftarrow[g^-]{} & G^- \end{array} \tag{2.8}$$

Claim 2.1.3. The bottom square in Diagram 2.7 is a pushout.

Proof. See Appendix B. □

Reversible rule refinement for expansive rules. Applications of expansive rules in our category **SimpGrph** perform the operations of node/edge addition and node merging. The latter operation introduces side-effects related to the fact that, by definition of simple graphs, there can be at most one edge between any ordered pair of nodes. As the result of node merging, some of the edges incident to the merged nodes are merged implicitly (i.e. these edge merges are not mentioned explicitly by the rule). Such side-effects make it impossible to reconnect edges incident to the merged nodes upon reversal. Consider Figure 2.3a illustrating an example of such a side-effect. Similarly to the case of restrictive rules, we are interested in refining our rules in a way that would allow for all the graph transformations to be captured explicitly by the underlying rules. More concretely, in the category of simple graphs we want our rules to define the merge of edges incident to merged nodes explicitly (see example in Figure 2.3b).

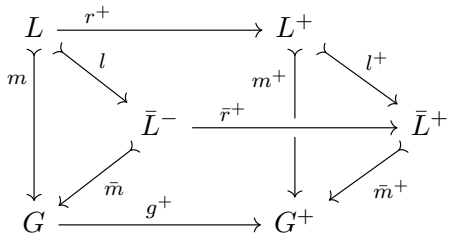


(a) Original rule *(b) Refined rule with no side-effects*

Figure 2.3: Example of node merge side-effects and a rule refinement that removes these side-effects. Graph elements affected by side-effects of rewriting are highlighted with red. Merging of nodes results in the merge of their incident edges with the same sources/targets.

More formally, for expansive rules we are interested in finding arrows $\bar{r}^+ : \bar{L} \to \bar{L}^+$, $\bar{m} : \bar{L} \rightarrowtail G$ and $l : L \rightarrowtail \bar{L}$ such that

- $m = \bar{m} \circ l$,
- arrows g^+ and $\bar{m}^+$ define the pushout from $\bar{m}$ and $\bar{r}^+$,
- and arrows $\bar{m}$ and g^+ define the final pullback complement to the arrows $\bar{r}^+$ and $\bar{m}^+$ (see Diagram 2.9).



(2.9)

We define the set of nodes of G merged by the rule r^+ as

$$V^m = \{v \in V_G \mid \exists w \in V_G : w \neq v \text{ and } r^+(m^{-1}(v)) = r^+(m^{-1}(w))\}$$

Note that the homomorphism m is a mono, therefore, for every $v \in V_G$ such that $v \in img(m)$, $m^{-1}(v)$ corresponds to a single node in L. The set of nodes to be added to the refined left-hand side of the rule is then defined as

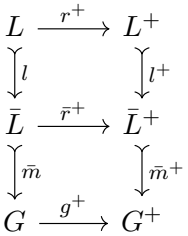
$$V^+ = \{v \in V_G \mid v \notin im(m) \text{ and } \exists w \in V^m : (v, w) \in E_G \text{ or } (w, v) \in E_G\}.$$

This set corresponds to all the nodes, not matched by the original rule, that are adjacent to the merged nodes. Then, the set of nodes of the refined left-hand side $\bar{L}$ is given by $V_{\bar{L}} := V_L \cup V^+$. We can then compute the set of edges of $\bar{L}$ as $E_{\bar{L}} := E_L \cup E^+$, where

$$E^+ = \{(u, v) \in V_{\bar{L}} \times V_{\bar{L}} \mid (u, v) \notin E_L, (\bar{m}(u), \bar{m}(v)) \in E_G \text{ and } (u \in V^m \text{ or } v \in V^m)\}.$$

The set E^+ captures all the edges incident to the merged nodes that were not present in the original left-hand side. The homomorphism $\bar{m}$ is defined as $\bar{m}(v) = v$, for all $v \in V^+$, and as $\bar{m}(v) = m(v)$, for all $v \in V^L$, while l is simply defined as $l(v) = v$, for all $v \in V^L$. It is easy to verify that by construction these homomorphisms satisfy $m = \bar{m} \circ l$.

To construct $\bar{L}^+$ and $\bar{r}^+ : \bar{L} \to \bar{L}^+$ we will now find the pushout from l and r^+ corresponding to the top square in Diagram 2.10. The application of $\bar{r}^+$ through $\bar{m}$, i.e. the pushout of these arrows corresponding to the bottom square of the diagram, produces the graph G^+ by the pasting lemma for pushouts (see Lemma A.4.2).



(2.10)

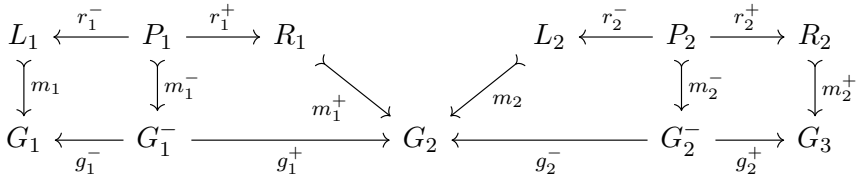
Claim 2.1.4. The arrows $\bar{m}$ and g^+ in Diagram 2.9 define the final pullback complement of $\bar{r}^+$ and $\bar{m}^+$.

Proof. See Appendix B. □

2.1.3 Composition of sesqui-pushout rewriting

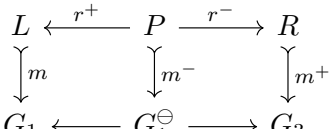
In this subsection we would like to study how consecutive applications of SqPO rules can be composed into a single rewrite. Constructing such compositions of rules is important, for example, when maintaining the history of updates of an object or multiple versions of this object (diverged as the result of conflicting transformations), expressing the operation of 'undoing' a sequence of transformations as a single rewriting rule, etc. Later in this chapter we will see how the graph audit trail (discussed in Section 2.1.4) makes an extensive use of the rule composition techniques presented in this subsection.

Let $r_1 : L_1 \leftarrow r_1^- - P_1 - r_1^+ \rightarrow R_1$ be a rewriting rule applied to an object G_1 through a match $m_1 : L_1 \rightarrowtail G_1$ and let G_2 be the result of application of this rule (corresponding to the left-most SqPO diagram in Diagram 2.11). Let $r_2 : L_2 \leftarrow r_2^- - P_2 - r_2^+ \rightarrow R_2$ be a rule applied to the resulting object G_2 through a match $m_2 : L_2 \rightarrowtail G_2$ (as the right-most SqPO diagram in Diagram 2.11).



(2.11)

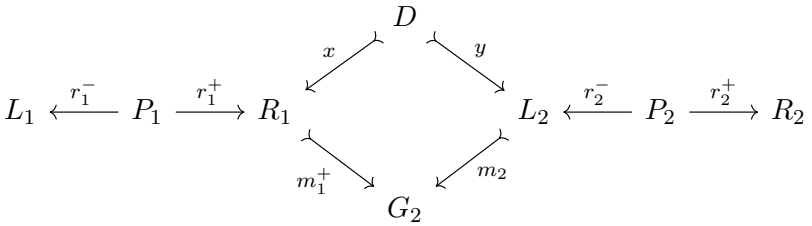
Given these two consecutive rule applications, we would like to find a rule $r : L \leftarrow r^- - P - r^+ \rightarrow R$ and a matching $m : R \rightarrowtail G_1$ that, applied to G_1, directly produces the object G_3, i.e. such that Diagram 2.12 is an SqPO diagram.



(2.12)

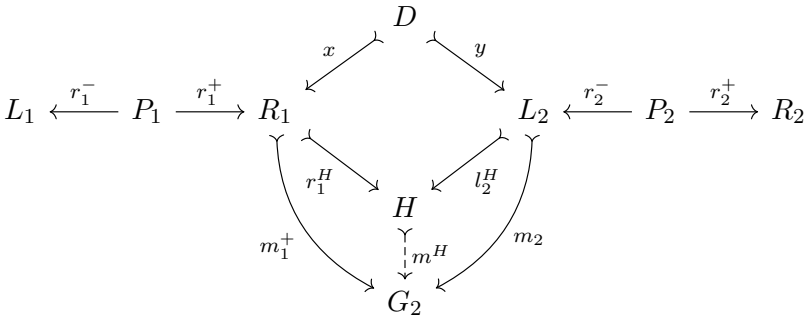
Apart from being well-structured for SqPO rewriting, construction of such a composed rule will require from the category in wich we are working to be *adhesive* [61]. Moreover, we will require the *pushout factorizations of pullbacks* along monos to be *monic* (see Definition A.4.8). This is always the case in the adhesive categories (see Theorem 5.1 [62]).

First of all, we construct the pullback $R_1 \leftarrow x \prec D \succ y \rightarrow L_2$ from $R_1 \succ m_1^+ \rightarrow G_2 \leftarrow m_2 \prec L_2$ as in Diagram 2.13. Note that arrows x and y are monos by Lemma A.3.4. We will call the span given by this pullback the *overlap* of R_1 and L_2 given their matching inside G_2, and we will denote it with o. Intuitively this object indicates whether the two rule applications are *sequentially independent* (see [21, 29] for more details on sequential independence of rewriting and its consequences), i.e. whether the two rules operate on disjoint parts of G_2. When D is non-empty, for example, the first rule can produce elements that are 'consumed' by the second rule.



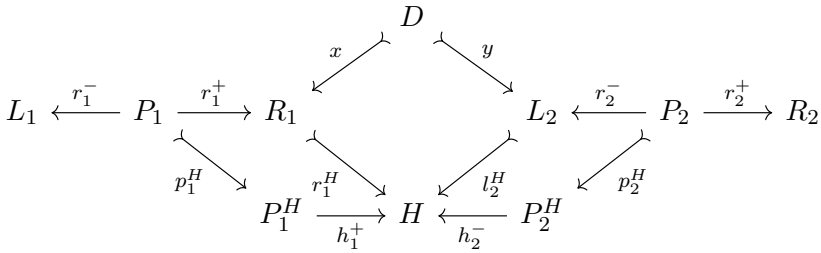
(2.13)

The pushout $R_1 \succ l_1^H \rightarrow H \leftarrow l_2^H \prec L_2$ from $R_1 \leftarrow x \prec D \succ y \rightarrow L_2$ (as in Diagram 2.14) constructs the object H that intuitively represents the union of two patterns R_1 and L_2 given their overlap o. By the universal property of pushouts, there exists a unique homomorphism $m^H : H \rightarrow G_2$, the pushout factorization of the pullback of m_1^+ and m_2, that renders the diagram commutative. Because m_1^+ and m_2 are monos, the homomorphism m^H is also a mono.



(2.14)

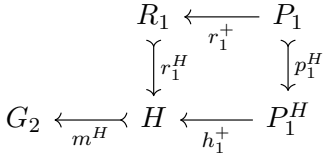
Using the object H we now construct two final pullback complements: $P_1 \rightarrowtail p_1^H \to P_1^H - h_1^+ \to H$ from $P_1 - r_1^+ \to R_1 \rightarrowtail r_1^H \to H$ and $P_2 \rightarrowtail p_2^H \to P_2^H - h_2^- \to H$ from $P_2 - r_2^- \to L_2 \rightarrowtail l_2^H \to H$ as in Diagram 2.15.



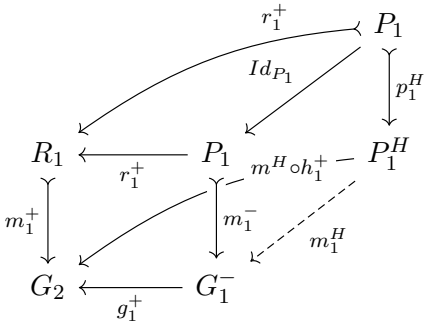
(2.15)

How can we interpret these constructions? The second pullback complement applies the rewrite specified by the arrow r_2^- to H. Now, for the first pullback complement to be meaningful, we need to make the assumption that the rewriting given by the rule r_1 is reversible. More specifically, we will assume that $L_1 \rightarrowtail m_1 \to G_1 \leftarrow g_1^- - G_1^-$ from Diagram 2.11 is the pushout from $L_1 \leftarrow r_1^- - P_1 \rightarrowtail m_1^- \to G_1^-$ and $P_1 \rightarrowtail m_1^- \to G_1^- - g_1^+ \to G_2$ is the final pullback complement from $P_1 - r_1^+ \to R_1 \rightarrowtail m_1^+ \to G_2$. Later, in the proof of Theorem 2.1.5, we will see a formal reason for making this assumption. Having made this assumption, the object P_1^H can be interpreted as the result of 'undoing' the rewrite specified by r_1^+ on H.

By the previously made assumption, $P_1 \rightarrowtail m_1^- \to G_1^- - g_1^+ \to G_2$ is the final pullback complement from $P_1 - r_1^+ \to R_1 \rightarrowtail m_1^+ \to G_2$. We would like to use the universal property of final pullback complements and show that there exists a unique arrow $m_1^H : P_1^H \to G_1^-$ that renders Diagram 2.17 commutative. By Lemma A.3.5 a pullback square post-composed with a mono is a pullback, therefore Diagram 2.16 is a pullback. This allows us to apply the above-mentioned universal property and show the existence of the unique m_1^H that renders Diagram 2.17 commutative.

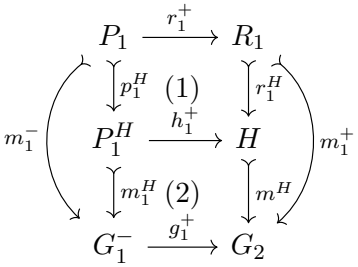


(2.16)



(2.17)

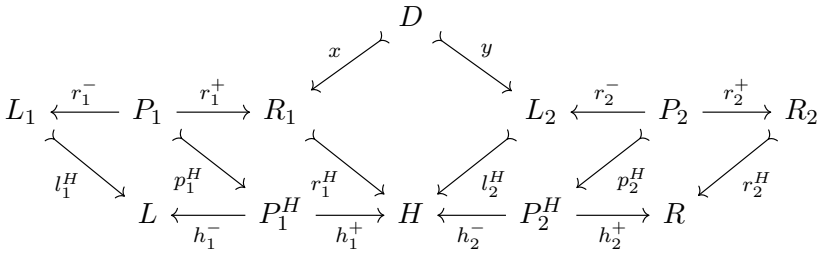
Observe Diagram 2.18, its outer square and the square (1) are final pullback complements by construction. By vertical pasting lemma for pullback complements (Lemma A.5.4), the square (2) is also a final pullback complement, and therefore a pullback. Recall that pullbacks preserve monos, $m^H : H \rightarrowtail G_2$ is a mono, and therefore m_1^H is a mono as well.



(2.18)

Analogous arguments can be used to show that there exists a unique mono $m_2^H : P_2^H \rightarrowtail G_2^-$ such that $g_2^- \circ m_2^H = m^H \circ h_2^-$ and $m_2^H \circ p_2^H = m_2^- \circ Id_{P_2}$.

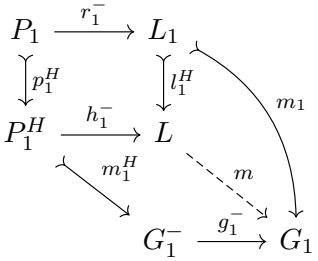
Next, we construct two pushouts: $L_1 \rightarrowtail l_1^H \rightarrow L \leftarrow h_1^- - P_1^H$ from r_1^- and p_1^H and $R_2 \rightarrowtail r_2^H \rightarrow R \leftarrow h_2^+ - P_2^H$ from r_2^+ and p_2^H as in Diagram 2.19. These pushout give us two objects L and R. Intuitively the first pushout 'undoes' the rewrite of P_1^H specified by r_1^-, and the second performs the rewrite of P_2^H using r_2^+ and p_2^H.



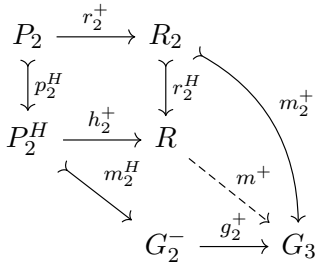
(2.19)

By the universal property of these pushouts there exist unique arrows $m : L \rightarrow G_1$ and $m^+ : R \rightarrow G_3$ that render Diagrams 2.20 and 2.21 commutative. Moreover, by Corollary A.4.3, $G_1^- - g_1^- \rightarrow G_1 \leftarrow m - L$ is the pushout from m_1^H and h_1^-; and $G_2^- - g_2^+ \rightarrow G_3 \leftarrow m^+ - R$ is the pushout from m_2^H and h_2^+. Recall that pushouts in adhesive categories preserve monos (see Lemma A.6.3), therefore m and m^+ are monos.

The constructed object L can be seen as the result of reversing the transformation of the union pattern H specified by r_1^- and r_1^+ with the matching r_1^H, which is possible because of the reversibility of the rule given by r_1^- and r_1^+. The object R, on the other hand, corresponds to the application of r_2^- and r_2^+ to H through the matching l_2^H.



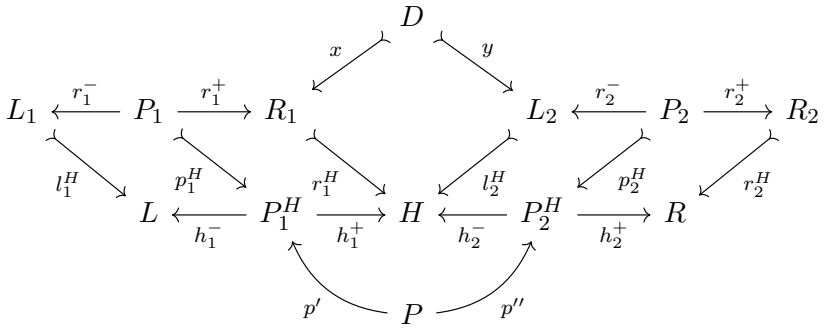
(2.20)



(2.21)

Finally, to construct the rule composition, we find the pullback $P_1^H \leftarrow p' - P - p'' \rightarrow P_2^H$ from h_1^+ and h_2^-. The resulting rule corresponds to the span $L \leftarrow h_1^- \circ p' - P - h_2^+ \circ p'' \rightarrow R$. We will refer

to it as the *composed rule* given the overlap o and write $r = \otimes(r_1, o, r_2)$.



(2.22)

Theorem 2.1.5. *In adhesive categories (see Appendix A.6.2), if rewriting given by r_1 is reversible, application of the rule r given by $\otimes(r_1, o, r_2)$ with the match $m : L \rightarrowtail G_1$ results into a rewrite $G_1 \rightsquigarrow G_3$, i.e. the following diagram is an SqPO diagram:*

$$\begin{array}{ccccc} L & \xleftarrow{h_1^- \circ p'} & P & \xrightarrow{h_2^+ \circ p''} & R \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle m^-} & & \downarrow{\scriptstyle m^+} \\ G_1 & \xleftarrow[g^-]{} & G_1^{\ominus} & \xrightarrow[g^+]{} & G_3 \end{array} \tag{2.23}$$

Proof. See Appendix B. □

Lemma 2.1.6. In adhesive categories, the composition of two reversible rewrites is a reversible rewrite.

Proof. See Appendix B. □

2.1.4 Audit trail for graph rewriting

As we have previously discussed in Subsection 1.1.2, we would like to design an audit trail system for our KR framework. This audit trail should record a history of graph transformations, provide means for rolling back to any previous state in this history and be able to maintain and merge multiple diverged versions of the graph. In this subsection we describe how reversibility and composition of rewriting can be used to construct the audit trail for transformations of individual objects (e.g. graphs with attributes used as building blocks of our KR system). The presented audit trail is implemented as part of the `ReGraph` library and discussed in more detail in Section 2.3.4.

Let G^0 be the starting object whose history of transformations we would like to maintain and let $\langle r_i : L^i \leftarrow r_i^- - P^i - r_i^+ \rightarrow R^i \mid i \in [1 \dots n] \rangle$ be a sequence of rules consecutively applied to G^0 through the instances $m_i : L^i \rightarrowtail G^{i-1}$, resulting in a sequence of objects $\langle G^i \mid i \in [1 \dots n] \rangle$ with $m_i^+ : R^i \rightarrowtail G^i$ for $1 \leq i \leq n$, i.e. such that for every $1 \leq i \leq n$, Diagram 2.24 is a SqPO diagram. To be able to build a sound audit trail, we additionally require such a sequence of

rewrites to be reversible.

$$\begin{array}{ccccc} L^i & \xleftarrow{r_i^-} & P^i & \xrightarrow{r_i^+} & R^i \\ \downarrow{\scriptstyle m_i} & & \downarrow{\scriptstyle m_i^-} & & \downarrow{\scriptstyle m_i^+} \\ G^{i-1} & \xleftarrow[\bar{g}_i^-]{} & \bar{G}^{i-1} & \xrightarrow[\bar{g}_i^+]{} & G^i \end{array} \tag{2.24}$$

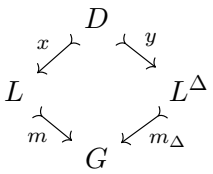
Definition 2.1.7. The *audit trail* for the resulting object G^n consists of the sequence of rules $\langle r_i \mid i \in [1 \dots n] \rangle$ and the right-hand side instances $m_i^+ : R^i \rightarrowtail G^i$ for $1 \leq i \leq n$.

Rollback. Using such an audit trail we can *rollback* rewriting to any point in the history of transformations corresponding to some intermediate object G^i for $0 \leq i \leq n-1$ by applying the sequence rules $\langle r_j^{-1} \mid j \in [n \dots i+1] \rangle$ with the corresponding instances $m_j^+ : R^j \rightarrowtail G^j$, where $i+1 \leq j \leq n$.

Maintain diverged versions. To maintain multiple diverged versions of an object in the audit trail, we use a technique known from VC systems as *delta compression*, i.e. only the current version of the object is stored at any moment, while the other versions are encoded in a *delta*, a representation of the 'difference' between versions.

Let v_1 and v_2 be two versions of the starting object G^0 with v_1 being the current version. The initial delta Δ from v_1 to v_2 is set to the identity rule (the rule that does not perform any transformations) $\emptyset \leftarrow Id_\emptyset - \emptyset - Id_\emptyset \rightarrow \emptyset$ and the instance $u : \emptyset \rightarrowtail G^0$, where $\emptyset$ stands for the initial object in $\mathbf{C}$ and u is the unique arrow from the initial object to G^0 (see Appendix A.2 for the definition of initial objects). Every rewrite of the current version of the object induces an update of the delta that consists of the composition of the previous delta and the reverse of the applied rule (recall that we assume that every rewriting in the audit trail is reversible).

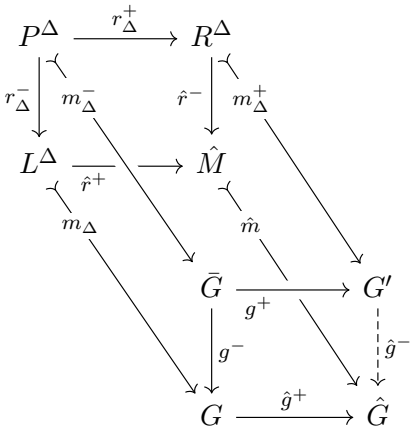
As before, let v_1 be the current version corresponding to some object G (e.g. obtained as the result of transformation of the initial object G^0) and let $r_\Delta : L^\Delta \leftarrow r_\Delta^- - P^\Delta - r_\Delta^+ \rightarrow R_\Delta$ and $m_\Delta : L^\Delta \rightarrowtail G$ be respectively the rule and the instance given by Δ. Let $r : L \leftarrow r^- - P - r^+ \rightarrow R$ be a rule applied to G through the instance $m : L \rightarrowtail G$ and G' be the result of application of r given m. To update the delta, we compute the composition $\otimes(r^{-1}, o, r^\Delta)$ with o being a span $L \leftarrow x \prec D - y \rightarrow L^\Delta$ obtained as a PB in Diagram 2.25. The new delta is, thus, set to the rule and the instance given by the composition $\otimes(r^{-1}, o, r_\Delta)$.



(2.25)

Switch version. Switching between different versions of the object can be done by simply applying the rule through the instance given by the delta. Namely, if v_1 is the current version corresponding to an object G with the delta to v_2 given by $\Delta = (r_\Delta, m_\Delta)$, switching to v_2 is performed by applying r_Δ to G through the instance m_Δ. If G' is the result of the above-mentioned rewriting and $m_\Delta^+ : R^\Delta \rightarrowtail G'$ is its right-hand side instance, then v_2 becomes the current version of the object and the new delta Δ is set to $(r_\Delta^{-1}, m_\Delta^+)$.

Merge versions. Let v_1 be the current version corresponding to an object G, v_2 be another version corresponding to an object G' and the delta between v_1 and v_2 be given by $\Delta = (r_\Delta, m_\Delta)$. The left and top faces of the cube in Diagram 2.26 correspond to the two phases of the application of r_Δ through m_Δ. *The canonical merging rules* for v_1 and v_2 are given by two arrows $\hat{r}^+$ and $\hat{r}^-$ constructed by the pushout $L^\Delta - \hat{r}^+ \rightarrow \hat{M} \leftarrow \hat{r}^- - R^\Delta$ from r_Δ^- and r_Δ^+ (see the back face of the cube in the diagram). The merging rule for the current object G is then applied by finding the pushout $G - \hat{g}^+ \rightarrow \hat{G} \leftarrow \hat{m} \prec \hat{M}$ from m_Δ and $\hat{r}^+$ as in the bottom face of Diagram 2.26. By the universal property of pushouts there exists a unique arrow $\hat{g}^- : G' \rightarrow \hat{G}$ that renders the cube commutative. The object $\hat{G}$ is the result of the canonical merging of G and G'.



(2.26)

Note that, because the application of r_Δ is reversible, the left face is also a pushout, which implies that the right face of the cube is also a pushout. Therefore, we can obtain the merged object $\hat{G}$ by applying the merging rule $\hat{r}^-$ to G' through the instance m_Δ^+.

Non-canonical merging rules are given by two arrows $r_M^+ : L^\Delta \rightarrow M$ and $r_M^- : R^\Delta \rightarrow M$ such that $r_M^+ \circ r_\Delta^- = r_M^- \circ r_\Delta^+$. The merged object G_M is, thus, obtained by applying the rule r_M^+ to G through the instance m_Δ or the rule r_M^- to G' through the instance m_Δ^+.

2.2 Hierarchies and rewriting in hierarchies

We have previously stated that our goal is to be able to divide knowledge into distinct fragments, relate and update them using well-defined mechanisms that preserve consistent relations.

In the previous section we have introduced the main building blocks of our KR system: graphs that can be used to represent fragments of knowledge, homomorphisms to relate these fragments and SqPO rewriting to update them. In this section we define the notion of a *hierarchy of graphs*, a directed acyclic structure consisting of graphs and homomorphisms. This structure can be viewed as a diagram where all the triangles and squares formed by the edges of the hierarchy are required to commute (we will call this the *consistency* requirement). Here we describe mechanisms for transformation of individual graphs in a hierarchy, which may require *propagation* to other graphs, in order to maintain the consistency of the hierarchy. The constructions presented in this section are formulated abstractly and can be instantiated not only for graphs, but, in general, any objects in categories with a certain structure (in particular, that allows to perform sesqui-pushout rewriting).

Intuitively, from the point of view of KR, it is often useful to interpret an individual graph homomorphism as specifying a relation between a more concrete and a more abstract representation of knowledge. Therefore, moving in a hierarchy along the direction of its edges would mean moving away from more concrete to more abstract representations. The aforementioned consistency condition of a graph hierarchy guarantees that the representation of knowledge from some concrete graph obtained by moving along alternative paths leading to the same abstract graph is consistent.

A first elementary example of an application of such a KR system is a two-level hierarchy that can be formulated using the database community nomenclature as a *schema* graph and a data *instance* graph typed by the schema. Schema nodes specify types of enities allowed in the system; its edges specify which edges between different types of nodes are allowed; attributes on its nodes and edges define a set of allowed attributes for nodes and edges of respective types. In this setting, we may be interested in performing an update of the instance graph, which can either respect the schema or violate it. In the latter case we would like to either reject the data update or perform a schema update in a way that would keep the typing in the system valid. The same applies for updates of the schema graph—they can cause the instance graph to not comply with the new version of the schema, in which case an update of the instance is necessary.

Certainly, much more elaborate applications of such a system are conceivable. For example, the bio-curation tool KAMI (discussed in Chapter 4) uses a representation based on a three-level hierarchy to represent cellular signalling knowledge corpora. Its first level consists of the meta-model graph—a small rigid domain-specific graph that represents the kinds of entities that can exist in the system. The second level is given by a graph, called the action graph, that represents knowledge about the actual entities that can exist in the system, e.g. particular proteins, their known domains, interactions with other proteins. Finally, the third level consists of a collection of small graphs, called nuggets, that encode knowledge about interactions that can appear between different types of proteins. The action graph is typed by the meta-model, and every nugget is typed by the action graph (and, indirectly, by the meta-model). The action graph represents a non-trivial summary of knowledge contained in nuggets, and, together with homomophisms from nuggets, it provides relations between entities and actions from different nuggets and defines the way nuggets are interpreted by KAMI. The rewriting techniques presented in this paper are used in KAMI to achieve various goals, e.g. to incrementally build the action graph from input nuggets, to perform bookkeeping updates and to reuse the knowledge in various concrete biological contexts.

2.2.1 Hierarchies

We will define hierarchies of objects in two alternative ways. We will first define them as superstructures, directed acyclic graphs (DAGs) whose nodes are associated with graphs and edges are associated with homomorphisms, and which satisfy certain consistency conditions. We will then illustrate how this definition can be reformulated, in an elegant way, using an abstract hierarchy as a finite category and its instantiation in categories of graphs (simple, non-simple, with or without attributes) as a functor. Note that, however cumbersome the first definition may seem, it provides a good intuition behind graph hierarchies and reliably represents the concrete data structures used in the `ReGraph` implementation of hierarchies discussed in Section 2.3.

Definition 2.2.1. For a fixed category $\mathbf{C}$ a *hierarchy of objects* in $\mathbf{C}$ is a tuple $\mathcal{H} = (V, E, \mathcal{O}, \mathcal{F}, \alpha, \beta)$, where:

- V is the set of hierarchy nodes,
- $E \subseteq V \times V$ is the set of hierarchy edges,
- $\mathcal{O}$ is a set of objects from $\mathbf{C}$,

- $\mathcal{F}$ is a set of homomorphisms from $\mathbf{C}$,
- $\alpha : V \to \mathcal{O}$ is a function that maps hierarchy nodes to hierarchy objects,
- $\beta : E \to \mathcal{F}$ is a function that maps hierarchy edges to homomorphisms,

which satisfies the following properties:

1. $dom(\beta(u, v)) = \alpha(u)$ and $codom(\beta(u, v)) = \alpha(v)$ for all $(u, v) \in E$;
2. *(acyclicity)* graph induced by a tuple (V, E) does not contain directed cycles, we will refer to this graph as the *skeleton* of the hierarchy;
3. *(consistency)* for any two paths $\pi_1 = (e_1, e_2, \ldots, e_n)$ and $\pi_2 = (f_1, f_2, \ldots, f_m)$ from s to t (where $s, t \in V$, $e_k, f_l \in E$, for all $1 \leq k \leq n$ and $1 \leq l \leq m$), the homomorphisms constructed by the composition of the respective homomorphisms along the paths are equal, i.e. for the homomorphisms $h_1 = \beta(e_n) \circ \beta(e_{n-1}) \circ \ldots \circ \beta(e_2) \circ \beta(e_1)$ and $h_2 = \beta(f_m) \circ \beta(f_{m-1}) \circ \ldots \circ \beta(f_2) \circ \beta(f_1)$ it holds that $h_1 = h_2$.

Therefore, a hierarchy is a DAG whose nodes and edges are respectively objects and homomorphisms from $\mathbf{C}$ and such that any two composed homomorphisms from the same source node to the same target node are equal. They can be thought of as commutative diagrams in $\mathbf{C}$. Let us now define few notions that will be useful in the rest of this section.

Definition 2.2.2. The *skeleton* of a hierarchy $\mathcal{H}$ is given by the DAG $\mathcal{G} = (V, E)$.

Definition 2.2.3. Let $\pi = (e_1, e_2, \ldots, e_n)$ be a path in a hierarchy $\mathcal{H}$, a *path homomorphism* $\circ\pi$ is given by the composition of the homomorphisms along π, i.e. $\circ\pi = \beta(e_n) \circ \beta(e_{n-1}) \circ \ldots \circ \beta(e_1)$.

For any node $u, v \in V$ in $\mathcal{H}$, let us write $u \leq_{\mathcal{H}} v$ if there exists a path from u to v.

Definition 2.2.4. For any vertex $v \in V$ in a hierarchy $\mathcal{H}$, the set of its *ancestors* is defined as $anc(u) = \{v \in V : v \leq_{\mathcal{H}} u\}$.

Definition 2.2.5. For any vertex $v \in V$ in a hierarchy $\mathcal{H}$, the set of its *descendants* is defined as $desc(u) = \{v \in V : u \leq_{\mathcal{H}} v\}$.

Remark 2.2.6. Due to the fact that the skeleton of a graph hierarchy is acyclic, for any vertex $u \in V$ it holds that $anc(u) \cap desc(u) = \emptyset$.

Let us now reformulate the definition of a hierarchy of objects from some category $\mathbf{C}$.

Definition 2.2.7. An *abstract hierarchy* is a finite category $\mathbf{H}$ freely generated from a DAG.

Definition 2.2.8. A *hierarchy of objects* in a category $\mathbf{C}$ is a functor $\mathcal{H} : \mathbf{H} \to \mathbf{C}$ from an abstract hierarchy $\mathbf{H}$ to $\mathbf{C}$.

It is not hard to verify that by the definition of a functor Definition 2.2.1 coincides with Definition 2.2.8. More precisely, the consistency condition in the first definition is encoded within the definition of a functor. Now, the concrete graph hierarchies of interest include:

- hierarchies of simple graphs $\mathcal{H} : \mathbf{H} \to \mathbf{SimpGrph}$,
- hierarchies of non-simple graphs $\mathcal{H} : \mathbf{H} \to \mathbf{Grph}$,

- hierarchies of simple graphs with attributes $\mathcal{H} : \mathbf{H} \to \mathbf{SimpGrph}_{attrs}$,
- hierarchies of non-simple graphs with attributes $\mathcal{H} : \mathbf{H} \to \mathbf{Grph}_{attrs}$.

Let us assume that we are working in a category $\mathbf{C}$ where SqPO rewriting is well-defined (a category with pushouts and pullback complements along monos). In the rest of this section we will describe the designed mechanisms for rewriting of a single object situated at a vertex of a hierarchy $\mathcal{H}$ defined over $\mathbf{C}$ preserving the structure and the consistency of $\mathcal{H}$.

We will first consider two objects G, T and their homomorphism $h : G \to T$ (or a simple hierarchy consisting of two nodes and one edge corresponding to G, T and h). We will proceed by studying two scenarios, the first one we call *backward propagation* and the second one we call *forward propagation*. In the first scenario we would like to apply a restrictive rule to T, while in the second we apply an expansive rule to G.

After this we will describe how, having applied a general SqPO rewriting rule to an individual object situated in an arbitrary hierarchy, these two scenarios can be extrapolated to perform an update of both hierarchy objects and homomorphisms, while maintaining the structure and consistency of the original hierarchy.

2.2.2 Backward propagation

In the rest of this subsection we fix two objects G, T and a homomorphism $h : G \to T$, we will apply a restrictive rule $r : L \leftarrow L^-$ to the object T through a mono $m : L \rightarrowtail T$. Recall that the application of a restrictive SqPO rule corresponds to the final pullback complement from $L^- \mathrel{-r\to} L \mathrel{\succ m\to} T$ as in the diagram below. We can now construct the pullback of h and m and compute a span $G \mathrel{\leftarrow \hat{m} \prec} L_G \mathrel{-\hat{h}\to} L$. The object L_G can be interpreted as the subobject of G whose typing may be affected by the rewriting of T.

$$\begin{array}{ccc} L & \xleftarrow{r} & L^- \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle m^-} \\ T & \xleftarrow{t} & T^- \end{array} \qquad (2.27) \qquad\qquad \begin{array}{ccc} L_G & \xrightarrow{\hat{h}} & L \\ \downarrow{\scriptstyle \hat{m}} & & \downarrow{\scriptstyle m} \\ G & \xrightarrow{h} & T \end{array} \qquad (2.28)$$

To restore the hierarchy edge that corresponds to $h : G \to T$ we would like to propagate a rewrite $T \rightsquigarrow T^-$ to G using this subobject L_G. This propagation is necessary as the application of a restrictive rule to the typing graph T may result in the removal or cloning of its elements (in this thesis we will use the term element to refer to any concrete constituent of an object in a concrete category of interest to us, for example, a node or edge of a graph). The main idea behind propagation can be formulated as follows. We decompose a rewrite $T \rightsquigarrow T^-$ into two steps $T \sim(1)\rightsquigarrow T' \sim(2)\rightsquigarrow T^-$. In the first step we will specify some arrow $L_G \to T'$ that would allow us to 'retype' G by T', we refer to this step as the *strict rewrite*. In the second step we propagate the transformation $T' \rightsquigarrow T^-$ to all the instances of the affected part of T' in G, we refer to this step as the *canonical backward propagation*. To decompose our original rewrite $T \rightsquigarrow T^+$ we specify a *backward factorization* of the rule r defined as follows.

Definition 2.2.9. Given a rule $r : L \leftarrow L^-$, its *backward factorization* is given by an object L'

and arrows $r' : L \leftarrow L'$, $r^- : L' \leftarrow L^-$ and $\hat{h}' : L_G \to L'$ such that $r = r' \circ r^-$ and $\hat{h} = r' \circ \hat{h}'$.

$$\begin{array}{ccc} L & \xleftarrow{r} & L^- \\ {\scriptstyle \hat{h}}\uparrow & \nwarrow^{r'} & \downarrow{\scriptstyle r^-} \\ L_G & \xrightarrow[\hat{h}']{} & L' \end{array} \tag{2.29}$$

The *strict rewrite* of T is defined by taking the final pullback complement from r' and m. By the definition of backward factorization and the universal property of the final pullback complement 2.30 to the pullback in 2.28, we obtain a unique typing $h' : G \to T'$ that renders Diagram 2.31 below commutative. Therefore, given $\hat{h}'$, we are able to 'retype' G by the intermediary result of rewriting T'.

$$\begin{array}{ccc} L & \xleftarrow{r'} & L' \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m'} \\ T & \xleftarrow[t']{} & T' \end{array} \tag{2.30}$$

$$\begin{array}{ccccc} L_G & & & \xrightarrow{\hat{h}'} & \\ {\scriptstyle \hat{m}}\downarrow & \searrow^{\hat{h}} & & & \\ G & & L & \xleftarrow{} & L' \\ & \searrow_{h} & {\scriptstyle h'}\downarrow & & \downarrow \\ & & T & \xleftarrow{} & T' \end{array} \tag{2.31}$$

The *canonical rewrite* of T is defined by taking the final pullback complement from r^- and m' and the corresponding *backward propagation* by constructing the pullback from h' and t^- as in the diagrams below. As the result of such propagation, we obtain an updated object G^- and its typing by the final result of rewriting T^-.

$$\begin{array}{ccc} L' & \xleftarrow{r^-} & L^- \\ {\scriptstyle m'}\downarrow & & \downarrow{\scriptstyle m^-} \\ T' & \xleftarrow[t^-]{} & T^- \end{array} \tag{2.32}$$

$$\begin{array}{ccc} G & \xrightarrow{g^-} & G^- \\ {\scriptstyle h'}\downarrow & & \downarrow{\scriptstyle h^-} \\ T' & \xrightarrow[t^-]{} & T^- \end{array} \tag{2.33}$$

Proposition 2.2.10 shows that splitting a rewrite into the strict and canonical propagation phases produces the same result as direct application of the original rule.

Proposition 2.2.10. Given a rule $r : L \leftarrow L^-$ and its backward factorization specified by $L \leftarrow r' - L' \leftarrow r^- - L^-$ and $\hat{h}' : L_G \to L'$, an application of r through m can be decomposed into an application of r' through m followed by an application of r^- through m', corresponding to the final pullback complements, as in the diagram below.

$$\begin{array}{ccccc} & & r & & \\ & \swarrow & & & \\ L & \xleftarrow{r'} & L' & \xleftarrow{r^-} & L^- \\ {\scriptstyle m}\downarrow & & {\scriptstyle m'}\downarrow & & \downarrow{\scriptstyle m^-} \\ T & \xleftarrow[t']{} & T' & \xleftarrow[t^-]{} & T^- \\ & \nwarrow & & \swarrow & \\ & & t & & \end{array} \tag{2.34}$$

Proof. The proof follows immediately from the horizontal pasting lemma for final pullback complements (Lemma A.5.3). □

An alternative way to propagate a rewrite specified by r^- to G consists in constructing a rule that applies directly to G. We refer to such a rule as a *rule lifting*.

Definition 2.2.11. The *lifting* $\hat{r}^- : L_G \leftarrow L_G^-$ of r^- is constructed as the pullback from $\hat{h}'$ and r^-.

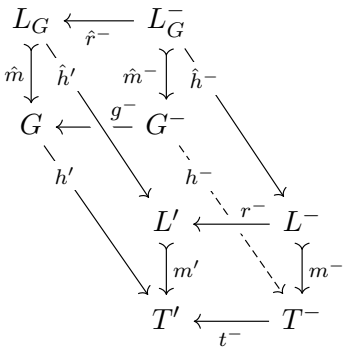
$$\begin{array}{ccc} L_G & \xleftarrow{\hat{r}^-} & L_G^- \\ {\scriptstyle \hat{h}'}\downarrow & & \downarrow{\scriptstyle \hat{h}^-} \\ L' & \xleftarrow[r^-]{} & L^- \end{array} \tag{2.35}$$

Theorem 2.2.12. *The graph G^- and the morphism $g^- : G \to G^-$ from 2.31 can be obtained by constructing the final pullback complement to $\hat{r}^-$ and $\hat{m}$ as in the following diagram.*

$$\begin{array}{ccc} L_G & \xleftarrow[\hat{r}^-]{} & L_G^- \\ {\scriptstyle \hat{m}}\downarrow & & \downarrow{\scriptstyle \hat{m}^-} \\ G & \xleftarrow[g^-]{} & G^- \end{array} \tag{2.36}$$

Proof. See Appendix B. □

If we construct G^- and g^- as the final pullback complement to $\hat{r}^-$ and $\hat{m}$, the arrow $h^- : G^- \to T^-$ can be found by the universal property of the final pullback complement that constructed T^- as in the diagram below.



(2.37)

Proposition 2.2.13. The two definitions of propagation coincide.

Proof. Consider the cube in Diagram 2.37. Let its front face define the canonical rewrite of T using r^- and m, its bottom face give the backward propagation to G and, finally, its top face be the lifting of r^-. We observe that the left, top and bottom faces of the cube are, therefore, pullbacks. By the pasting lemma for pullbacks, the right face is also a pullback. Finally, because the front face is a pullback complement, by Lemma A.5.5, the back face is a pullback complement as well. □

Example 2.2.1. *Figure 2.4 illustrates the presented constructions for sets (the reader may also think of it as graphs with no edges). Note that a homomorphism $h : G \to T$ in* **Set** *provides the* intentional representation of a multi-set. *The rule factorization splits the rewriting of T into two phases: in the first phase cloning of the square element (into the white and the black square) is performed, while removal of the circle is postponed to the second phase. Now the arrow $L_G \to L'$ specifies that the instance of the square in G denoted with (1) is typed by the white square and the instance (2) is typed by the black square. The strict phase of rewriting performs the cloning and retypes G by the intermediary result T'. The canonical backward propagation phase removes the circle from T' and all its instances in G.*

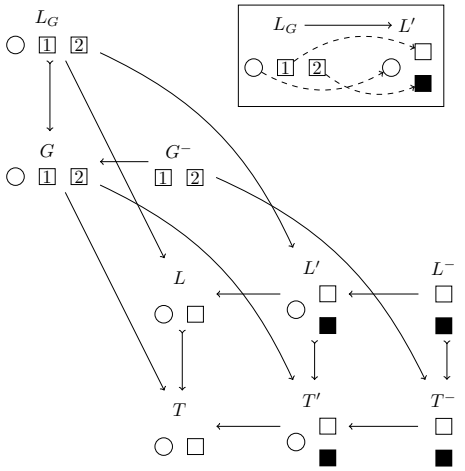


Figure 2.4: Example of backward propagation

A scenario where typing elements are being cloned is often referred to as concept refinement*, i.e. splitting given concepts into some more fine-grained ones. The strict phase of our rewriting allows us to 'protect' elements of G from being cloned; instead, it uses the information provided in $L_G \to L'$ to retype those elements with the more refined types from T'.*

Alternatively, using the same rule factorization, we construct the lifting of $L' \leftarrow L^-$ corresponding to the arrow $L_G \leftarrow L_G^-$ in Figure 2.5. This lifting corresponds to a rule that can be directly applied to G and that, in this example, removes the instances of the circle node.

Example 2.2.2. *In Figure 2.6 we apply the same rule as in Example 2.2.1, i.e. we clone the square node and remove the circle from T. We also use the same rule factorization. In this example, however, the graph G is slightly modified, i.e. it contains three instances of the square node: (1), (2) and (3). We now would like to re-use the information from the previous example, namely, that the instance (1) is retyped with the white square and (2) with the black square. However, as we do not know how to retype the square (3), we cannot establish a homomorphism $L_G \to L'$. This example motivates the following backward propagation phase, called the clean-up phase.*

The backward clean-up phase

Additionally, we can specify a clean-up phase of propagation to G^-. We specify a mono $r^{\ominus} : L_G^- \leftarrow L_G^{\ominus}$ (and, clearly, to do so, we need to construct L_G^-). The clean-up phase allows us to remove undesired clones that were not specified during the strict phase of rewriting, e.g.

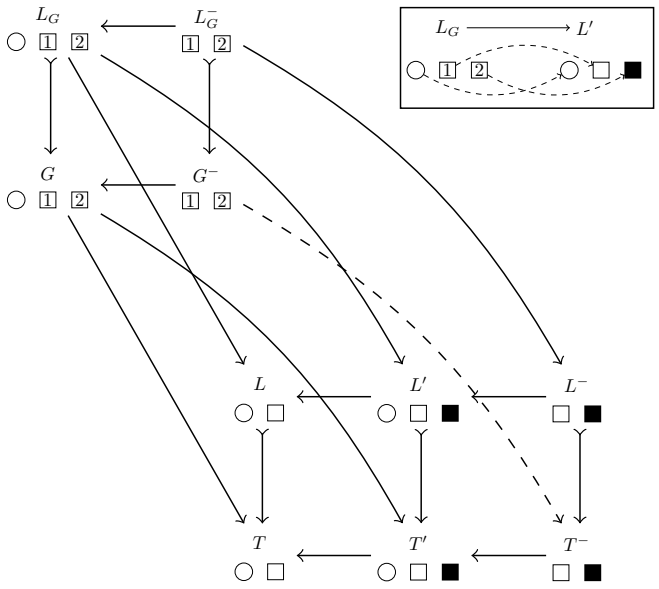


Figure 2.5: Example of backward propagation using rule lifting

a partial concept refinement, where some instances of a cloned element cannot be assigned a unique type in T. The arrow r^- is required to be a mono to avoid creation of additional clones in the clean-up phase (it will also serve us later, in the formal proofs of composability of propagations in Subsection 2.2.6). Now, the application of the clean-up arrow $r^\ominus$ can be achieved by constructing the final pullback complement to $r^\ominus$ and $\hat{m}^-$ as in Diagram 2.38. Alternatively, if we do not construct G^- immediately, but just find the rule lifting $\hat{r}^-$, we can apply the canonical propagation to G together with the clean-up as the final pullback complement to $\hat{r}^- \circ r^\ominus$ and $\hat{m}$ as in Diagram 2.39. The two approaches are equivalent by the horizontal pasting lemma for pullback complements (Lemma A.5.3).

$$\begin{array}{ccccc} L_G^- & \xleftarrow[r^\ominus]{} & L_G^\ominus & & \\ \downarrow{\scriptstyle \hat{m}^-} & & \downarrow{\scriptstyle \hat{m}^\ominus} & & \\ G & \xleftarrow[g^-]{} & G^- & \xleftarrow[g^\ominus]{} & G^\ominus \\ \downarrow{\scriptstyle h'} & & \downarrow{\scriptstyle h^-} & & \\ T' & \xleftarrow[t^-]{} & T^- & & \end{array} \qquad (2.38)$$

$$\begin{array}{ccccc} L_G & \xleftarrow[\hat{r}^-]{} & L_G^- & \xleftarrow[r^\ominus]{} & L_G^\ominus \\ \downarrow{\scriptstyle \hat{m}} & & & & \downarrow{\scriptstyle \hat{m}^\ominus} \\ G & & \xleftarrow[g^- \circ g^\ominus]{} & & G^\ominus \end{array} \qquad (2.39)$$

Example 2.2.3. *Similarly to Example 2.2.2, in Figure 2.7 we have three instances of the square node in G. We now specify the rule factorization that performs both cloning and removal of the circle in the second phase. As the result, cloning is propagated to G in the canonical propagation phase to every instance of the square. We now encode the information on the instance (1) being the white square and (2) being the black square into the clean-up arrow $L_G^\ominus \to L_G^-$.*

Remark 2.2.14. So far, we have discussed how restrictive rewrites can be propagated 'backwards'. But how do such rewrites behave in the case when the updated object is itself typed. For example, consider an arrow $i : T \to U$ and a restrictive rewrite of T with $r : L \leftarrow L^-$

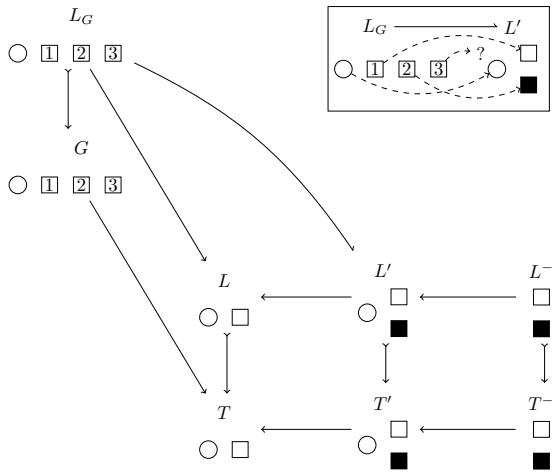


Figure 2.6: Example of failed backward propagation

through $m : L \rightarrowtail T$ and let the object T^- in the following diagram be the result of such rewriting. In the case of restrictive rewrite the result T^- stays typed by U through the composed arrow $i \circ t^-$. Therefore, no propagation to U is required.

$$\begin{array}{ccc} L & \xleftarrow{r} & L^- \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle m^-} \\ T & \xleftarrow{} & T^- \\ & {\scriptstyle t^-}\searrow & \downarrow{\scriptstyle i \circ t^-} \\ & {\scriptstyle i}\searrow & U \end{array} \tag{2.40}$$

2.2.3 Forward propagation

As in the case of backward propagation, we fix two objects G, T and a homomorphism $h : G \to T$. In this subsection we study the application of an expansive rule $r : L \to L^+$ to the object G through a mono $m : L \rightarrowtail G$. The application of the expansive rule corresponds to the pushout from m and r^+ illustrated in the diagram below.

$$\begin{array}{ccc} L & \xrightarrow{r^+} & L^+ \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle m^+} \\ G & \xrightarrow{g} & G^+ \end{array} \tag{2.41}$$

To restore the edge of the hierarchy corresponding to h we would like to propagate a rewrite $G \rightsquigarrow G^+$. Inversely to the application of a restrictive rule to T, the application of an expansive rule to G may require propagation of changes to T, because some of the elements that are added or merged by the rule do not have a proper type in T. The main idea behind this propagation is similar: we decompose a rewrite $G \rightsquigarrow G^+$ into two steps $G \sim(1) \rightsquigarrow G' \sim(2) \rightsquigarrow G^+$. This time, in the first step we specify some arrow $G' \to T$ that would allow us to type some of the new elements introduced by the rule by T, we refer to thus step as the *strict rewrite*. In the second

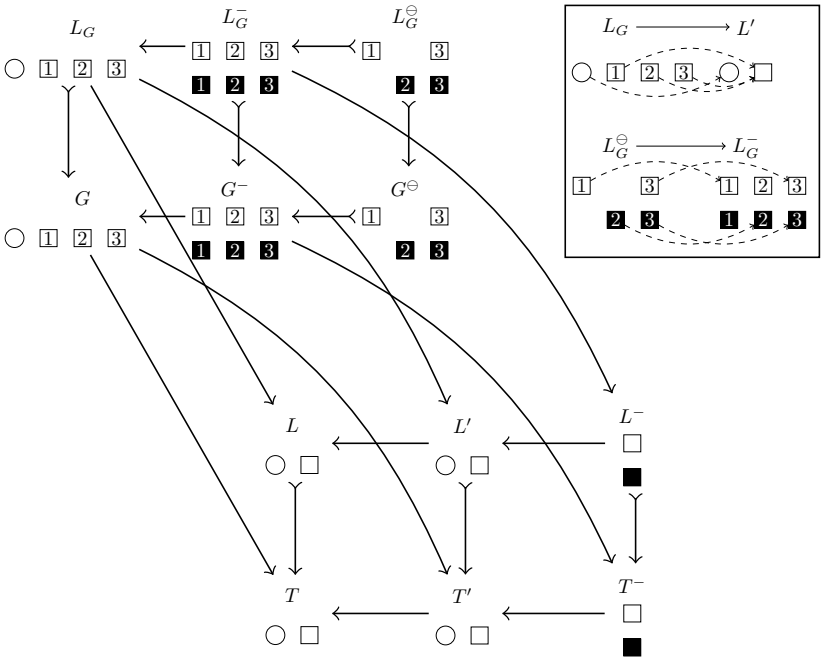


Figure 2.7: Example of backward propagation with the clean-up phase

step we propagate the transformation $G' \rightsquigarrow G^+$, i.e. perform some additions and merges in T to retype G^+, this step is called the *canonical forward propagation*. To decompose our original rewrite $G \rightsquigarrow G^+$ we specify a *forward factorization* of the rule r defined as follows.

Definition 2.2.15. Given a rule $r : L \to L^+$, its *forward factorization* is given by an object L' and arrows $r' : L \to L'$, $r^+ : L' \to L^+$ and $x : L' \to T$ such that $h \circ m = x \circ r'$ and $r = r^+ \circ r'$.

$$\begin{array}{ccc} L & \xrightarrow{r} & L^+ \\ {\scriptstyle h\circ m}\downarrow & \searrow^{r'} & \uparrow{\scriptstyle r^+} \\ T & \xleftarrow[x]{} & L' \end{array} \tag{2.42}$$

The *strict rewrite* of G is defined by constructing the pushout from m and r'. By definition of forward factorization and the universal property of the pushout 2.43, we obtain a unique typing $h' : G' \to T$ that renders Diagram 2.44 commutative. Therefore, we are able to type the intermediary result of rewriting G' by the original object T.

$$\begin{array}{ccc} L & \xrightarrow{r'} & L' \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m'} \\ G & \xrightarrow{g'} & G' \end{array} \tag{2.43}$$

$$\begin{array}{ccc} L & \xrightarrow{r'} & L' \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m'} \\ G & \xrightarrow{g'} & G' \\ & {\scriptstyle h}\searrow & \downarrow{\scriptstyle h'} \\ & & T \end{array} \qquad x : L' \to T \tag{2.44}$$

The *canonical forward rewrite* of G is defined by finding the pushout from m' and r^+ and the corresponding *propagation* to T by constructing the pushout h' and g^+. As the result of such a propagation, we obtain an updated T^+ that types the final result of rewriting G^+.

$$\begin{array}{ccc} L' & \xrightarrow{r^+} & L^+ \\ {\scriptstyle m'}\downarrow & & \downarrow{\scriptstyle m^+} \\ G' & \xrightarrow{g^+} & G^+ \end{array} \qquad (2.45) \qquad\qquad \begin{array}{ccc} G' & \xrightarrow{g^+} & G^+ \\ {\scriptstyle h'}\downarrow & & \downarrow{\scriptstyle h^+} \\ T & \xrightarrow{t^+} & T^+ \end{array} \qquad (2.46)$$

Proposition 2.2.16 shows that splitting a rewrite into the strict and canonical propagation phases produces the same result as direct application of the original rule.

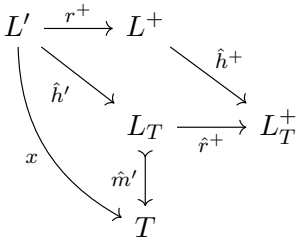
Proposition 2.2.16. Given a rule $r : L \to L^+$ and its forward factorization specified by $L -r'\to L' -r^+\to L^+$ and $x : L' \to T$, an application of r through m can be decomposed into an application of r' through m followed by an application of r^+ through m', corresponding to the pushouts as in the diagram below.

$$\begin{array}{ccccc} L & \xrightarrow{r'} & L' & \xrightarrow{r^+} & L^+ \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle m'} & & \downarrow{\scriptstyle m^+} \\ T & \xrightarrow{t'} & T' & \xrightarrow{t^+} & T^+ \end{array} \qquad (2.47)$$

Proof. The proof follows immediately from the pasting lemma for pushouts (Lemma A.4.2). □

Similarly to the backward propagation, an alternative way to propagate a rewrite specified by r^+ to T consists in constructing a rule that applies directly to T, a *rule projection*. Its construction is presented in the following definition.

Definition 2.2.17. The *projection* $\hat{r}^+ : L_T \to L_T^+$ of r^+ is constructed as the image factorization (defined in Appendix A.7) $L' -\hat{h}'\to L_T \rightarrowtail \hat{m}'\to T$ of the arrow x and the pushout from $\hat{h}'$ and r^+ as in the following diagram.



(2.48)

Theorem 2.2.18. *The graph T^+ and the morphism $t^+ : T \to T^+$ from 2.47 can be obtained by constructing the pushout from $\hat{m}'$ and $\hat{r}^+$ as in the following diagram.*

$$\begin{array}{ccc} L_T & \xrightarrow[\hat{r}^+]{} & L_T^+ \\ {\scriptstyle \hat{m}'}\downarrow & & \downarrow{\scriptstyle \hat{m}^+} \\ T & \xrightarrow[t^+]{} & T^+ \end{array} \qquad (2.49)$$

Proof. See Appendix B. □

If we construct T^+ and t^+ as the pushout from $\hat{m}'$ and $\hat{r}^+$, the arrow $h^+ : G^+ \to T^+$ can be found by the universal property of the pushout that constructed G^+.

Example 2.2.4. *Figure 2.8 illustrates the presented constructions for multi-sets. The rule factorization splits the rewriting of G into two phases: in the first phase the square node is added, in the second phase the black and the white circles are merged. The arrow $L' \to T$ specifies that the square added to G by the rule is actually an instance of the square in T. The strict phase performs the addition, types the intermediary result G' by T. The canonical forward propagation phase merges the circles selected by the rule in G and propagates this merge to their types in T.*

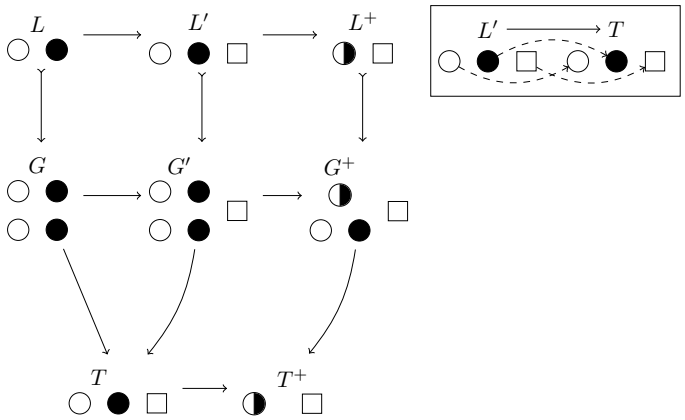


Figure 2.8: Example of forward propagation

Note that addition of new elements to G can be performed in the strict phase, given that all these elements are typed by existing elements in T (using $L' \to T$). However, merges can be performed in the strict phase if and only if they merge elements of the same type.

Alternatively, as in the case of backward propagation, using the same rule factorization we construct the projection of $L' \to L^+$ corresponding to the arrow $L_T \to L_T^+$ in Figure 2.9. This projection corresponds to a rule that can be directly applied to T and that, in this example, merges the white and the black circle in T.

The forward clean-up phase

Additionally, we can specify a clean-up phase of propagation to G^-. We provide a mono $r^\oplus : L_T^+ \twoheadrightarrow L_T^\oplus$. The clean-up phase allows us to further merge some types in T, which is necessary if we want to create multiple instances of the same new type (see Example 2.2.5). The arrow r^+ is required to be an epimorphism (see Definition A.2.3) to avoid addition of new types not required by the original rewriting rule (the fact of $r^\oplus$ being an epi is also used in some composability proofs presented in Subsection 2.2.6). Now, the application of the clean-up arrow $r^\oplus$ can be done by constructing the pushout from $\hat{m}^+$ and $r^\oplus$ as in Diagram 2.50. Note that, by Lemma A.4.4, the arrow $t^\oplus$ is also an epi. Alternatively, if we do not construct T^+ immediately, but first find the rule projection $\hat{r}^+$ instead, we can apply the canonical propagation

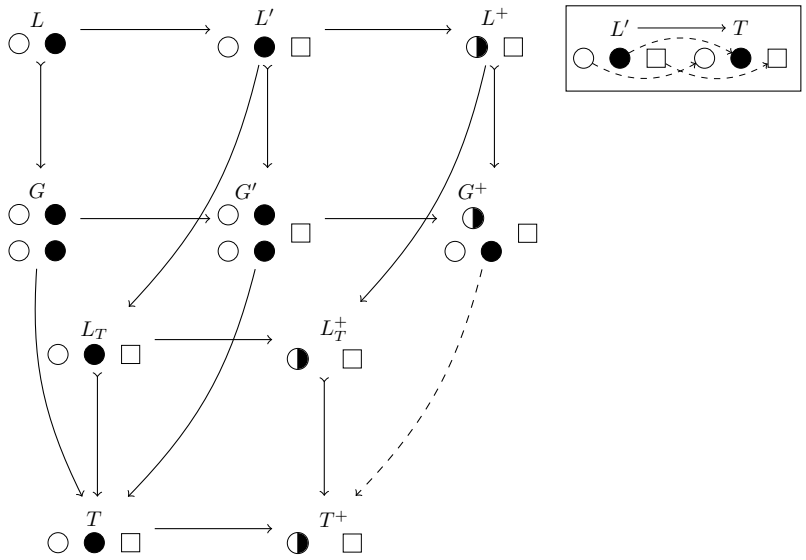


Figure 2.9: Example of forward propagation using rule projection

to T together with the clean-up and the pushout of $\hat{m}'$ and $\hat{r}^{\oplus} \circ \hat{r}^{+}$ as in Diagram 2.51. The two approaches are equivalent by the pasting lemma for pushouts (Lemma A.4.2).

$$\begin{array}{ccc} L_T^+ & \xrightarrow{r^{\oplus}} & L_T^{\oplus} \\ {\scriptstyle \hat{m}^+}\downarrow & & \downarrow{\scriptstyle \hat{m}^{\oplus}} \\ T^+ & \xrightarrow{t^{\oplus}} & T^{\oplus} \end{array} \qquad (2.50) \qquad \begin{array}{ccccc} L_T & \xrightarrow{\hat{r}^+} & L_T^+ & \xrightarrow{r^{\oplus}} & L_T^{\oplus} \\ {\scriptstyle \hat{m}'}\downarrow & & & & \downarrow{\scriptstyle \hat{m}^{\oplus}} \\ T & & \xrightarrow{t^{\oplus} \circ t^{+}} & & T^{\oplus} \end{array} \qquad (2.51)$$

Example 2.2.5. *In this example we merge the circle nodes in G and add two black squares as in Figure 2.10. Addition of the squares is performed in the second phase, because we want to create new types for these elements in T. As the result of the canonical propagation the two black squares are added to T. However, we would like the two squares from G^+ to be instances of the same type in T, therefore we merge their corresponding typing elements in the clean-up phase specified by $L_T^+ \twoheadrightarrow L_T^{\oplus}$.*

2.2.4 Controlling propagation in SimpGrph

Previously in this section we have discussed the ways to specify propagations using rule factorizations and clean-up arrows. However, for our practical applications, e.g. implementation of hierarchies of simple graphs in `ReGraph`, we would like to design a more concise way for specifying strategies of propagation. More precisely, we would like to provide a set of techniques that enables the user to *control* backward and propagation without constructing multiple complex objects and homomorphisms defining rule factorizations and clean-up arrows. In the rest of this section we focus on such techniques for the propagation of rewriting in the category **SimpGrph**. They apply to $\mathbf{SimpGrph}_{attrs}$ and can be easily extended to **Grph** and $\mathbf{Grph}_{attrs}$.

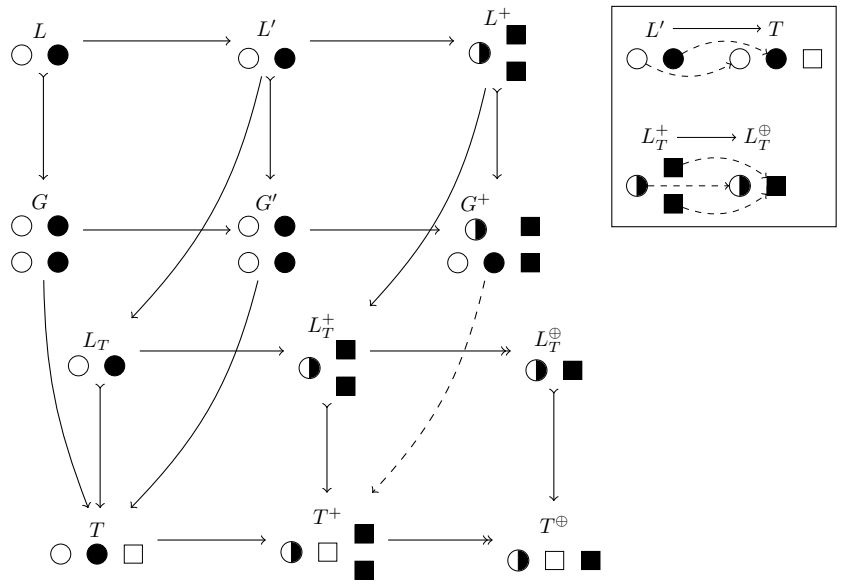
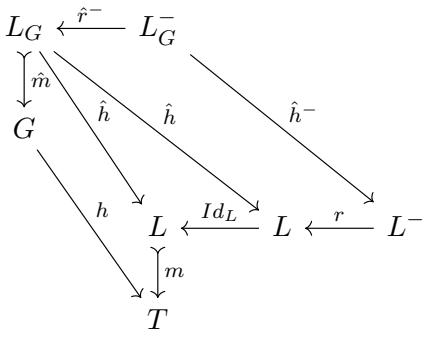


Figure 2.10: Example of forward propagation with the clean-up phase

Controlling backward propagation

Recall that the application of a restrictive rule to some simple graph T can be interpreted in terms of node/edge deletions and node clones. Consider a graph G and a homomorphism $h : G \to T$ as before. For any backward propagation along h, deletion of some components from T will always result in deletion of the respective components from G (of the nodes typed by the removed nodes or edges implicitly typed by the removed edges, due to the edge-preservation property of our homomorphisms). Meanwhile, node cloning leaves us some freedom of choice. We have previously seen the interplay between different backward rule factorizations and clones produced as the result of propagation. Moreover, we were able to remove some of the produced clones using the clean-up phase.

For a given rule $r : L \leftarrow L^-$ and a match $m : L \rightarrowtail T$, instead of explicitly specifying a rule factorization and a clean-up arrow, we can specify a binary relation $R \subseteq V_G \times V_{L^-}$, which allows us to indicate which nodes from G should 'correspond' to which nodes from L^- in the propagation. To perform the propagation controlled by R we will construct the rule factorization $L', r' : L \leftarrow L', r^+ : L' \leftarrow L^-$ and $\hat{h}' : L_G \to L'$ with $L' = L$, $r' = Id_L$, $r^+ = r$ and $\hat{h}' = \hat{h}$. We will also construct the corresponding rule lifting $\hat{r}^-$ by finding the pullback from $\hat{h}$ and r as in the diagram below. Application of the lifting $\hat{r}^-$ through $\hat{m}$ results in the canonical backward propagation and performs cloning of all the instances of cloned nodes in G.



(2.52)

Now, we use R to compute an object $L_G^{\ominus}$ and a clean-up arrow $r^{\ominus} : L_G^- \leftarrowtail L_G^{\ominus}$ that removes unnecessary clones according to R. Let us denote with π_G and π_{L^-} the projections of elements in R to V_G and V_{L^-} respectively. The set of nodes of $L_G^{\ominus}$ is then defined as

$$V_{L_G^{\ominus}} = \{v \in V_{L_G^-} \mid (\hat{m} \circ \hat{r}^-(v) \notin \pi_G) \text{ or } ((\hat{m} \circ \hat{r}^-(v), \hat{h}^-(v)) \in R)\}.$$

The homomorphism $r^{\ominus}$ is defined as $r^{\ominus}(v) = v$ for all $v \in V_{L_G^{\ominus}}$. It is not hard to verify that, by construction, $r^{\ominus}$ is a mono. Finally, the set of edges of $L_G^{\ominus}$ is defined as

$$E_{L_G^{\ominus}} = \{(u, v) \in E_{L_G^-} \mid u \in V_{L_G^{\ominus}} \text{ and } v \in V_{L_G^{\ominus}}\}.$$

Example 2.2.6. *Let us consider a small example of backward propagation controlled by a relation $R \subseteq V_G \times V_{L^-}$ in Figure 2.11. The original rule specifies two clone operations, one for the circle and one for the square node in T. We construct a factorization in which $L' \to L$ is the identity and find the rule lifting $L_G \leftarrow L_G^-$. The application of this rule clones all the instances of the circle and the square nodes. Our relation R specifies the following correspondences between the nodes of G and L^-: the black circle corresponds to the left semicircle, the white circle corresponds to the right semicircle and the black square corresponds to the upper semisquare in the figure. We construct the clean-up arrow that removes from G^- unnecessary clones. Note that R does not specify any correspondence to the white circle from G (i.e. the white node is not in π_G), therefore both clones of the white square are kept in the propagation (i.e. we propagate cloning to the white square canonically).*

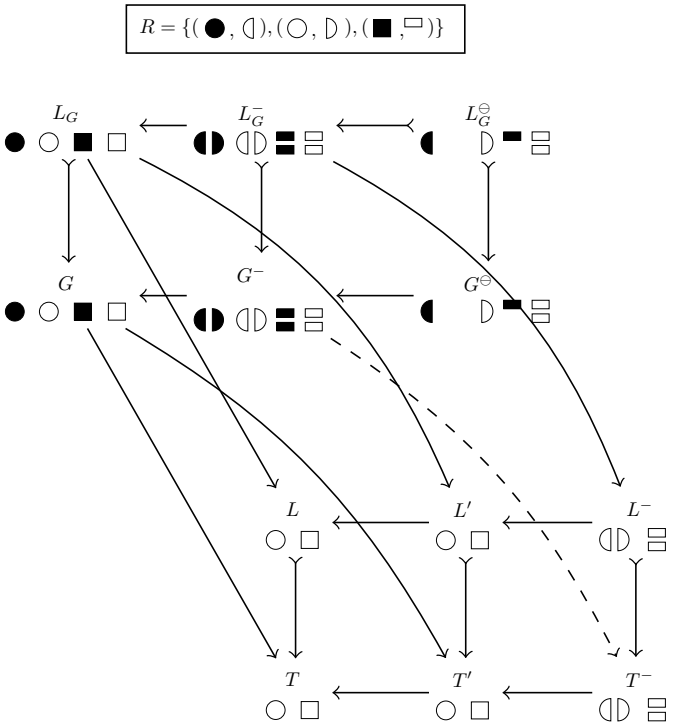


Figure 2.11: Example of controlled backward propagation

Controlling forward propagation

As before, consider two graphs G, T and $h : G \to T$. The application of an expansive rule performs node/edge addition and node merge. To design a way for controlling forward propagations

we need to analyse the effect of these operations on h. For any forward propagation a merge of two nodes from G typed by different elements in T results in a merge of these two elements in T. Similarly addition of new edges to G^+ results in addition of corresponding edges to T^+ (due to the edge-preservation property of homomorphisms). Meanwhile, addition of nodes leaves us some degree of freedom. We have previously discussed the ways to use rule factorizations and clean-up arrows to control the addition of new types. Similarly to backward propagation, here we provide a more concise way to control forward propagations.

For a given rule $r : L \to L^+$ and a match $m : L \rightarrowtail G$, instead of explicitly specifying a rule factorization, we can specify a *functional* relation $R \subseteq V_{L^+} \times V_T$. Recall that a binary relation $R \subseteq X \times Y$ is functional, if for $x \in X$ and $y_1, y_2 \in Y$, $(x, y_1) \in R$ and $(x, y_2) \in R$ implies that $y_1 = y_2$ (functional relations are also referred to as partial maps). We use this relation to partially type the newly added nodes in L^+. For every element $v \in V_{L^+}$ we write $R(v) = w$ if $(v, w) \in R$, in this case we say v is typed in w in T.

Unlike in the case of backward propagation, we use R to construct a forward rule factorization (and not a clean-up arrow). Namely, we construct L', $r' : L \to L'$, $r^+ : L' \to L^+$ and $x : L' \to T$ as in Definition 2.2.15. Let us denote with π_{L^+} and π_T the projections of R to V_{L^+} and V_T respectively. The set of nodes and the set of edges of L' are defined as follows

$$V_{L'} = \{v \in V_L\} \uplus \{v \in V_{L_+} \mid v \notin im(r^+) \text{ and } v \in \pi_{L^+}\}$$

$$E_{L'} = \{(u, v) \in E_L\}$$

Then, for all $v \in V_L$, $r'(v) = v$, while r^+ and x are defined as follows.

$$r^+(v) = \begin{cases} r(v), & \text{for } v \in V_L \\ v, & \text{for } v \in V_{L^+} \text{ such that } v \notin im(r^+) \text{ and } v \in \pi_{L^+} \end{cases}$$

$$x(v) = \begin{cases} h \circ m(v), & \text{for } v \in V_L \\ R(v), & \text{for } v \in V_{L^+} \text{ such that } v \notin im(r^+) \text{ and } v \in \pi_{L^+} \end{cases}$$

Note that, because we do not construct a clean-up arrow, we cannot express the entire range of possible forward propagation strategies, namely, we cannot encode addition of multiple instances of the same *new* type in T. This is due to the fact that R operates on V_T, i.e. the set of already existing elements of T.

Example 2.2.7. *Consider a small example of forward propagation controlled by a relation $R \subseteq V_{L^+} \times V_T$ in Figure 2.12. The original rule specifies a merge of two circular nodes and an addition of two nodes, the square and the triangle. The relation specifies that the new square node in L^+ is typed by the already existing square node in T. We construct the rule factorization encoding this information and propagate the addition of the triangle canonically.*

2.2.5 Composability of propagation

We will now extend our theory of backward and forward propagation to triples of objects and homomorphisms forming *undirected cycles*. We will study how the existence of a unique homomorophism between the results of propagations can be guaranteed in the two scenarios:

1. for three objects G_1, G_2, T and homomorphisms as in Diagram 2.53, given a restrictive rewrite of T and two backward propagations to G_1 and G_2;

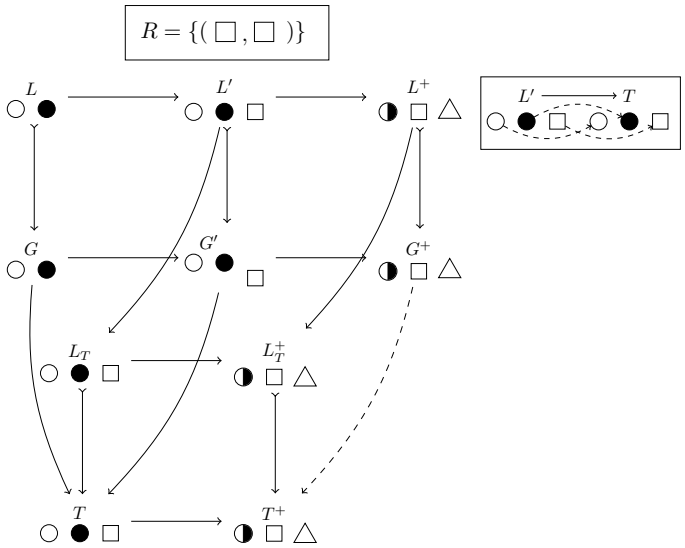
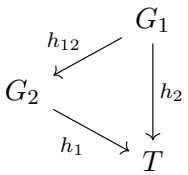
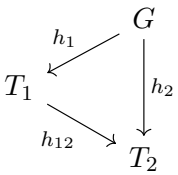


Figure 2.12: Example of controlled backward propagation

2. for objects G, T_1, T_2 and homomorphisms as in Diagram 2.54, given an expansive rewrite of G and two forward propagations to T_1 and T_2.



(2.53)

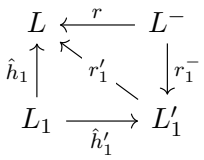


(2.54)

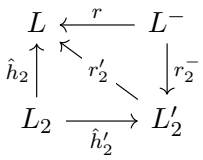
In this section, we give the conditions, called *composability conditions*, under which the above-mentioned homomorphism exists and is unique for the updated objects.

Backward composability

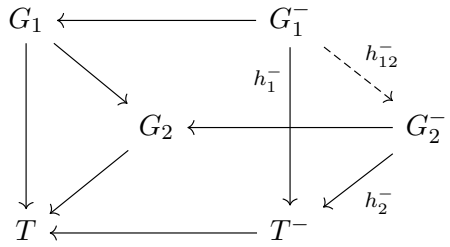
Definition 2.2.19. Given a restrictive rule $r : L \leftarrow L^-$, a match $m : L \rightarrowtail T$ and two rule factorizations as in Diagrams 2.55 and 2.56, defining a propagation of r to G_1 and G_2 respectively (where L_i is shorthand for L_{G_i} as defined by the pullback in 2.28), the two propagations $h_1^- : G_1^- \to T^-$ and $h_2^- : G_2^- \to T^-$ are *composable* if and only if there exists a *unique* $h_{12}^- : G_1^- \to G_2^-$ that renders Diagram 2.57 commutative.



(2.55)



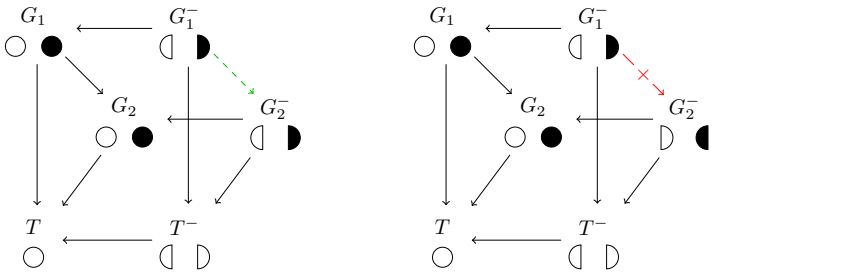
(2.56)



(2.57)

Remark 2.2.20. Note that we can interpret the requirement of the arrow h_{12}^- in the previous definition to be unique as the requirement for it to be unambiguously determined by the two rule factorizations. If we think of it in terms of implementation (like it is done in `ReGraph`) it means that we can compute such an arrow automatically.

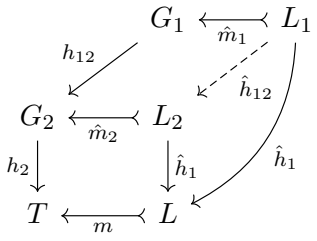
Example 2.2.8. *Figure 2.13 illustrates a pair of composable and a pair of not composable propagations. The circle node in T is split into two semi-circles (the left semi-circle and the right semi-circle). In Subfigure (a) the white circles in G_1 and G_2 are retyped by the left semi-circle, while the black ones are retyped by the right semi-circle. In Subfigure (b) the white circle from G_1 is retyped with the left semi-circle while the white circle from G_2 is retyped with the right semi-circle (similarly for the black circles). In this case we cannot construct a homomorphism $G_1^- \to G_2^-$ that renders the diagram in (b) commutative.*



(a) Composable propagations *(b) Not composable propagations*

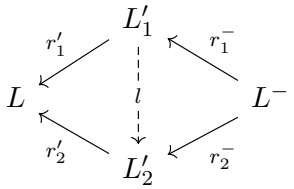
Figure 2.13: Example of composability of backward propagations

First of all, we can easily show that the subgraphs L_1 and L_2 of G_1 and G_2 affected by rewriting of T are homomorphic. This can be done by using the universal property of the pullback and showing that there exists a unique homomorphism $\hat{h}_{12} : L_1 \to L_2$ that renders the following diagram commutative. Moreover, by Lemma A.3.3, the square formed by $h_{12} \circ \hat{m}_1$ and $\hat{m}_2 \circ \hat{h}_{12}$ is a pullback.



(2.58)

Theorem 2.2.21. *Given the backward rule factorizations as in Diagrams 2.55 and 2.56 and a homomorphism $l : L_1' \to L_2'$ rendering Diagrams 2.59 and 2.60 commutative, there exists a unique homomorphism $h_{12}^- : G_1^- \to G_2^-$ rendering Diagram 2.57 commutative. In other words, given such an l, the two propagations are composable.*

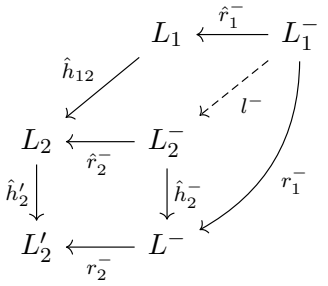


(2.59)

$$\begin{array}{ccc} L'_1 & \xleftarrow{\hat{h}'_1} & L_1 \\ {\scriptstyle l}\downarrow & & \downarrow{\scriptstyle \hat{h}_{12}} \\ L'_2 & \xleftarrow[\hat{h}'_2]{} & L_2 \end{array} \tag{2.60}$$

Proof. See Appendix B. □

Theorem 2.2.21 states the conditions under which backward rule factorizations produce composable propagations. We now would like to state similar conditions on the arrows specifying the further clean-up. Consider the analogous setting: a hierarchy shaped as Diagram 2.53, rewriting with a restrictive rule $r : L \leftarrow L^-$ and two rule factorizations from Diagrams 2.55 and 2.56 that, moreover, satisfy the composability conditions stated in our previous theorem. Let $\hat{r}_1^- : L_1 \leftarrow L_1^-$ and $\hat{r}_2^- : L_2 \leftarrow L_2^-$ be two liftings of r_1^- and r_2^- respectively. Using the fact that $r_2^- = l \circ r_1^-$, we can apply the universal property of the pullback that constructed L_2^- and show that there exists a unique arrow $l^- : L_1^- \to L_2^-$ that renders Diagram 2.61 commutative. Moreover, by Lemma A.3.3 the square formed by $\hat{h}_{12} \circ \hat{r}_1^-$ and $\hat{r}_2^- \circ l^-$ is a pullback.



(2.61)

Let $r_1^\ominus : L_1^- \rightarrowtail L_1^\ominus$ and $r_2^\ominus : L_2^- \rightarrowtail L_2^\ominus$ be two clean-up arrows and $G_1^\ominus$ and $G_2^\ominus$ be the results of the respective clean-up phase given by the final pullback complements in the following diagrams. Note that, by Lemma A.5.2, arrows $g_1^\ominus$ and $g_2^\ominus$ in the diagram are also monos.

$$\begin{array}{ccc} L_1^- & \xrightarrow[r_1^\ominus]{} & L_1^\ominus \\ {\scriptstyle \hat{m}_1^-}\downarrow & & \downarrow{\scriptstyle \hat{m}_1^\ominus} \\ G_1^- & \xrightarrow[g_1^\ominus]{} & G_1^\ominus \end{array} \tag{2.62}$$

$$\begin{array}{ccc} L_2^- & \xrightarrow[r_2^\ominus]{} & L_2^\ominus \\ {\scriptstyle \hat{m}_2^-}\downarrow & & \downarrow{\scriptstyle \hat{m}_2^\ominus} \\ G_2^- & \xrightarrow[g_2^\ominus]{} & G_2^\ominus \end{array} \tag{2.63}$$

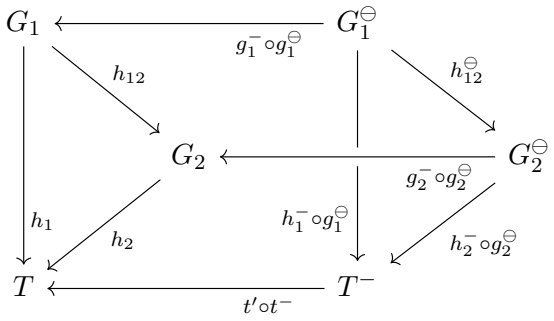
Theorem 2.2.22. *Given backward rule factorizations as in Diagrams 2.55 and 2.56 satifying the composability conditions from Theorem 2.2.21, two clean-up arrows $r_1^\ominus : L_1 \rightarrowtail L_1^\ominus$ and $r_2^\ominus : L_2 \rightarrowtail L_2^\ominus$ and a homomorphism $l^\ominus : L_1^\ominus \to L_2^\ominus$ rendering Diagram 2.64 commutative, there exists a unique $h_{12}^\ominus : G_1^\ominus \to G_2^\ominus$ that renders Diagram 2.65 commutative.*

$$\begin{array}{ccc} L_1^- & \xrightarrow[r_1^\ominus]{} & L_1^\ominus \\ {\scriptstyle l^-}\downarrow & & \vdots{\scriptstyle l^\ominus} \\ L_2^- & \xrightarrow[r_1^\ominus]{} & L_2^\ominus \end{array} \tag{2.64}$$

$$\begin{array}{ccc} G_1^- & \xleftarrow[g_1^\ominus]{} & G_1^\ominus \\ \downarrow{\scriptstyle h_{12}} & & \vdots{\scriptstyle h_{12}^\ominus} \\ G_2^- & \xleftarrow[g_1^\ominus]{} & G_2^\ominus \end{array} \tag{2.65}$$

Proof. See Appendix B. □

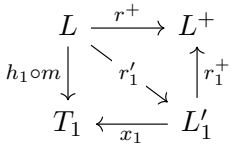
After obtaining the unique $h^{\ominus}_{12}$ that satisfies $g^{\ominus}_2 \circ h^{\ominus}_{12} = h^{-}_{12} \circ g^{\ominus}_1$, we can now verify that $h^{\ominus}_{12}$ also satisfies $g^{-}_2 \circ g^{\ominus}_2 \circ h^{\ominus}_{12} = h_{12} \circ g^{-}_1 \circ g^{\ominus}_1$ and $h^{-}_2 \circ g^{\ominus}_2 \circ h^{\ominus}_{12} = h^{-}_1 \circ g^{\ominus}_1$, i.e. all the squares and triangles in the following diagram commute, and we can consistently compose the results of the two clean-up phases.



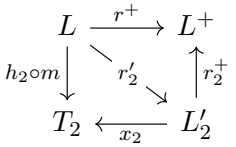
(2.66)

Forward composability

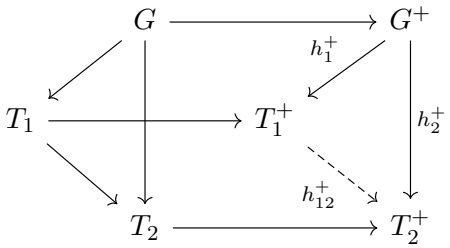
Definition 2.2.23. Given an expansive rule $r : L \to L^+$, a match $m : L \rightarrowtail G$ and two rule factorizations as in Diagrams 2.70 and 2.71, defining the propagation of r to T_1 and T_2 respectively, the two propagations $h^+_1 : G^+ \to T^+_1$ and $h^+_2 : G^+ \to T^+_2$ are *composable* if and only if there exists a *unique* $h^+_{12} : T^+_1 \to T^+_2$ that renders Diagram 2.69 commutative.



(2.67)



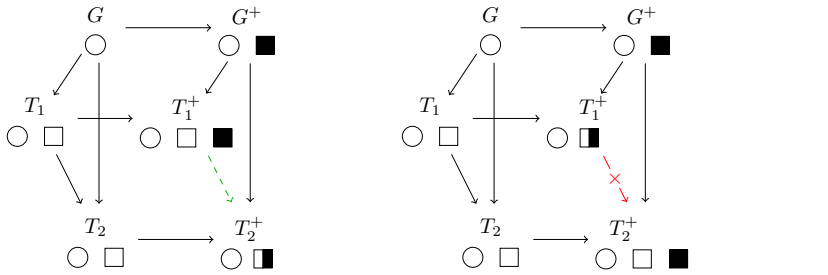
(2.68)



(2.69)

Remark 2.2.24. Note that, similarly to backward propagation, we can interpret the requirement of the arrow h^+_{12} in the previous definition to be unique as the requirement for it to be unambiguously determined by the two rule factorizations. It means that we can compute such an arrow automatically.

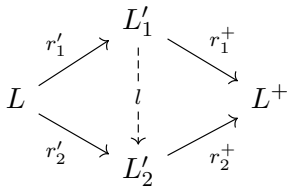
Example 2.2.9. *Figure 2.14 illustrates a pair of composable and a pair of not composable propagations. In Subfigure (a) the black square is added to G by a rule, the canonical propagation to T_1 is performed, which produces a new type corresponding to the black square in T_1^+. At the same time, the new black square in G^+ is retyped by the existing square node in T_2, therefore no canonical propagation is performed (we fall in the strict rewriting case with respect to T_2). There exists a unique homomorphism that maps two squares in T_1^+ to the square node in T_2^+ (denoted with split square). In Subfigure (a) the same rule is applied to G. Now, the newly added square is retyped by the existing square in T_1, and we fall in the strict rewriting phase. On the other hand, the addition of the square is canonically propagated to T_2^+. In this case, we cannot construct a homomorphism $T_1^+ \to T_2^+$ that renders the diagram in (b) commutative.*



(a) Composable propagations *(b) Not composable propagations*

Figure 2.14: Example of composability of forward propagations

Theorem 2.2.25. *Given forward rule factorizations as in Diagrams 2.70 and 2.71 and a homomorphism $l : L_1' \to L_2'$ rendering Diagrams 2.70 and 2.71 commutative, there exists a unique homomorphism $h_{12}^+ : T_1^+ \to T_2^+$ rendering diagram 2.69 commutative. In other words, given such an l, the two propagations are composable.*

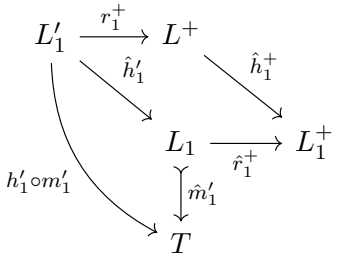


(2.70)

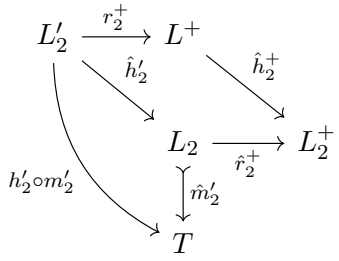
$$\begin{array}{ccc} L_1' & \xrightarrow{x_1} & T_1 \\ {\scriptstyle l}\downarrow & & \downarrow{\scriptstyle h_{12}} \\ L_2' & \xrightarrow[x_2]{} & T_2 \end{array} \qquad (2.71)$$

Proof. See Appendix B. □

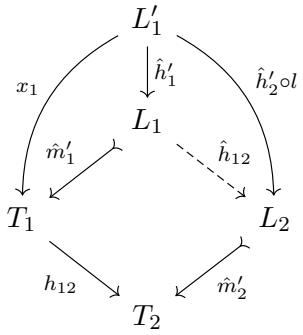
Theorem 2.2.25 states the conditions under which forward rule factorizations produce composable propagations. We now would like to state similar conditions on the arrows specifying the further clean-up. Consider the analogous setting: a hierarchy shaped as Diagram 2.54, rewriting with an expansive rule $r : L \to L^+$ and two rule factorizations from Diagrams 2.70 and 2.71 that, moreover, satisfy the composability conditions stated in our previous theorem. Let $\hat{r}_1^+ : L_1 \to L_1^+$ and $\hat{r}_2^+ : L_2 \to L_2^+$ be two projections of r_1^+ and r_2^+ respectively as in Diagrams 2.72 and 2.73 (where L_i is shorthand for L_{T_i}). Recall that L_1 and L_2 are obtained as image factorizations of $h_1' \circ m_1'$ and $h_2' \circ m_2'$. We use this fact to apply Lemma A.7.4 and show that there exists a unique arrow $\hat{h}_{12} : L_1 \to L_2$ that renders Diagram 2.74 commutative. This further allows us to state that the outer square in Diagram 2.75 commutes, use the universal property of the pushout that gives L_1^+ and show that there exists a unique $l^+ : L_1^+ \to L_2^+$ that renders this diagram commutative. This means that the right-hand sides of the two rule propagations are homomorphic.



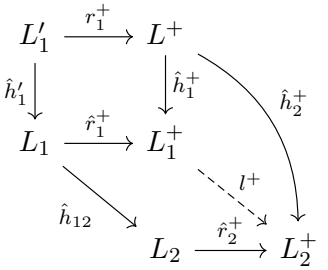
(2.72)



(2.73)

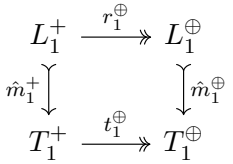


(2.74)

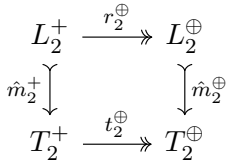


(2.75)

Let $r_1^\oplus : L_1^+ \twoheadrightarrow L_1^\oplus$ and $r_2^\oplus : L_2^+ \twoheadrightarrow L_2^\oplus$ be two clean-up arrows and $T_1^\oplus$ and $T_2^\oplus$ be the results of the respective clean-up phases given by the pushouts in the following diagrams. Note that arrows $t_1^\oplus$ and $t_2^\oplus$ in this diagram are also epis (by Lemma A.4.4).

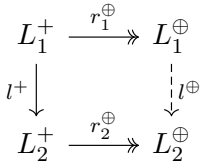


(2.76)

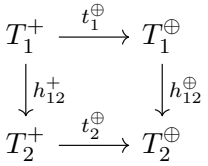


(2.77)

Theorem 2.2.26. *Given the forward rule factorizations as in Diagrams 2.70 and 2.71 satifying composability conditions from Theorem 2.2.25, two clean-up arrows $r_1^\oplus : L_1^+ \twoheadrightarrow L_1^\oplus$ and $r_2^\oplus : L_2^+ \twoheadrightarrow L_2^\oplus$ and a homomorphism $l^\oplus : L_1^\oplus \to L_2^\oplus$ rendering Diagram 2.78 commutative, there exists a unique $h_{12}^\oplus : T_1^\oplus \to T_1^\oplus$ that renders Diagram 2.79 commutative.*



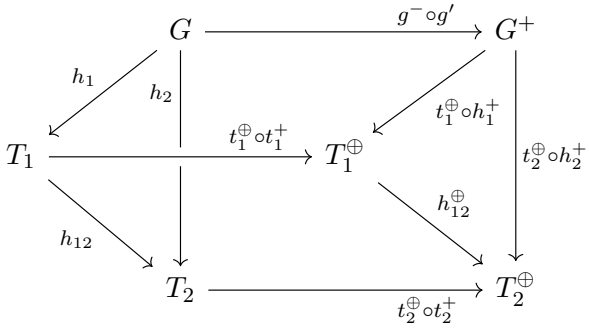
(2.78)



(2.79)

Proof. See Appendix B. □

Having obtained such a unique $h^{\oplus}_{12}$ that satisfies $h^{\oplus}_{12} \circ t^{\oplus}_1 = t^{\oplus}_2 \circ h^{+}_{12}$, it is not hard to verify that $h^{\oplus}_{12}$ also satisfies $t^{\oplus}_2 \circ h^{+}_2 = h^{\oplus}_{12} \circ t^{\oplus}_1 \circ h^{+}_1$ and $t^{\oplus}_2 \circ t^{+}_2 \circ h_{12} = h^{\oplus}_{12} \circ t^{\oplus}_1 \circ t^{+}_1$, i.e. all the squares and triangles in Diagram 2.80 commute and we can consistently compose the results of the two clean-up phases.



(2.80)

2.2.6 Hierarchy rewriting

Earlier in this section we have discussed the notions of backward and forward propagation, and the conditions under which these propagations can be composed. In this subsection we would like to describe how these notions can be used to perform the update of objects and homomorphisms in a hierarchy induced by the application of a general SqPO rewriting rule to one of its objects. Here, instead of considering pairs or triples of objects, we will consider an arbitrary hierarchy of objects $\mathcal{H}$ given by a tuple $(V, E, \mathcal{O}, \mathcal{F}, \alpha, \beta)$. We are interested in applying a general SqPO rewriting rule $r : L \leftarrow r^{+} - P - r^{-} \rightarrow R$ (containing both the restrictive and the expansive phases) to an object situated at a vertex $v \in V$ of the hierarchy, called the *origin of rewriting*, through a matching $m : L \rightarrowtail \alpha(v)$. Then, given specifications for forward and backward propagation of this rewrite (e.g. rule factorizations and clean-up arrows) for all the ancestors and descendants of v, we would like to construct an updated hierarchy $\mathcal{H}' = (V, E, \mathcal{O}', \mathcal{F}', \alpha', \beta')$, whose structure is identical to H, but whose underlying objects and homomorphisms correspond to the result of the specified rewriting and propagation. Note that in this subsection we assume that the specifications for forward and backward propagation produce pairwise composable propagations, i.e. locally satisfy the composability conditions stated in Subsection 2.2.5.

Updating hierarchy objects

Given a rule $r : L \leftarrow r^{+} - P - r^{-} \rightarrow R$ and a matching $m : L \rightarrowtail \alpha(v)$, the update of the origin simply corresponds to the application of the rule r through the matching m to $\alpha(v)$. Then, for every ancestor node $a \in anc(v)$, we proceed by finding a path π from a to the origin v and by computing the homomorphism $\circ\pi : \alpha(a) \rightarrow \alpha(v)$. This homomorphism allows us to use the previously defined framework for backward propagation of changes specified by r^{-} to the ancestor a. We proceed in a similar way for all the descendants of v, i.e. for every $d \in desc(v)$, we compute the path π from $\alpha(v)^{-}$, the origin rewritten using r^{-}, to d and find the homomorphism $\circ\pi : \alpha(v)^{-} \rightarrow \alpha(d)$. We then are able to apply the framework for forward propagation of r^{+} to d. The rest of the objects in a hierarchy, i.e. the ones that do not correspond to neither ancestors or descendants of the origin, stay unchanged.

Updating hierarchy homomorphisms

First of all, the homomorphisms corresponding to the edges incident to the origin of rewriting can be obtained immediately, as the result of backward and forward propagation. The unique homomorphisms between objects corresponding to pairs of ancestors (or descendants) are given by composability stated in Subsection 2.2.5.

Now, to understand how the homomorphisms corresponding to the edges from ancestors to descendants of the origin can be obtained, let us consider the following scenario. Let $r : L \leftarrow r^- - P - r^+ \rightarrow R$ be a general SqPO rule (containing both restrictive and expansive updates) applied to the graph G through a matching $m : L \rightarrowtail G$, as in Diagram 2.81. If the graph G is situated in a hierarchy shaped according to Diagram 2.82, we need to combine backward and forward propagations for r^- and r^+ respectively to find H^- and T^+. We then need to reconstruct the edges of the hierarchy with homomorphisms h_1^+, h_2^+ and h_3' as in Diagram 2.83 (where G^+ stands for the result of original rewriting) in a way that makes this diagram commute.

$$\begin{array}{ccccc} L & \xleftarrow{r^-} & P & \xrightarrow{r^+} & R \\ \downarrow{\scriptstyle m} & & \downarrow{\scriptstyle m^-} & & \downarrow{\scriptstyle m^+} \\ G & \xleftarrow[g^-]{} & G^- & \xrightarrow{g^+} & G^+ \end{array} \tag{2.81}$$

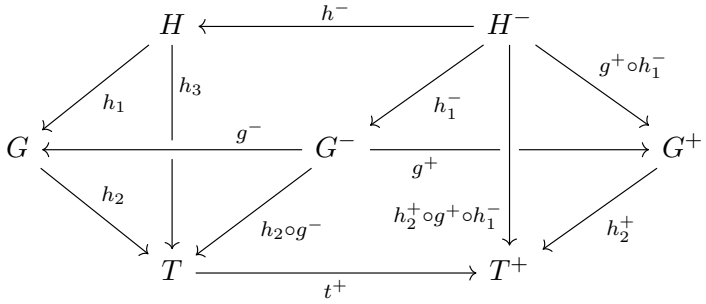
$$\begin{array}{ccc} & H & \\ {\scriptstyle h_1}\swarrow & & \downarrow{\scriptstyle h_3} \\ G & & T \\ {\scriptstyle h_2}\searrow & & \end{array} \tag{2.82}$$

$$\begin{array}{ccc} H^- & \xrightarrow{h_1^+} & \\ {\scriptstyle h_3'}\downarrow & & G^+ \\ T^+ & \xleftarrow{h_2^+} & \end{array} \tag{2.83}$$

First of all, we perform a backward propagation of r^- to H as in Diagram 2.84. Then, we can perform a forward propagation of r^+ to T using the composed arrow $h_2 \circ g^-$ as in Diagram 2.85. We can then reconstruct the homomorphism $h_1^+ : H^- \rightarrow G^+$ as the composition $g^+ \circ h_1^-$ and, finally, the arrow $H^- \rightarrow T^+$ as simply the composition $h_2^+ \circ g^+ \circ h_1^-$ as in Diagram 2.86.

$$\begin{array}{ccc} H & \xleftarrow{h^-} & H^- \\ {\scriptstyle h_1}\downarrow & & \downarrow{\scriptstyle h_1^-} \\ G & \xleftarrow[g^-]{} & G^- \end{array} \tag{2.84}$$

$$\begin{array}{ccc} G^- & \xrightarrow{g^+} & G^+ \\ {\scriptstyle h_2 \circ g^-}\downarrow & & \downarrow{\scriptstyle h_2^+} \\ T & \xrightarrow{t^+} & T^+ \end{array} \tag{2.85}$$

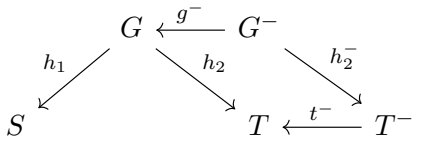


(2.86)

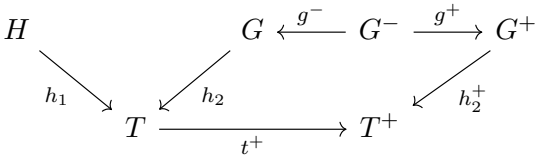
Previously, we have stated that only the objects corresponding to the origin node, its ancestors and descendants are updated, while the rest of the objects stay unchanged. To be able to update the homomorphisms associated to the in-/outgoing edges from the nodes whose corresponding objects are unchanged, let us study the following two scenarios.

Consider a hierarchy consisting of three objects G, S, T and two homomorphisms $h_1 : G \to S$ and $h_2 : G \to T$ (as Diagram 2.87). Let T correspond to the origin of rewriting (with an arbitrary SqPO rule) and let $g^- : G \leftarrow G^-$ be the homomorphism obtained as the result of backward propagation of this rewriting, i.e. its restrictive phase producing $t^- : T \leftarrow T^-$. The node associated with S is neither an ancestor nor a descendant of the origin, therefore S stays unchanged. However, we still need to update the hierarchy edge that was going from the updated G to the unchanged S, which can be done simply by associating this edge with the homomorphism $h_1 \circ g^-$.

On the other hand, let us consider a hierarchy consisting of three objects G, H, T and two homomorphisms $h_1 : H \to T$ and $h_2 : G \to T$ (as Diagram 2.88). Let G correspond to the origin of rewriting (with an arbitrary SqPO rule) and the arrows $g^- : G \to G^-$ and $g^+ : G^- \to G^+$ be the results of the restrictive and expansive rewriting phases respectively. Let $t^+ : T \to T^+$ be the homomorphism obtained as the result of forward propagation of this rewriting, i.e. its expansive phase producing $g^+ : G^- \to G^+$. As before, the node associated with H is neither an ancestor nor a descendant of the origin, therefore H stays unchanged. However, we still need to update the hierarchy edge that was going from the unchanged H to the updated T, which can be done simply by associating this edge with the homomorphism $t^+ \circ h_1$.



(2.87)

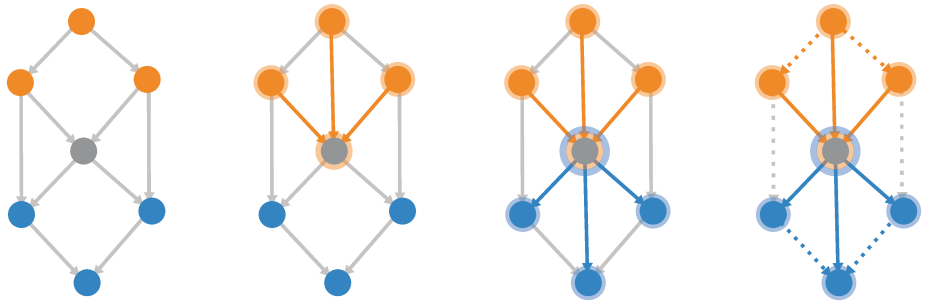


(2.88)

Algorithm 1 presented in Appendix C provides a concrete algorithmic procedure that, given a hierarchy $\mathcal{H} = (V, E, \mathcal{O}, \mathcal{F}, \alpha, \beta)$, a rewrite of an object at some vertex $v \in V$ corresponding to a rule r and a match m; and a set of rule factorizations for all the ancestors and descendants of v, outputs an updated hierarchy $\mathcal{H}' = (V, E, \mathcal{O}', \mathcal{F}', \alpha', \beta')$. This procedure first computes the homomorphisms corresponding to the paths to all the ancestors and descendants of the origin of rewriting, propagates changes according to these homomorphisms and reconstructs the hierarchy arrows between pairs of ancestors or descendants using subroutines, whose correctness is guaranteed by the composability theorems (see a schematic example in Figure 2.15).

2.2.7 Rule hierarchies

Recall that, given two graphs G, T and a homomorphism $h : G \to T$, we have described two scenarios of: (1) backward propagation to G induced by a restrictive rewrite of T (Subsection 2.2.2), and (2) forward propagation to T induced by an expansive rewrite of G (Subsection 2.2.3). In Subsection 2.2.6 we have also discussed how an arbitrary SqPO rule defining both restrictive and expansive updates can be applied in a general hierarchy combining the techniques of backward and forward propagation. In this subsection we would like define how, given a general hierarchy and a rewrite of an individual object situated in this hierarchy, we can



(a) (b) (c) (d)

Figure 2.15: Schematic example of the propagation prodecure from Algorithm 1. Figure (a) depicts the initial hierarchy, homomorphisms are denoted with gray arrows, the origin of rewriting is depicted with the gray node, the ancestors of the origin are depicted with orange and the descendants—with blue. Figure (b) shows the backward stage of propagation, i.e. propagation of restrictive updates to all the ancestors along the orange arrows. Figure (c) shows the forward stage of propagation, i.e. propagation of expansive updates to all the descendants along the blue arrows. Finally, figure (d) illustrates the restored hierarchy edges: orange dashed edges are obtained using the unique homomorphisms given by backward composability (Theorem 2.2.21), blue dashed edges by forward composability (Theorem 2.2.25), gray dashed arrows by composing backward and forward propagations.

construct a *hierarchy of rules* that, applied to the respective objects in the hierarchy, performs the original rewrite together with all the specified propagations. This construction will provide us with the means for analysis of the transformations induced by an SqPO rewrite of a fixed individual object in a hierarchy. It will also allow us to compose transformations induced by consecutive rewrites of two arbitrary objects in the hierarchy and will serve as one of the building blocks for hierarchy audit trails presented in Subsection 2.2.10.

Let us start by defining a couple of notions that will be used in the rest of this subsection. We consider SqPO rewriting rules operating on objects from some fixed category $\mathbf{C}$. Such rules are spans formed by objects and arrows from $\mathbf{C}$.

Definition 2.2.27. A *rule homomorphism* f of $r_1 : L_1 \leftarrow r_1^- - P_1 - r_1^+ \rightarrow R_1$ and $r_2 : L_2 \leftarrow r_2^- - P_2 - r_2^+ \rightarrow R_2$ is given by three homomorphisms $\lambda : L_1 \rightarrow L_2$, $\pi : P_2 \rightarrow P_2$ and $\rho : R_1 \rightarrow R_2$ such that the following diagram commutes:

$$\begin{array}{ccccc} L_1 & \xleftarrow{r_1^-} & P_1 & \xrightarrow{r_1^+} & R_1 \\ \downarrow{\scriptstyle \lambda} & & \downarrow{\scriptstyle \pi} & & \downarrow{\scriptstyle \rho} \\ L_2 & \xleftarrow[r_2^-]{} & P_2 & \xrightarrow[r_2^+]{} & R_2 \end{array} \tag{2.89}$$

We write $f : r_1 \rightarrow r_2$, then $dom(f) = r_1$ and $codom(f) = r_2$.

Definition 2.2.28. Given two rule homomorphisms $f : r_1 \rightarrow r_2$ and $g : r_2 \rightarrow r_3$ with $r_1 : L_1 \leftarrow r_1^- - P_1 - r_1^+ \rightarrow R_1$, $r_2 : L_2 \leftarrow r_2^- - P_2 - r_2^+ \rightarrow R_2$, $r_3 : L_3 \leftarrow r_3^- - P_3 - r_3^+ \rightarrow R_3$, f being defined by homomorphisms $\lambda^f : L_1 \rightarrow L_2$, $\pi^f : P_1 \rightarrow P_2$, $\rho^f : R_1 \rightarrow R_2$ and g being defined by $\lambda^g : L_2 \rightarrow L_3$

$\pi^g : P_2 \to P_3$ and $\rho^g : R_2 \to R_3$, the *composition* of f and g, denoted $g \circ f$, is given by the homomorphisms $\lambda^g \circ \lambda^f$, $\pi^g \circ \pi^f$ and $\rho^g \circ \rho^f$.

Definition 2.2.29. Two rule homomorphisms $f : r_1 \to r_2$ and $g : r_2 \to r_3$ with $r_1 : L_1 \leftarrow r_1^- - P_1 - r_1^+ \to R_1$, $r_2 : L_2 \leftarrow r_2^- - P_2 - r_2^+ \to R_2$, $r_3 : L_3 \leftarrow r_3^- - P_3 - r_3^+ \to R_3$, f being defined by homomorphisms $\lambda^f : L_1 \to L_2$, $\pi^f : P_1 \to P_2$, $\rho^f : R_1 \to R_2$ and g being defined by $\lambda^g : L_2 \to L_3$ $\pi^g : P_2 \to P_3$ and $\rho^g : R_2 \to R_3$ are *equal*, denoted $f = g$, if $\lambda^f = \lambda^g$, $\pi^f = \pi^g$ and $\rho^f = \rho^g$.

It is easy to verify that, using rules as objects and rule homomorphisms as arrows, we obtain the category of rules $\mathbf{Rule}[\mathbf{C}]$ over the category $\mathbf{C}$.

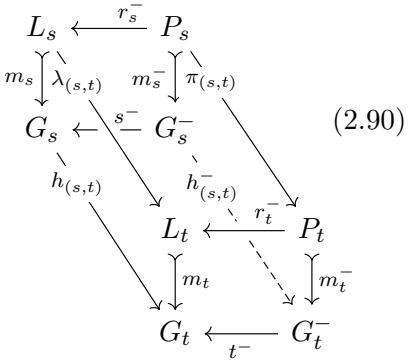
Definition 2.2.30. A *rule hierarchy* is a hierarchy of objects in the category of rules, i.e. a functor $\mathcal{R} : \mathbf{H} \to \mathbf{Rule}[\mathbf{C}]$.

Let $\mathcal{H}$ be a hierarchy of objects in $\mathbf{C}$ and $\mathcal{R}$ be a hierarchy of rules operating on objects in $\mathbf{C}$ both defined over the same skeleton DAG $\mathcal{G} = (V, E \subseteq V \times V)$. For the sake of simplicity, in the rest of this subsection we will assume that we are working on a fixed pair $(\mathcal{H}, \mathcal{R})$ defined over the same skeleton. As a short-hand, for every node $v \in V$ we will denote the object associated to v in $\mathcal{H}$ with G_v and the rule associated to v in $\mathcal{R}$ with $r_v : L_v \leftarrow r_v^- - P_v - r_v^+ \to R_v$. For every edge $(s,t) \in E$ we will denote the associated homomorphism in $\mathcal{H}$ as $h_{(s,t)}$ and the arrows constituting the associated rule homomorphism in $\mathcal{R}$ as $\lambda_{(s,t)}$, $\pi_{(s,t)}$ and $\rho_{(s,t)}$.

Definition 2.2.31. An *instance* of $\mathcal{R}$ in $\mathcal{H}$ is given by a function $\mathcal{I} : V \to Monos(\mathbf{C})$ that associates every node of the skeleton to an instance of the corresponding rule from $\mathcal{R}$ in the corresponding object from $\mathcal{H}$, i.e. $\mathcal{I}(v) : L_v \rightarrowtail G_v$ for all $v \in V$. For every node $v \in V$ we will denote the instance $\mathcal{I}(v)$ as m_v.

Definition 2.2.32. $\mathcal{R}$ is *applicable* to $\mathcal{H}$ through an instance $\mathcal{I}$ if for any pair of nodes $s, t \in V$ such that $(s,t) \in E$:

- $h_{(s,t)} \circ m_s = m_t \circ \lambda_{(s,t)}$, i.e. their instances commute;
- if G_s^- and G_t^- are the results of the restrictive phase of rewriting given by the final pullback complement to r_s^- and m_s, and the final pullback complement of r_t^- and m_t respectively, then there exists a unique $h_{(s,t)}^- : G_s^- \to G_t^-$ that renders Diagram 2.90 commutative.



(2.90)

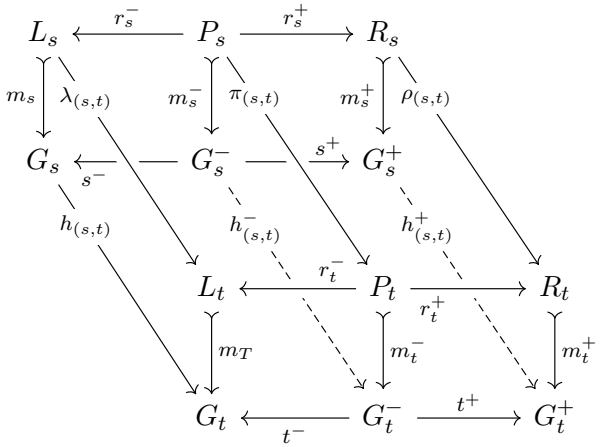
Applying a rule hierarchy

Here we would like to study how a rule hierarchy can be applied to the corresponding hierarchy of objects, namely how given a hierarchy $\mathcal{H}$, a rule hierarchy $\mathcal{R}$ defined over the same skeleton $\mathcal{G} = (V, E)$ and applicable given a set of instances $\mathcal{I}$, we can apply $\mathcal{R}$ through the instances $\mathcal{I}$.

To rewrite $\mathcal{H}$ using $\mathcal{R}$ and $\mathcal{I}$, for every node $v \in V$ of the skeleton, we simply apply the associated rule to the associated object through the instance specified by $\mathcal{I}$ as in Diagram 2.91.

$$\begin{array}{ccccc} L_v & \xleftarrow[r_v^-]{} & P_v & \xrightarrow{r_v^+} & R_v \\ \downarrow{\scriptstyle m_v} & & \downarrow{\scriptstyle m_v^-} & & \downarrow{\scriptstyle m_v^+} \\ G_v & \xleftarrow[g_v^-]{} & G_v^- & \xrightarrow{g_v^+} & G_v^+ \end{array} \tag{2.91}$$

To restore the arrows of $\mathcal{H}$, for every edge $(s,t) \in E$, we use the applicability condition and the universal property of pushouts as follows. Let the back and the front faces of the cube in Diagram 2.92 be two SqPO diagrams corresponding to the above-mentioned rewriting of the objects G_s and G_t respectively. First of all, by the applicability of $\mathcal{R}$ given $\mathcal{I}$, there exists a unique arrow $h^-_{(s,t)}$ such that $h_{(s,t)} \circ s^- = t^- \circ h^-_{(s,t)}$ and $h^-_{(s,t)} \circ m_s^- = m_t^- \circ \pi_{(s,t)}$. This enables us to use the universal property of the pushout G_s^+ and show that there exists a unique arrow $h^+_{(s,t)}$ that renders Diagram 2.92 commutative.



(2.92)

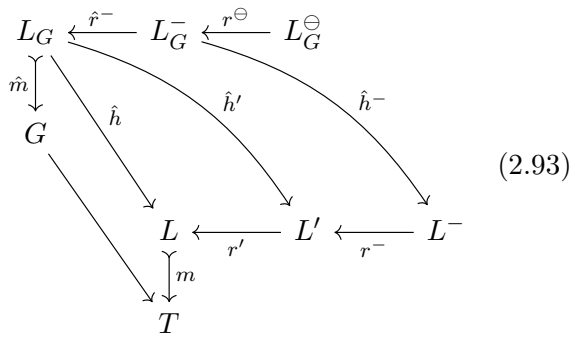
Therefore, to obtain the updated hierarchy, for every hierarchy object, we apply both phases of rewriting and reconstruct the homomorphisms corresponding to hierarchy edges as described above. Because, by definition, all pairs of paths from the same source rule to the same target rule commute, application of the rule hierarchy to the original hierarchy can be seen as a large commutative diagram and it guarantees the consistency of the updated hierarchy.

In the rest of this subsection we are interested in answering the following question: given a hierarchy $\mathcal{H}$, a rewrite of an object situated at the hierarchy node $v \in V$ with a rule $r : L \leftarrow r^- - P - r^+ \rightarrow R$ through an instance $m : L \rightarrowtail G_v$, specification for backward and forward propagation of this rewrite, how can we construct a rule hierarchy $\mathcal{R}$ and an instance $\mathcal{I}$ such that the application of $\mathcal{R}$ to $\mathcal{H}$ through $\mathcal{I}$ performs the specified rewriting and propagation.

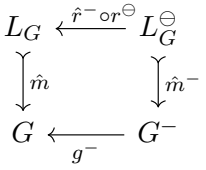
Recall that, upon rewriting of an object in a hierarchy, the objects associated to the ancestors and descendants of the origin of rewriting are updated according to the framework of backward and forward propagation. The objects whose associated hierarchy nodes are neither ancestors nor descendants of the origin stay unaffected by propagation. We would like to construct a rule hierarchy that is defined over the skeleton of $\mathcal{H}$ and, therefore, contains rules for both affected and unaffected objects.

Expressing backward propagations as rules

Let us recall how, given two homomorphic objects $G \mathrel{-}h\mathrel{\to} T$, a restrictive rule $r : L \leftarrow L^-$ applied to an object T though an instance $m : L \rightarrowtail T$, by specifying a backward factorization $L \leftarrow r' - L' \leftarrow r^- - L^-$ and $\hat{h}' : L_G \to L'$ as in Diagram 2.29, we were able to construct the *lifting* $\hat{r}^-$ of r^- along $\hat{h}'$ (see Definition 2.2.11). Such a lifting was defining a rule that, when applied to G through an instance $\hat{m}$, performs the canonical phase of backward propagation to G. In subsection 2.2.2, we have further described means for specifying the clean-up phase of propagation by providing an arrow $r^\ominus$ (see Diagram 2.93 illustrating the rule projection and the clean-up arrow). Recall that we can construct the result of the canonical propagation combined with the clean-up phase by directly finding the final pullback complement $L_G^\ominus \rightarrowtail \hat{m}^- \to G^- \mathrel{-}g^-\mathrel{\to} G$ from $\hat{r}^- \circ r^\ominus$ and $\hat{m}$ (as in Diagram 2.94), i.e. $\hat{r}^- \circ r^\ominus$ defines a *restrictive rule* that, when applied to G through the instance $\hat{m}$ performs both the canonical backward propagation and a clean-up.

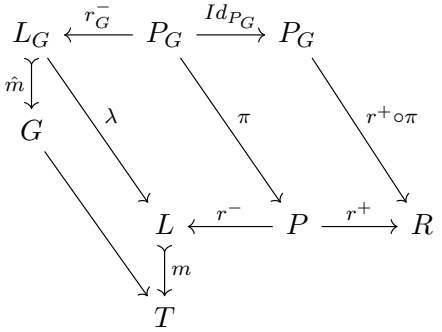


(2.93)



(2.94)

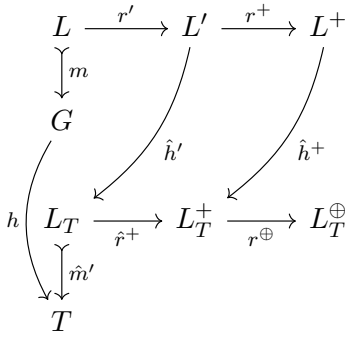
Let us imagine that we are given a general SqPO rule $L \leftarrow r^- - P - r^+ \to R$ applied to T though some $m : L \rightarrowtail G$. Let $r_G^- : L_G \leftarrow P_G$ be a rule constructed according to the previously defined framework of backward propagation of restrictive updates that performs both the canonical propagation and a clean-up for G given $\hat{m} : L_G \rightarrowtail G$. For example, such P_G can be set to $L_G^\ominus$ and r_G^- to $\hat{r}^- \circ r^\ominus$ from Diagram 2.93 above. Let $\lambda : L_G \to L$ and $\pi : P_G \to P$ be two homomorphisms mapping the left-hand side and the interface of the propagation rule to the original rule. For example, in Diagram 2.93, λ corresponds to $\hat{h}$ and π corresponds to $\hat{h}^- \circ r^\ominus$. Setting the right-hand side of the propagation rule to its interface P_G we obtain a general SqPO rule that does not perform any expansive updates, i.e. $L_G \leftarrow r_G^- - P_G - Id_{P_G} \to P_G$. The result of its application is equivalent to the application of the propagation rule given by r_G^-. However, we are able to obtain a *homomorphism* from the propagation rule to the original rewriting rule given by arrows λ, π and $r^+ \circ \pi$ as in Diagram 2.95.



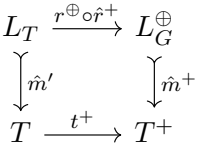
(2.95)

Expressing forward propagations as rules

Recall that given two homomorphic objects $G \mathrel{-}h\mathrel{\to} T$, an expansive rule $r : L \to L^+$ applied to an object G though an instance $m : L \rightarrowtail G$, by specifying a forward factorization $L \mathrel{-}r'\mathrel{\to} L' \mathrel{-}r^+\mathrel{\to} L^+$ and $x : L' \to T$ as in Diagram 2.42, we were able to construct a *projection* $\hat{r}^+$ of r^+ along x (see Definition 2.2.17). This projection was defining a rule that, when applied to T through an instance $\hat{m}'$, performs a canonical phase of forward propagation to T. In Subsection 2.2.3, we have further described means for specifying the clean-up phase of propagation by providing an arrow $r^\oplus$ (see Diagram 2.96 illustrating the rule projection and a clean-up arrow). Recall that we can construct the result of the canonical propagation combined with the clean-up phase by finding the pushout $T \mathrel{-}t^+\mathrel{\to} T^+ \mathrel{\leftarrow}\hat{m}^+\mathrel{\prec} L_T^\oplus$ from $\hat{m}'$ and $r^\oplus \circ \hat{r}^+$ (as in Diagram 2.97), i.e. $r^\oplus \circ \hat{r}^+$ defines an *expansive rule* that, when applied to G through the instance $\hat{m}'$, performs both the canonical forward propagation and a clean-up.



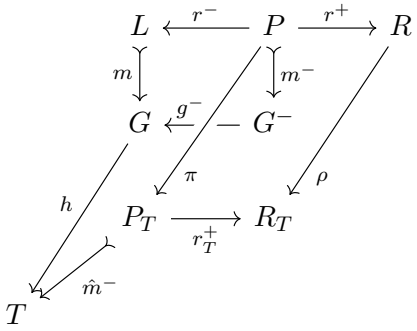
(2.96)



(2.97)

Similarly to the case of backward propagation, let us imagine that we are given a general SqPO rule $L \mathrel{\leftarrow}r^-\mathrel{-} P \mathrel{-}r^+\mathrel{\to} R$ applied to G though some $m : L \rightarrowtail G$. Let G^- be the result of the restrictive phase of rewriting as in Diagram 2.97. Recall that restrictive rewrites do not affect the typing object T, i.e. we have a homomorphism $h \circ g^- : G^- \to T$. Let $r_T^+ : P_T \to R_T$ be a rule constructed according to the previously defined framework of forward propagation of expansive updates that performs both the canonical propagation and a clean-up for T given $\hat{m}^- : P_T \rightarrowtail T$. For example, using Diagram 2.97 above, we can set P_T to L_T, R_T to $L_T^\oplus$, r_T^+ to $r^\oplus \circ \hat{r}^+$ and $\hat{m}^+$ to $\hat{m}'$. Let $\pi : P \to P_T$ and $\rho : R \to R_T$ be two homomorphisms mapping the interface and the right-hand side of the propagation rule to the respective parts of the original rule. For example, in Diagram 2.97, π corresponds to $\hat{h}'$ and ρ corresponds to $r^\oplus \circ \hat{h}^+$.

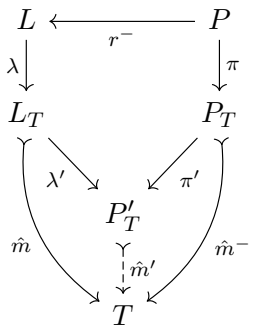
Let us observe Diagram 2.98 presenting the objects and arrows described above. Similarly to the previous case, we would like to construct a rule homomorphism from the original rule to the propagation rule. This propagation rule is given by r_T^+ and the match $\hat{m}^-$ that was constructed according to some specifications for forward propagation to T (e.g. some backward rule factorization and a clean-up arrow). However, during the construction of this rule we have taken into account only



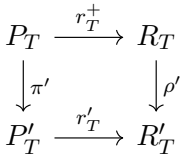
(2.98)

the typing of G^-, and not G. For example, if
the application of r^- removes some elements of G whose typing is not present in the subgraph of T given by P_T, we are not able to find a homomorphism from the left-hand side of the original rule to P_T.

To fix this issue, we first construct the image factorization $L \rightarrow L_T \rightarrowtail T$ of $h \circ m$. The object L_T represents the subobject of T that types the subobject of G affected by the original rewriting. Next, we find the pushout $L_T \rightarrow P'_T \leftarrow P_T$ from $\lambda \circ r^-$ and π as in Diagram 2.99. The constructed object P'_T can be interpreted as the union of L_T and P_T given the original interface P. By the universal property of pushouts, there exists a unique $\hat{m}' : P'_T \rightarrow T$ that renders Diagram 2.99 commutative. Moreover, as can be easily shown, $\hat{m}'$ is a mono. This P'_T can be also interpreted as a *refinement* of P_T that takes into account the typing of elements removed from the original graph G. Having constructed such a refinement, we can find the pushout $P'_T \rightarrow R'_T \leftarrow R_T$ from π' and r_T^+ as in Diagram 2.100. R'_T can be seen as a refinement of R_T that, again, takes into account the typing of elements removed from the original graph G.

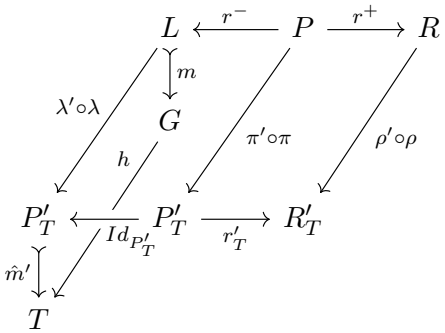


(2.99)



(2.100)

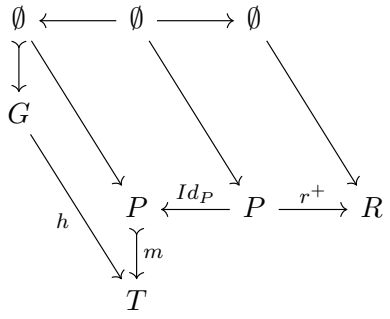
Now, we can construct a general SqPO rule given by the span $P'_T \leftarrow Id_{P'_T} - P'_T \rightarrow R'_T$. It is not hard to illustrate (using the pasting lemmas for pullback complements and pushouts) that the result of its application is equivalent to the application of the original propagation rule given by r_T^+. Moreover, we are able to obtain a *homomorphism* from the original rewriting rule to the propagation rule given by arrows $\lambda' \circ \lambda$, $\pi' \circ \pi$ and $\rho' \circ \rho$ as in Diagram 2.101.



(2.101)

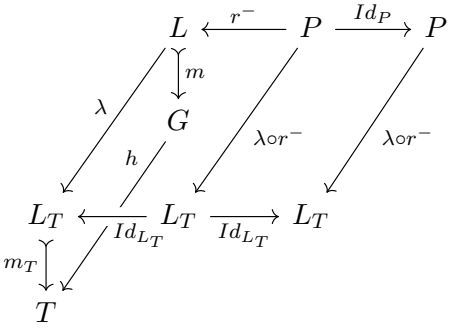
Expressing identity transformations

As previously discussed in Subsection 2.2.6, the objects corresponding to hierarchy nodes that are neither ancestors nor descendants for the origin of rewriting stay unchanged. However, to produce the hierarchy of rules over the same skeleton as the original hierarchy of objects, we still need to construct the rules corresponding to the *identity transformations* of the unchanged objects. To understand how such rules can be constructed, consider the following two scenarios.



(2.102)

Let T be the object corresponding to a descendant of the origin, let $P \leftarrow Id_P - P - r^+ \rightarrow R$ be a forward propagation rule constructed for this descendant and $m : P \rightarrowtail T$ be its instance (recall that forward propagation rules do not perform restrictive updates, therefore, their left-hand side and interface are given by the same object). For any predecessor of this descendant corresponding to the object G and the respective homomorphism $h : G \rightarrow T$, we construct the 'identity' rule $\emptyset \leftarrow \emptyset \rightarrow \emptyset$, where $\emptyset$ corresponds to the initial object of our category $\mathbf{C}$ (see Appendix A.2 for the definition of initial objects). In the categories of graphs, for example, such an object corresponds to an empty graph with no vertices and edges. We then set the instance of this rule to be the unique homomorphism $\emptyset \rightarrowtail G$ as in Diagram 2.102. Note that in this diagram all the unlabeled arrows correspond to the unique homomorphisms from the initial object. We get a rule homomorphism from this identity rule to the specified forward propagation rule given by three unique homomorphisms from the initial object (for example, empty node/edge maps for the categories of graphs).



(2.103)

To study the second case in which identity transformations are applied, consider the object G corresponding to an ancestor of the origin, let $L \leftarrow r^- - P - Id_P \rightarrow P$ be a backward propagation rule constructed for this ancestor and $m : L \rightarrowtail G$ be its instance (recall that backward propagation rules do not perform expansive updates, therefore, their interface and right-hand side are given by the same object). For any successor of this ancestor corresponding to the object T and the respective homomorphism $h : G \rightarrow T$, we construct the 'identity' rule in the following way. First, we find the image factorization $L - \lambda \rightarrow L_T \rightarrowtail_{m_T} \rightarrow$ of $h \circ m$. Then we set the rule to the span $L_T \leftarrow Id_{L_T} - L_T - Id_{L_T} \rightarrow L_T$ and its instance to the arrow m_T as in Diagram 2.103. We get a rule homomorphism from the specified backward propagation rule to this identity rule given by the homomorphisms λ, $\lambda \circ r^-$ and $\lambda \circ r^-$.

Combining rules into a hierarchy

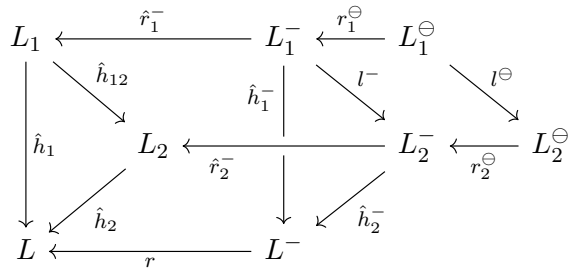
So far we have described how to construct rules corresponding to backward and forward propagations as well as identity rules for the unchanged hierarchy objects. We have also seen how to construct the following rule homomorphisms:

- from a backward propagation rule to the original rewriting rule,
- from the original rewriting rule to a forward propagation rule,
- from an identity rule to a forward propagation rule,
- from a backward propagation rule to an identity rule.

In this section we will discuss how, using the previously presented composability results, we can find homomorphisms between pairs of backward or forward propagation rules, as well as how to find homomorphisms from a backward propagation rule to a forward propagation rule. This will allow us to construct the complete rule hierarchy corresponding to a rewrite and propagation from a single object in a given hierarchy.

In Subsection 2.2.5 we have described the conditions under which the two backward propagations to objects forming an undirected cycle in a hierarchy are composable. Namely, we have seen that, given a commutative triangle formed by $G_1 - h_1 \to T$, $G_2 - h_2 \to T$ and $G_1 - h_{12} \to G_2$, a restrictive rewrite of T with a rule $r : L \leftarrow L^-$ through a match $m : L \rightarrowtail T$, two backward factorizations for G_1 and G_2 as in Diagrams 2.55 and 2.56 respectively, and two clean-up arrows $r_1^\ominus : L_1^- \leftarrow L_1^\ominus$ and $r_2^\ominus : L_2^- \leftarrow L_2^\ominus$ such that there exists a unique $l^\ominus : L_1^\ominus \to L_2^\ominus$ as in Diagram 2.64, produce composable backward propagations. First of all, we have seen that there always exists a unique arrow $\hat{h}_{12} : L_1 \to L_2$ mapping the subobject of G_1 affected by the rewriting to such a subobject of G_2 (recall here L_i is a shorthand for L_{G_i}) that makes Diagram 2.58 commute. We have also seen that, for two composable propagations, there exists a unique arrow $l^- : L_1^- \to L_2^-$ that renders Diagram 2.61 commutative. Finally, the existence of a unique arrow $l^\ominus : L_1^\ominus \to L_2^\ominus$ as in Diagram 2.64 has guaranteed that the result of the clean-up phase of the two backward propagations is composable.

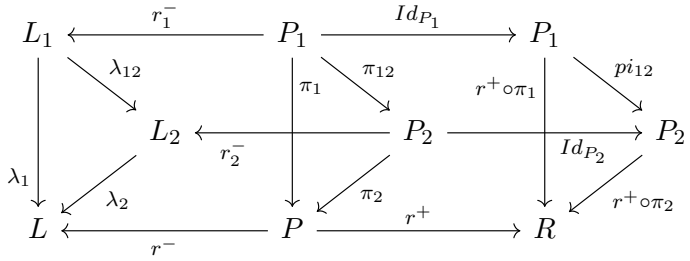
Therefore, having applied the rule given by $\hat{r}_1^- \circ r_1^\ominus$ through $\hat{m}_1$ to G_1 and the one given by $\hat{r}_2^- \circ r_2^\ominus$ through $\hat{m}_2$ to G_2 we were able to obtain composable propagations. Moreover, we have obtained the arrows $\hat{h}_{12}$, l^- and $l^\ominus$ that render Diagram 2.104 commutative.



(2.104)

As before, let us imagine that instead of applying a restrictive rule to T in our undirected cycle formed by G_1, G_2 and T, we have applied a general SqPO rule $r : L \leftarrow r^- - P - r^+ \to R$ through a matching $m : L \rightarrowtail T$. Let $r_1^- : L_1 \leftarrow P_1$ and $r_2^- : L_2 \leftarrow P_2$ be two rules constructed according to our backward propagation framework, performing both the canonical propagation and a clean-up for G_1 and G_2 respectively and in such a way that the resulting propagations are composable. For example, we can set P_1 to $L_1^\ominus$, r_1^- to $\hat{r}_1^- \circ r_1^\ominus$, P_2 to $L_2^\ominus$ and r_2^- to $\hat{r}_2^- \circ r_2^\ominus$ from Diagram 2.104. Recall that, according to our framework, we can obtain the following homomorphisms: $\lambda_1 : L_1 \to L$, $\lambda_2 : L_2 \to L$ (λ_1 corresponds to $\hat{h}_1$ and λ_2 to $\hat{h}_2$ from the previous diagram) $\pi_1 : P_1 \to P$ and $\pi_2 : P_2 \to P$ (similarly, π_1 corresponds to $\hat{h}_1^- \circ r_1^\ominus$ and π_2 to $\hat{h}_2^- \circ r_2^\ominus$). Moreover, because the two propagations are composable we can construct two homomorphism $\lambda_{12} : L_1 \to L_2$ and $\pi_{12} : P_1 \to P_2$ (corresponding to $\hat{h}_{12}$ and $l^\ominus$ in the previous diagram). Setting the right-hand side of the propagation rules to their interfaces we can obtain two general SqPO rules that do not perform expansive updates, i.e. $L_1 \leftarrow r_1^- - P_1 - Id_{P_1} \to P_1$ and $L_2 \leftarrow r_2^- - P_2 - Id_{P_2} \to P_2$. This allows us to obtain three rule homomorphisms: (1) a homomorphism f_1 from the propagation rule for G_1 to the original rule, given by λ_1, π_1 and $r^+ \circ \pi_1$; (2) f_2 from the propagation rule for G_2 to the original rule, given by λ_2, π_2 and $r^+ \circ \pi_2$; and, finally,

(3) f_{12} between the propagation rules for G_1 and G_2, given by λ_{12}, π_{12} and π_{12} (see Diagram 2.105). Moreover, we can check that $f_1 = f_2 \circ f_{12}$, i.e. the following diagram commutes.

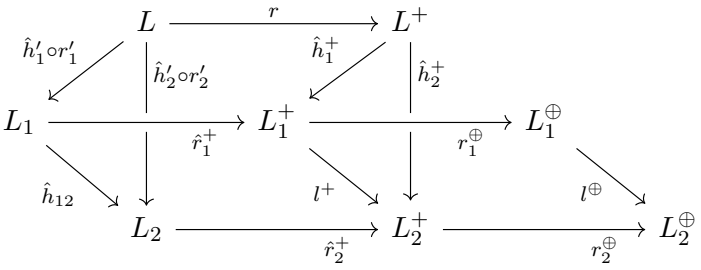


(2.105)

Thus, given a general SqPO rewriting rule, applied in a hierarchy, and specifications for backward propagation of this rule, we can reconstruct a *subhierarchy* of rules performing the original rewrite and the specified backward propagations, given that these propagations are composable.

In Subsection 2.2.5 we have described the conditions under which the two forward propagations to objects forming an undirected cycle in a hierarchy are composable. Namely, we have seen that, given a commutative triangle formed by $G \text{–} h_1 \rightarrow T_1$, $G \text{–} h_2 \rightarrow T_2$ and $T_1 \text{–} h_{12} \rightarrow T_2$, an expansive rewrite of G with a rule $r : L \rightarrow L^+$ through a match $m : L \rightarrowtail G$, two forward factorizations for T_1 and T_2 as in Diagrams 2.70 and 2.71 respectively, and two clean-up arrows $r_1^{\oplus} : L_1^+ \rightarrow L_1^{\oplus}$ and $r_2^{\oplus} : L_2^+ \rightarrow L_2^{\oplus}$ such that there exists a unique $l^{\oplus} : L_1^{\oplus} \rightarrow L_2^{\oplus}$ as in Diagram 2.78, produce composable forward propagations. First of all, we have seen that there exists a unique arrow $\hat{h}_{12} : L_1 \rightarrow L_2$ mapping the subobject of T_1 typing the affected subobject of G to such subobject of T_2 (recall here L_i is a shorthand for L_{T_i}) that makes Diagram 2.74 commute. We have also seen that, for two composable propagations, there exists a unique arrow $l^+ : L_1^+ \rightarrow L_2^+$ that renders Diagram 2.74 commutative. Finally, the existence of a unique arrow $l^{\oplus} : L_1^{\oplus} \rightarrow L_2^{\oplus}$ as in Diagram 2.78 has guaranteed that the result of the clean-up phases of the two forward propagations are composable.

Therefore, having applied the rule given by $r_1^{\oplus} \circ \hat{r}_1^+$ through $\hat{m}_1'$ to T_1 and the one given by $r_2^{\oplus} \circ \hat{r}_2^+$ through $\hat{m}_2'$ to T_2 we were able to obtain composable propagations. Moreover, we have obtained the arrows $\hat{h}_{12}$, l^+ and $l^{\oplus}$ that render Diagram 2.106 commutative.

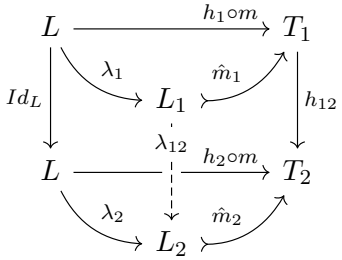


(2.106)

As before, let us imagine that instead of applying an expansive rule to G in our undirected cycle formed by G, T_1 and T_2, we have applied a general SqPO rule $r : L \text{←} r^- \text{–} P \text{–} r^+ \rightarrow R$ through a matching $m : L \rightarrowtail G$. Let $r_1^+ : P_1 \rightarrow R_1$ and $r_2^+ : P_2 \rightarrow R_2$ be two propagation rules constructed according to our forward propagation framework, performing both the canonical propagation and a clean-up for T_1 and T_2 respectively, and in such a way that the resulting propagations are composable. For example, using Diagram 2.106, we can set P_1 to L_1, R_1 to $L_1^{\oplus}$, r_1^+ to $r_1^{\oplus} \circ \hat{r}_1^+$, P_2 to L_2, R_2 to $L_2^{\oplus}$ and r_2^+ to $r_2^{\oplus} \circ \hat{r}_2^+$. Recall that, according to our framework,

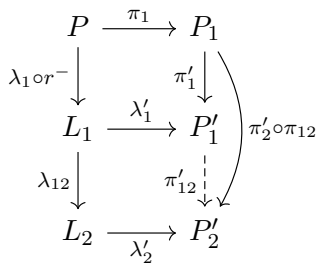
we can obtain the following homomorphisms: $\pi_1 : P \to P_1$, $\pi_2 : P \to P_2$ (π_1 corresponds to $r'_1 \circ \hat{h}'_1$ and π_2 to $r'_2 \circ \hat{h}'_2$ from the previous diagram), $\rho_1 : R \to R_1$ and $\rho_2 : R \to R_2$ (similarly, ρ_1 corresponds to $r_1^{\oplus} \circ \hat{h}_1^{+}$ and ρ_2 to $r_2^{\oplus} \circ \hat{h}_2^{+}$). Moreover, because the two propagations are composable we can construct the two unique homomorphism $\pi_{12} : P_1 \to P_2$ and $\rho_{12} : R_1 \to R_2$ (corresponding to $\hat{h}_{12}$ and $l^{\oplus}$ in the previous diagram).

As we have previously described, to construct general SqPO rules from our forward propagation rules, we find two image factorizations: the factorization $L \rightarrowtail L_1 \twoheadrightarrow T_1$ of $h_1 \circ m$ and $L \rightarrowtail L_2 \twoheadrightarrow T_2$ of $h_2 \circ m$ as in Diagram 2.107. By Lemma A.7.4 there exists a unique arrow $\lambda_{12} : L_1 \to L_2$ rendering this diagram commutative.

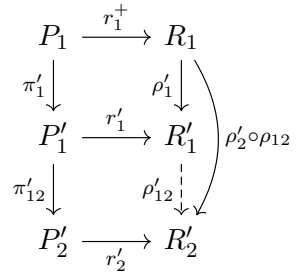


(2.107)

To refine the interface of the propagation rules, we construct two pushouts: $L_1 \rightarrowtail \lambda'_1 \to P'_1 \leftarrow \pi'_1 \twoheadleftarrow P_1$ from $\lambda_1 \circ r^-$ and π_1, and the pushout $L_2 \rightarrowtail \lambda'_2 \to P'_2 \leftarrow \pi'_2 \twoheadleftarrow P_2$ from $\lambda_2 \circ r^-$ and π_2. We can apply the universal property of pushouts and find the unique arrow $\pi'_{12} : P'_1 \to P'_2$ that renders Diagram 2.108 commutative. Finally, to find the refined right-hand side of the propagation rules, we construct another two pushouts: $P'_1 \rightarrowtail r'_1 \to R'_1 \leftarrow \rho'_1 \twoheadleftarrow R_1$ from π'_1 and r_1^{+}, and the pushout $P'_2 \rightarrowtail r'_2 \to R'_2 \leftarrow \rho'_2 \twoheadleftarrow R_2$ from π'_2 and r_2^{+}. As before, we use their universal property to find the unique arrow $\rho'_{12} : R'_1 \to R'_2$ that makes Diagram 2.109 commutative.



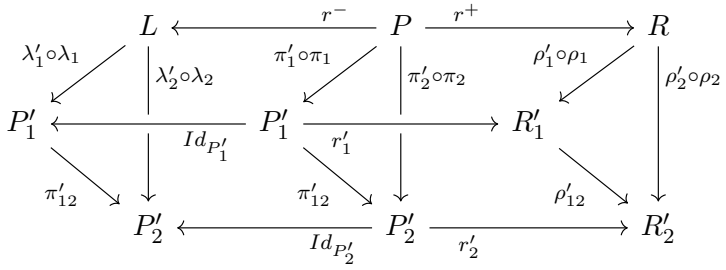
(2.108)



(2.109)

Setting the left-hand side of the propagation rules to their interfaces we can obtain two general SqPO rules that do not perform restrictive updates, i.e. $P'_1 \leftarrow Id_{P_1} \twoheadleftarrow P'_1 \rightarrowtail r'_1 \to R'_1$ and $P'_2 \leftarrow Id_{P_2} \twoheadleftarrow P'_2 \rightarrowtail r'_2 \to R'_2$. This allows us to obtain three rule homomorphisms: (1) a homomorphism f_1 from the original rewriting rule to the propagation rule for T_1, given by $\lambda'_1 \circ \lambda_1$, $\pi'_1 \circ \pi_1$ and $\rho'_1 \circ \rho_1$; (2) f_2 from the original rule to the propagation rule for T_2, given by $\lambda'_2 \circ \lambda_2$, $\pi'_2 \circ \pi_2$ and $\rho'_2 \circ \rho_2$; and, finally, (3) f_{12} between the propagation rules for T_1 and T_2, given by π'_{12}, π'_{12} and ρ'_{12} (see Diagram 2.110). Moreover, we can check that $f_2 = f_{12} \circ f_1$, i.e. the following

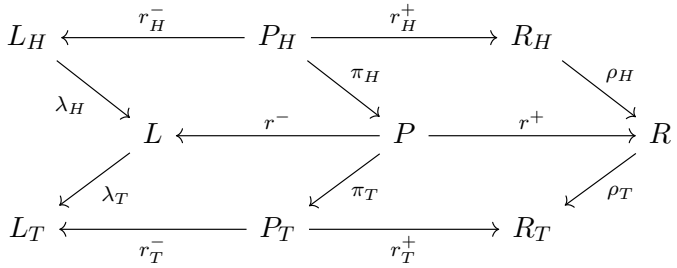
diagram commutes.



(2.110)

Therefore, given a general SqPO rewriting rule applied in a hierarchy and specifications for forward propagation of this rule, we can reconstruct a *subhierarchy* of rules performing the original rewrite and the specified forward propagations, given that these propagations are composable.

Finally, what is left to do is to describe how to obtain the rule homomorphism from a backward propagation rule to a specified forward propagation rule. Consider a diagram below. Let $L \leftarrow r^- - P - r^+ \rightarrow R$ be the original rewriting rule applied to an object in a given hierarchy. Let H be the object corresponding to an ancestor of the origin and $L_H \leftarrow r_H^- - P_H - r_H^+ \rightarrow R_H$ be a backward propagation rule constructed for this ancestor. Let λ_H, π_H and ρ_H be three homomorphisms defining a rule homomorphism from this propagation rule to the original one. Similarly, let T be the object corresponding to a descendant of the origin and $L_T \leftarrow r_T^- - P_T - r_T^+ \rightarrow R_T$ be a forward propagation rule constructed for this descendant. Let λ_T, π_T and ρ_T be three homomorphisms defining a rule homomorphism from the original rule to this propagation rule. The homomorphism from the propagation rule applied to H to the propagation rule applied to T can be constructed by simply composing the rule homomorphisms in the diagram, i.e. by taking the homomorphisms $\lambda_T \circ \lambda_H$, $\pi_T \circ \pi_H$ and $\rho_T \circ \rho_H$.



(2.111)

Applicability of constructed rule hierarchies

Here we would like to discuss how a rule hierarchy obtained using the previously described constructions can be applied to the initial hierarchy of objects to produce the result of rewriting and propagation.

Proposition 2.2.33. Given a hierarchy $\mathcal{H}$, a rewrite of an object situated at the node $v \in V$ with a rule $r : L \leftarrow r^- - P - r^+ \rightarrow R$ through an instance $m : L \rightarrowtail G_v$, and a specification for backward and forward propagation of this rewrite, let $\mathcal{R}$ and $\mathcal{I}$ be the rule hierarchy and its instances constructed using the framework described above. Then $\mathcal{R}$ is applicable to $\mathcal{H}$ given $\mathcal{I}$.

Proof. See Appendix B. □

Therefore, using the described constructions, we can obtain an applicable rule hierarchy that, when applied to the original hierarchy of objects, performs the specified rewriting and propagation.

2.2.8 Reversible rule hierarchies

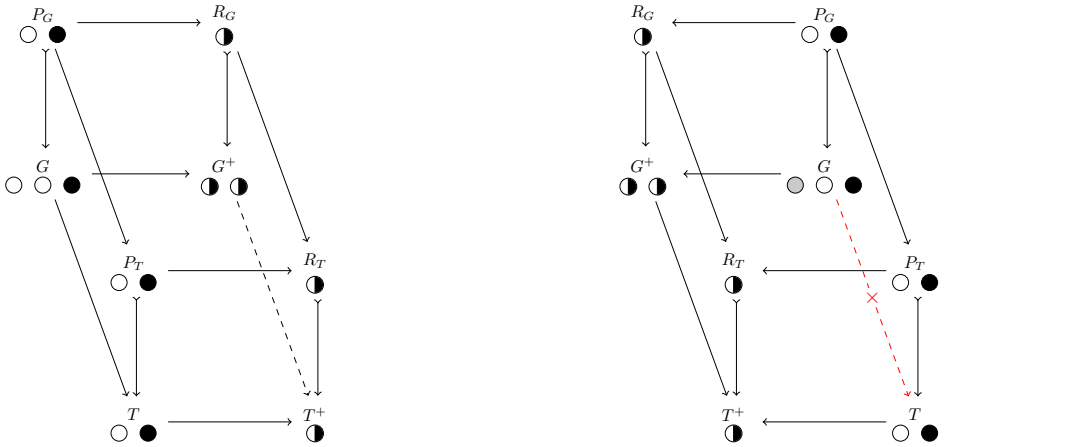
In this subsection we would like to study the side-effects introduced by the application of a rule hierarchy. As in the case of individual SqPO rewriting rules, these side-effects are graph transformations not explicitly specified by the underlying rules. Moreover, these side-effects can represent some implicit changes to the homomorphisms representing hierarchy edges. Due to such implicit changes, having applied a rule hierarchy, we may not be able to restore the original hierarchy of objects by simply looking at the applied rule hierarchy. The side-effects of SqPO rewriting on individual objects have already been discussed in Subsection 2.1.2. Here we will study the second kind of side-effect, namely implicit changes to the homomorphisms in a hierarchy.

In general, the second kind of side-effect introduced by the application of a rule hierarchy makes the rewriting produced by reversing the original rule hierarchy not *applicable*. To understand the nature of the side-effects affecting hierarchy homomorphisms, let us consider the following example.

Example 2.2.10. *Let G and T in Subfigure 2.16a be two homomorphic objects and let the arrows $P_G \to R_G$ and $P_T \to R_T$ specify expansive phases of two homomorphic rules applied to these objects. For example, G and T can represent the result of the restrictive rewriting phase given some rule hierarchy, while $P_G \to R_G$ and $P_T \to R_T$ are the second arrows of the rules applied to G and T. The object G has two instances of the white circle and the rule $P_G \to R_G$ selects one of these instances and merges it with an instance of the black square. The rule $P_T \to R_T$ selects the white and the black circle and merges them. The unique arrow $G^+ \to T^+$ is obtained by the universal property of the pushout that constructed G^+. As a side-effect of such a merge, after the application of the rules, the first instance of the white circle in G is also typed by the merged node in T^+. As a consequence, we 'forget' that it was an instance of the white circle in T. In Subfigure 2.16b we reverse our rules and apply them to the resulting objects G^+ and T^+. We select the merged node in T^+ and we clone it into two circle nodes: the white circle and the black circle. In G^+ we select one instance of the merged node and clone it. As a result, we recover the object G, however, we are no longer able to type this object by the original T. This happens precisely because, as a side-effect of our hierarchy transformation, we 'forgot' how the circle denoted with gray in Subfigure 2.16b was typed in T. From the formal point of view, this happens, because, while finding a unique arrow G to T, we fail to use the universal property of the final pullback complement that constructed T.*

Definition 2.2.34. The *reverse* $\mathcal{R}^{-1}$ of $\mathcal{R}$ is the rule hierarchy whose nodes correspond to the rules r_v^{-1} for all $v \in V$, and whose edges correspond to the rule homomorphisms $(\rho_{(s,t)}, \pi_{(s,t)}, \lambda_{(s,t)})$ for all edges $(s,t) \in E$.

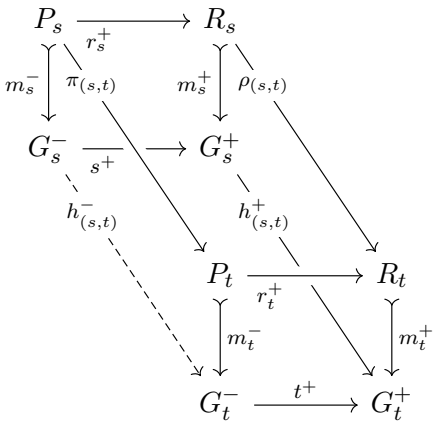
Definition 2.2.35. Rewriting of $\mathcal{H}$ with $\mathcal{R}$, applicable through an instance $\mathcal{I}$, is *reversible*, if rewriting of every individual object is reversible and the reverse $\mathcal{R}^{-1}$ is applicable, i.e. for any pair of nodes $s, t \in V$ such that $(s,t) \in E$ corresponding to objects and rules as in Diagram 2.92, if G_s^- is given as the final pullback complement of r_s^+ and m_s^+ and G_t^- as the final pullback



(a) Application of the original rules *(b) Application of the reversed rules*

Figure 2.16: Example of side-effects affecting hierarchy homomorphisms

complement of r_t^+ and m_t^+, there exists a unique homomorphism $h^-_{(s,t)} : G_s^- \to G_t^-$ that makes the cube in Diagram 2.112 commute.



(2.112)

Constructing reversible rule hierarchies

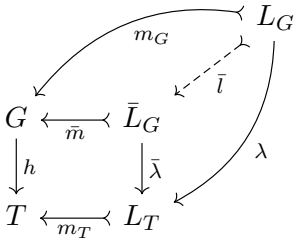
Similarly to simple SqPO rewriting, for the practical applications of our interest we would like to develop a constructive procedure that, given an arbitrary rule hierarchy and its instance inside some initial hierarchy of objects, allows us to obtain its reversible version. We refer to the resulting rule hierarchy as the *reversible rule hierarchy refinement*.

According to Definition 2.2.35, to make a rule hierarchy reversible given specified rule instances, we need to make sure that: (1) application of every rule in the hierarchy is reversible, and (2) the reverse of the hierarchy is applicable. To satisfy the first condition, we need to find the reversible rule refinement for every individual rule in the hierarchy. As we have discussed in Subsection 2.1.2, the procedure for finding such refinements is determined by the category in which we are working (for example, in 2.1.2 we have described such procedure for **SimpGrph**).

From now on, we will assume that we are working with a fixed instance $\mathcal{I}$, individual rewrites given this instance are reversible and the rule hierarchy is applicable given $\mathcal{I}$. On the other hand, to satisfy the second condition, we can design an abstract category-independent procedure, and to do so, we need to recall the rule constructions presented in Subsection 2.2.7.

Because the reversibility condition that we call 'applicability of the reverse' concerns the second phase of rule hierarchy application, we need to have a closer look at the rules that perform expansive updates. These are the rules that introduce the side-effects of the second kind mentioned previously. As before, consider two homomorphic objects $G \mathrel{-}h\mathrel{\to} T$ and let $r_G : L_G \mathrel{\leftarrow} r_G^- \mathrel{-} P_G \mathrel{-} r_G^+ \mathrel{\to} R_G$ and $r_T : L_T \mathrel{\leftarrow} r_T^- \mathrel{-} P_T \mathrel{-} r_T^+ \mathrel{\to} R_T$ be two rules corresponding to these objects such that r_T specifies an expansive update. Let $m_G : L_G \rightarrowtail G$ and $m_T : L_T \rightarrowtail T$ be two instances of the above mentioned rules. To refine the rule hierarchy containing these objects given the specified instances we perform the following sequence of steps.

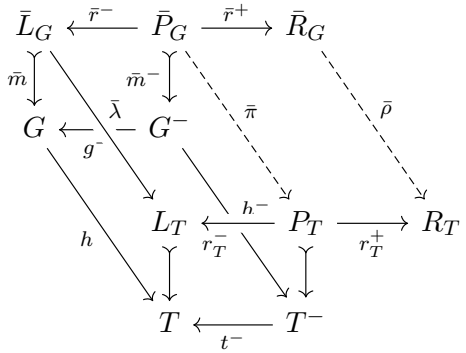
First of all, we construct the pullback $G \mathrel{\leftarrow} \bar{m} \mathrel{\prec} \bar{L}_G \mathrel{-} \bar{\lambda} \mathrel{\to} L_T$ from h and m_T. By the universal property of this pullback there exists a unique arrow $\bar{l} : L_G \to \bar{L}_G$ that renders Diagram 2.113 commutative, and this arrow is a mono. Then, we find the final pullback complement $P_G \mathrel{\succ} \bar{p} \mathrel{\to} \bar{P}_G \mathrel{-} \bar{r}^- \mathrel{\to} \bar{L}_G$ to r_G^- and $\bar{l}$, followed by the pushout $\bar{P}_G \mathrel{-} \bar{r}^+ \mathrel{\to} \bar{R}_G \mathrel{\leftarrow} \bar{r} \mathrel{\prec} R_G$ from r_G^+ and $\bar{p}$, as in Diagram 2.114.



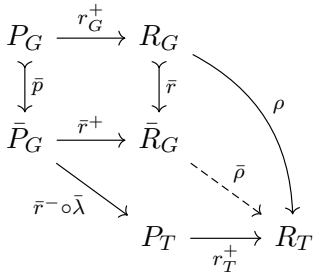
(2.113)

$$\begin{array}{ccccc} L_G & \xleftarrow{r_G^-} & P_G & \xrightarrow{r_G^+} & R_G \\ \downarrow{\scriptstyle \bar{l}} & & \downarrow{\scriptstyle \bar{p}} & & \downarrow{\scriptstyle \bar{r}} \\ \bar{L}_G & \xleftarrow{\bar{r}^-} & \bar{P}_G & \xrightarrow{\bar{r}^+} & \bar{R}_G \end{array} \tag{2.114}$$

Because the rule hierarchy is applicable, by definition, there exists a unique homomorphism $h^- : G^- \to T^-$, i.e. the results of the restrictive rewriting phases of G and T are homomorphic. Using the universal properties of final pullback complements and pullbacks, it can be shown that there exists a unique arrow $\bar{\pi} : \bar{P}_G \to P_T$ that renders Diagram 2.115 commutative. We now find the arrow $\bar{\rho}$ that makes Diagram 2.115 commute by applying the universal property of the pushout that constructed $\bar{R}_G$ (we can show that there exists a unique arrow $\bar{\rho}$ that makes Diagram 2.116 commute).



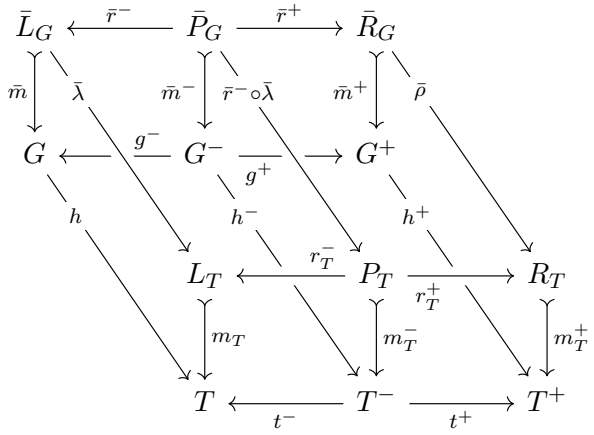
(2.115)



(2.116)

The following proposition shows that, given the above-constructed refinement of r_G specified by $\bar{r}_G : \bar{L}_G \leftarrow \bar{r}^- - \bar{P}_G - \bar{r}^+ \rightarrow \bar{R}_G$, the reverse of the application of the pair of rules $\bar{r}_G$ and r_T is applicable.

Proposition 2.2.36. In Diagram 2.117, let $G^+ - h^+ \rightarrow T^+$ be the result of the rewrite of $G - h \rightarrow T$ with the pair of rules $\bar{r}_G$ and r_T, their homomorphism given by the arrows $\bar{\lambda}$, $\bar{r}^- \circ \bar{\lambda}$ and $\bar{\rho}$, and the respective instances $\bar{m}$ and m_T. The right-most face of the cube in the diagram is a pullback, i.e. $G^+ \leftarrow \bar{m}^+ \prec R'_G - \bar{\rho} \rightarrow R_T$ is the pullback from $G^+ - h^+ \rightarrow T^+ \leftarrow m_T^+ \prec R_T$.



(2.117)

Proof. See Appendix B. □

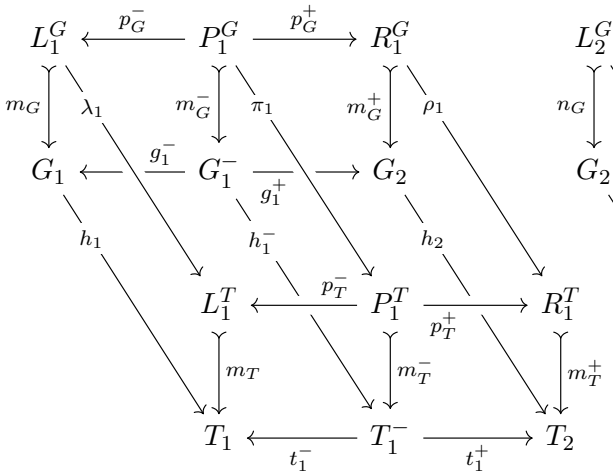
Note that we perform the previously described refinement for any rule $r_T : L_T \leftarrow r_T^- - P_T - r_T^+ \rightarrow R_T$ that performs an expansive update, independently of the nature of the rule $r_G : L_G \leftarrow r_G^- - P_G - r_G^+ \rightarrow R_G$ homomorphic to it. If $r_T^+ = Id_{P_T}$, i.e. our rule is of the form $r_T : L_T \leftarrow r_T^- - P_T - Id_{P_T} \rightarrow P_T$, it does not perform an expansive update. Then, the results of the first and the second rewriting phases are isomorphic. In this case the reverse is always applicable. Namely, if $g^+ : G^- \rightarrow G^+$ and $t^+ : T^- \rightarrow T^+$ are the results of the second rewriting phase for r_G and r_T and $h^+ : G^+ \rightarrow T^+$ is their homomorphism (whose existence and uniqueness is guaranteed by the applicability of our hierarchy), then the unique $h^- : G^- \rightarrow T^-$ can be obtained by a simple composition $h^+ \circ g^+$.

2.2.9 Composition of rule hierarchies

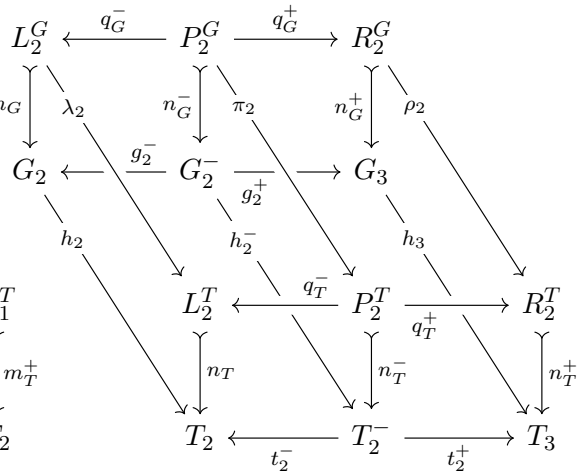
In this subsection we would like to study how consecutive applications of rule hierarchies can be composed into a single application. Similarly to such composition for individual SqPO rewrites, discussed in Subsection 2.1.3, composition of rule hierarchies is useful when maintaining the history of hierarchy updates or multiple versions of the hierarchy. Moreover, it is extensively used in the hierarchy audit trail discussed in Subsection 2.2.10. To construct rule composition, as before, we require (1) the category in wich we are working to be adhesive and (2) the pushout factorizations of pullbacks along monos to be monic. Recall that this is always the case in the concrete categories of interest (e.g. sets, graphs).

Let $\mathcal{H}_1$ be a hierarchy corresponding to two homomorphic objects $G_1 - h_1 \rightarrow T_1$ and let $\mathcal{R}_1$ be a rule hierarchy corresponding to rules $p_G : L_1^G \leftarrow p_G^- - P_1^G - p_G^+ \rightarrow R_1^G$ and $p_T : L_1^T \leftarrow p_T^- - P_1^T - p_T^+ \rightarrow R_1^T$, whose homomorphism $f^p : p_G \rightarrow p_T$ is given by arrows λ_1, π_1 and ρ_1 as in Diagram 2.118. Let

$G_2 - h_2 \to T_2$ correspond to the hierarchy $\mathcal{H}_2$, the result of the application of $\mathcal{R}_1$ through the instances m_G and m_T to $\mathcal{H}_1$ (we assume that $\mathcal{R}_1$ is applicable given m_G and m_T). Let $\mathcal{R}_2$ be another rule hierarchy given by a homomorphic pair of rules $q_G : L_2^G \leftarrow q_G^- - P_2^G - q_G^+ \to R_2^G$ and $q_T : L_2^T \leftarrow q_T^- - P_2^T - q_T^+ \to R_2^T$ as in Diagram 2.119. Their homomorphism $f^q : q_G \to q_T$ is given by arrows λ_2, π_2 and ρ_2. Let $G_3 - h_3 \to T_3$ correspond to the hierarchy $\mathcal{H}_3$, the result of the application of $\mathcal{R}_2$ through the instances n_G and n_T to $\mathcal{H}_2$, as in the diagram (similarly, we assume that $\mathcal{R}_2$ is applicable given the instances).

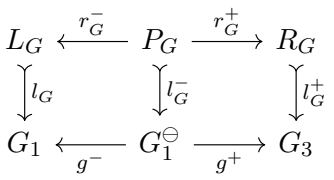


(2.118)

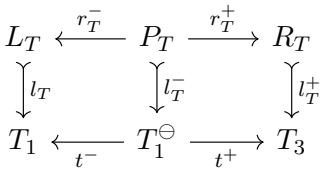


(2.119)

We can compose these pairs of rewrites using the constructions presented in Subsection 2.1.3. Namely, if the rules p_G and p_T are reversible, we can find a pair of rules $r_G : L_G \leftarrow r_G^- - P_G - r_G^+ \to R_G$ and $r_T : L_T \leftarrow r_T^- - P_T - r_T^+ \to R_T$, together with a pair of instances $l_G : L_G \rightarrowtail G_1$ and $l_T : L_T \rightarrowtail T_1$, such that, applying r_G to G_1 and r_T to T_1 through l_G and l_T respectively (as in Diagrams 2.120 and 2.121), we directly obtain G_3 and T_3 from Diagram 2.119.



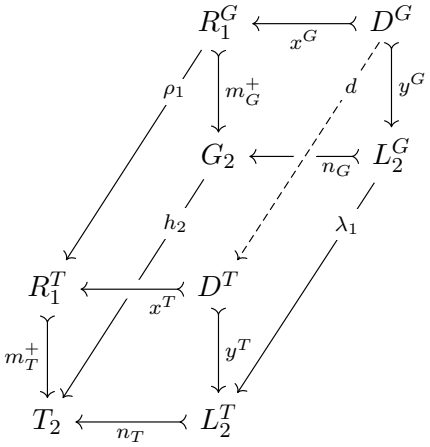
(2.120)



(2.121)

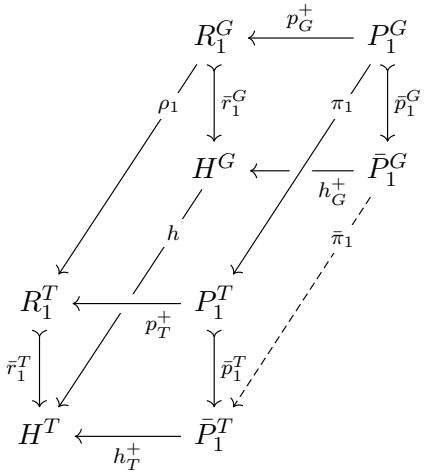
To be able to construct a rule homomorphism $f : r_G \to r_T$, we need to make an assumption that the rewriting specified by $\mathcal{R}_1$ given m_G and m_T is reversible, i.e. for G_1^- being the final pullback complement of p_G^+ and m_G^+, and T_1^- to p_T^+ and m_T^+, there always exists a unique arrow h_1^- that renders the right-most cube in Diagram 2.118 commutative.

Let D^G, x^G, y^G, D^T, x^T and y^T from Diagram 2.122 be the overlaps of respectively R_1^G with L_2^G and R_1^T with L_2^T, constructed as described in Subsection 2.1.3 and denoted with o^G and o^T. By the universal property of pullbacks, there exists a unique arrow $d : D^G \to D^T$ that renders Diagram 2.122 commutative. Using $D^G - d \to D^T$ we can construct a hierarchy of such overlaps defined over the same skeleton as $\mathcal{H}_1$, and together with arrows x^G, y^G, x^T and y^T, such a hierarchy gives us the *hierarchy overlap* $\mathcal{O}$.

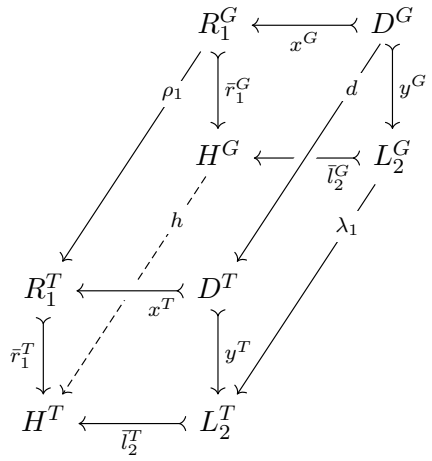


(2.122)

Let $R_1^G \rightarrowtail \bar{r}_1^G \rightarrow H^G \leftarrow \bar{l}_2^G \prec L_2^G$ be the pushout from x^G and y^G, and $R_1^T \rightarrowtail \bar{r}_1^T \rightarrow H^T \leftarrow \bar{l}_2^T \prec L_2^T$ be the pushout from x^T and y^T, as in Diagram 2.124. We can use the universal property of the pushout that constructed H^G to show that there exists a unique arrow $h : H^G \rightarrow H^T$ that renders our diagram commutative. Now, let the object $\bar{P}_1^G$ be constructed as the final pullback complement of p_G^+ and $\bar{r}_1^G$, and $\bar{P}_1^T$ of p_T^+ and $\bar{r}_1^T$, as in Diagram 2.123. We would like to show that there exists a unique arrow $\bar{\pi}_1 : \bar{P}_1^G \rightarrow \bar{P}_1^T$ that renders this diagram commutative.

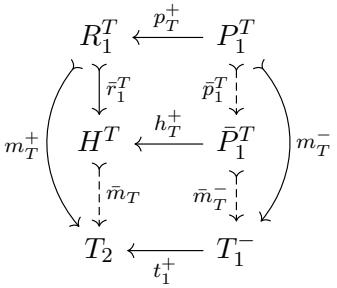


(2.123)

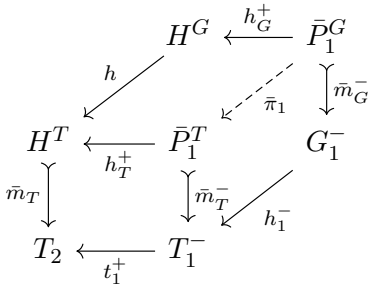


(2.124)

First of all, let us recall that, by our assumption, the rule p_T is reversible, therefore, T_1^- can be obtained as the final pullback complement of p_T^+ and m_T^+. As have been described in Subsection 2.1.3 (see Diagrams 2.14 and 2.17), we can find arrows $\bar{m}_T : H^T \rightarrowtail T_2$ and $\bar{m}_T^- : \bar{P}_1^- \rightarrowtail T_1^-$ as in Diagram 2.125. Moreover, by the vertical pasting lemma for final pullback complements, it is not hard to show that $\bar{P}_1^T \rightarrowtail \bar{m}_T^- \rightarrow T_1^- \rightarrow t_1^+ \rightarrow T_2$ is the final pullback complement of h_T^+ and $\bar{m}_T$. Recall also that, by the applicability of the reverse for the homomorphic pair of rules p_G and p_T, having constructed G_1^- as the final pullback complements of p_G^+ and m_G^+, and T_1^- of p_T^+ and m_T^+, there exists a unique arrow h_1^- that renders Diagram 2.119 commutaive. This allows us to apply the universal property of the pullback $\bar{P}_1^T$ and show that there exists a unique arrow $\bar{\pi}_1 : \bar{P}_1^G \rightarrow \bar{P}_1^T$ such that Diagram 2.126 commutes. Moreover, it is not hard to further verify that $\bar{\pi}_1$ is also the unique arrow that renders Diagram 2.123 commutative.

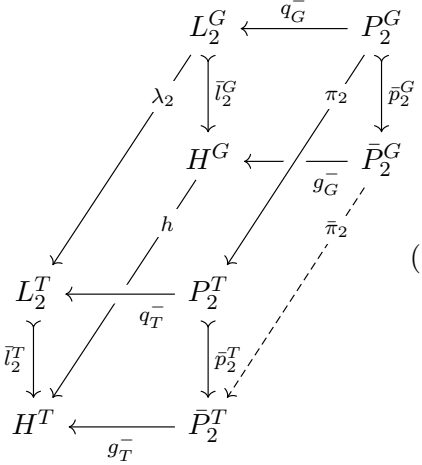


(2.125)



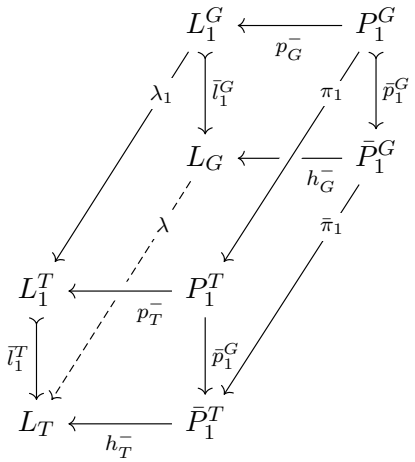
(2.126)

For $\bar{P}_2^G$ being the final pullback complement of q_G^- and $\bar{l}_2^G$, and $\bar{P}_2^T$ of q_T^- and $\bar{l}_2^T$, as in Diagram 2.127, we can show that there exists a unique arrow $\bar{\pi}_2 : \bar{P}_2^G \to \bar{P}_2^T$ that renders this diagram commutative in a similar way, by simply using the applicability of the homomorphic pair of rules q_G and q_T.

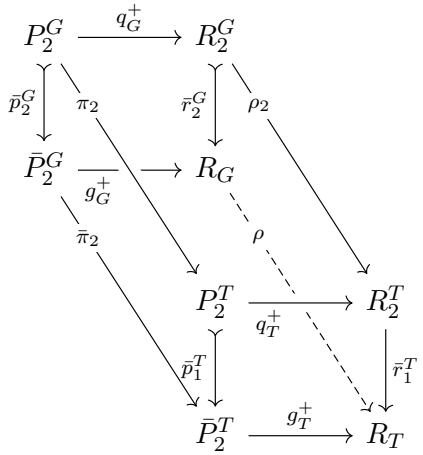


(2.127)

To find the left-hand side of the composed rules for G_1 and T_1 we construct the pushout $L_1^G \rightarrowtail \bar{l}_1^G \to L_G \leftarrow h_G^- - \bar{P}_1^G$ from p_G^- and $\bar{p}_1^G$ and the pushout $L_1^T \rightarrowtail \bar{l}_1^T \to L_T \leftarrow h_T^- - \bar{P}_1^T$ from p_T^- and $\bar{p}_1^T$ as in Diagram 2.128. By the universal property of pushouts there exists a unique arrow $\lambda : L_G \to L_T$ that renders this diagram commutative. In a similar way we construct the right-hand side of the composed rules by finding the pushout $\bar{P}_2^G - g_G^+ \to R_G \leftarrow h_G^- \prec R_2^G$ from $\bar{p}_2^G$ and q_G^+ and the pushout $\bar{P}_2^T - g_T^+ \to R_T \leftarrow h_T^- \prec R_2^T$ from $\bar{p}_2^T$ and q_T^+ as in Diagram 2.129. By the universal property of pushouts, there exists a unique arrow $\rho : R_G \to R_T$ that renders this diagram commutative.

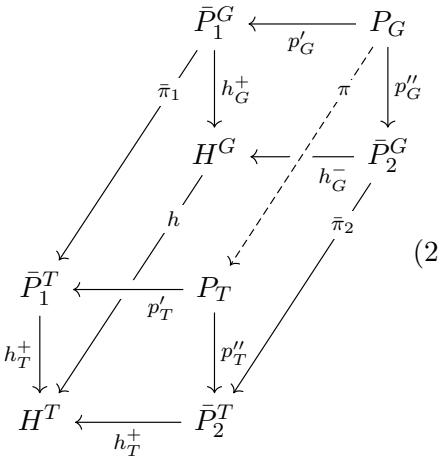


(2.128)



(2.129)

Finally, we find the interface of the composed pair of rules by finding the pullback $\bar{P}_1^G \leftarrow p'_G - P_G - p''_G \rightarrow \bar{P}_2^G$ from h_G^+ and h_G^- and the pullback $\bar{P}_1^T \leftarrow p'_T - P_T - p''_T \rightarrow \bar{P}_2^T$ from h_T^+ and h_T^- as in Diagram 2.129. By the universal property of pullbacks there exists a unique arrow $\pi : P_G \rightarrow P_T$ that renders this diagram commutative.



(2.130)

Using the previously presented constructions, the composed rules correspond to the pair of spans $r_G : L_G \leftarrow h_G^- \circ p'_G - P_G - g_G^+ \circ p''_G \rightarrow R_G$ and $r_T : L_T \leftarrow h_T^- \circ p'_T - P_T - g_T^+ \circ p''_T \rightarrow R_T$. The homomorphism $f : r_G \rightarrow r_T$ is given by the arrows λ, π and ρ. Therefore, the homomorphic rules $r_G - f \rightarrow r_T$ constitute a rule hierarchy $\mathcal{R}$ that we call the *composed rule hierarchy* given the hierarchy overlap $\mathcal{O}$ and we write $\mathcal{R} = \otimes(\mathcal{R}_1, \mathcal{O}, \mathcal{R}_2)$. In Subsection 2.1.3 we have described how an arrow $l_G : L_G \rightarrowtail G_1$ can be found (see Diagram 2.20). This arrow defines the matching of the left-hand side of the composed rule into G_1. Similarly, we can construct the matching $l_T : L_T \rightarrowtail T_1$ of the left-hand side of the composed rule in T_1. Therefore, l_G and l_T give us an instance of $\mathcal{R}$ in $\mathcal{H}_1$.

Theorem 2.2.37. *In adhesive categories, if rewriting specified by $\mathcal{R}_1$ is reversible, $\mathcal{R}$ is applicable given the instances l_G and l_T, and its application results into $G_3 - h_3 \rightarrow T_3$.*

Proof. See Appendix B. □

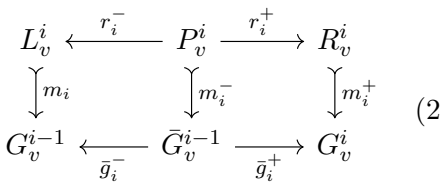
Lemma 2.2.38. In adhesive categories, the composition of two reversible hierarchy rewrites is a reversible rewrite.

Proof. The proof can be obtained by combining Lemma 2.1.6 and Theorem 2.2.37, where the rule hierarchy is given by $\mathcal{R}^{-1}$ and is applied to $G_3 - h_3 \rightarrow T_3$ through $o_G^+ : R_G \rightarrowtail G_3$ and $o_T^+ : R_T \rightarrowtail T_3$. □

2.2.10 Audit trail for rewriting in hierarchies

In this subsection we describe how reversibility and composition of rewriting can be used to construct an audit trail for transformations in hierarchies (e.g. hierarchies of graphs with attributes representing our KR system). The described audit trail system is implemented as part of the `ReGraph` library and discussed in more detail in Section 2.3.4.

Let $\mathcal{H}_0$ be the starting hierarchy of objects, defined over a skeleton DAG $\mathcal{G} = (V, E)$, whose history of transformations we would like to maintain. Let $\langle \mathcal{R}_i \mid i \in [1 \dots n] \rangle$ be a sequence of rule hierarchies consecutively applied to $\mathcal{H}_0$ through the corresponding instances $\mathcal{I}_i$, resulting in a sequence of hierarchies $\langle \mathcal{H}^i \mid i \in [1 \dots n] \rangle$ with the right-hand side instances given by $\mathcal{I}_i^+$ for $1 \leq i \leq n$, i.e. for every $v \in V$ and $1 \leq i \leq n$, $\mathcal{I}_i^+(v) : R_v^i \rightarrowtail G_v^i$ and Diagram 2.131 is a SqPO diagram. As in the case of individual objects, to be able to build a sound audit trail, we require each rewrite in the sequene to be reversible.



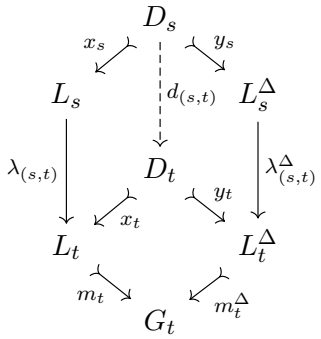
(2.131)

Definition 2.2.39. The *audit trail* for the resulting hierarchy $\mathcal{H}^n$ consists of the sequence of rule hierarchies $\langle \mathcal{R}_i \mid i \in [1 \dots n] \rangle$ and the right-hand side instances $\mathcal{I}_i^+$ for $1 \leq i \leq n$.

Rollback. Using the audit trail we can *rollback* rewriting to any point in the history of transformation corresponding to some intermediate hierarchy $\mathcal{H}^i$ for $0 \leq i \leq n-1$. This can be done by applying the rule hierarchies $\langle \mathcal{R}_j^{-1} \mid j \in [n \dots i+1] \rangle$ with the corresponding instances $\mathcal{I}_j^+$, where $\mathcal{I}_j^+(v) : R_v^j \rightarrowtail G_v^j$ for every $v \in V$ and $j \in [n \dots i+1]$.

Maintain diverged versions. To be able to accommodate multiple versions of a hierarchy, we use delta compression, as in the case of individual objects. Let v_1 and v_2 be two versions of the starting hierarchy $\mathcal{H}^0$ with v_1 being the current version. The initial delta Δ from v_1 to v_2 is set to the identity rule hierarchy with the rule $\emptyset \leftarrow Id_\emptyset - \emptyset - Id_\emptyset \rightarrow \emptyset$ corresponding to every node $v \in V$. We set the instance $\mathcal{I}(v)$ for every $v \in V$ to be the unique homomorphism $u_v : \emptyset \rightarrowtail G_v^0$. Every rewrite of the current version of the hierarchy induces an update of the delta that consists of the composition of the previous delta and the reverse of the applied rule hierarchy.

Let v_1 be the current version corresponding to some hierarchy $\mathcal{H}$ (e.g. obtained as the result of transformation of the initial hierarchy $\mathcal{H}^0$). Let $\mathcal{R}_\Delta$ and $\mathcal{I}_\Delta$ be respectively the rule hierarchy and the instance given by Δ, where $r_v^\Delta : L_v^\Delta \leftarrow r_{v,\Delta}^- - P_v^\Delta - r_{v,\Delta}^+ \rightarrow R_v^\Delta$ and $m_v^\Delta : L_v^\Delta \rightarrowtail G_v$ are the rule and the instance corresponding to a node $v \in V$. Let $\mathcal{R}$ be a rule hierarchy applied to $\mathcal{H}$ through the instance $\mathcal{I}$ and $\mathcal{H}'$ be the result of the corresponding rewriting. The new delta is given by the rule hierarchy and the instance obtained by constructing the composition $\otimes(\mathcal{R}^{-1}, \mathcal{O}, \mathcal{R}^\Delta)$ with $\mathcal{O}$ being the hierarchy overlap computed by finding the overlaps between L_v and L_v^Δ for every node $v \in V$ and the homomorphisms between overlap objects found by the universal property of final pullback complements, as in Diagram 2.132, for every edge $(s,t) \in E$.



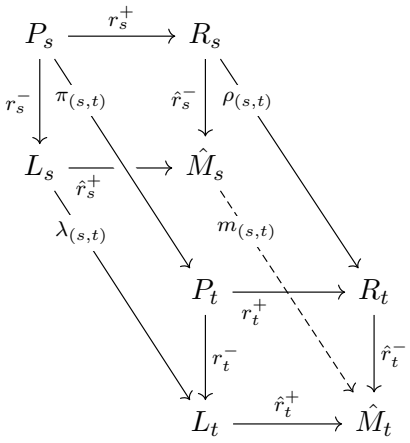
(2.132)

Switch version. Switching between different versions of the hierarchy is performed by applying the rule hierarchy though the instance given by the delta. If v_1 is the current version corresponding to a hierarchy $\mathcal{H}$ with the delta given by $\Delta = (\mathcal{R}_\Delta, \mathcal{I}_\Delta)$, switching to v_2 is performed by applying $\mathcal{R}_\Delta$ to $\mathcal{H}$ through $\mathcal{I}_\Delta$. If $\mathcal{H}'$ is the result of the above-mentioned rewriting and $\mathcal{I}_\Delta^+$ is its right-hand side instance (where for every $v \in V$, $\mathcal{I}_\Delta^+(v) : R_v^\Delta \rightarrowtail G'_v$), then v_2 becomes the current version of the object and the new delta Δ is set to $(\mathcal{R}_\Delta^{-1}, \mathcal{I}_\Delta^+)$.

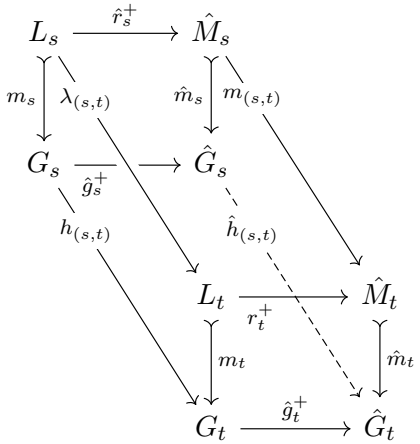
Merge versions. Let v_1 be the current version corresponding to a hierarchy $\mathcal{H}$, v_2 be another version corresponding to a hierarchy $\mathcal{H}'$ and the delta between v_1 and v_2 be given by $\Delta = (\mathcal{R}_\Delta, \mathcal{I}_\Delta)$. *The canonical merging rule hierarchy* can be constructed in the following way. For every individual hierarchy node we construct the canonical merging rule according to the framework described in Subsection 2.1.4. Let the back and front faces of the cube in Diagram 2.133 correspond to the pushouts defining pairs of merging rules corresponding to nodes $s, t \in V$ such that $(s,t) \in E$. We can apply the universal property of the pushouts and show that there exists a unique arrow $m_{(s,t)} : \hat{M}_s \rightarrow \hat{M}_t$ that makes the diagram commute. The merging rule hierarchy $\hat{\mathcal{R}}^+$ for $\mathcal{H}$ is, thus, given by rules $L_v \leftarrow Id_{L_v} - L_v - \hat{r}_v^+ \rightarrow \hat{M}_v$, for all $v \in V$, and rule

homomorphisms defined by arrows $(\lambda_{(s,t)}, \lambda_{(s,t)}, m_{(s,t)})$, for all $(s,t) \in E$. On the other hand, the merging rule hierarchy for $\mathcal{H}'$ is given by rules $R_v \leftarrow Id_{R_v} - R_v - \hat{r}_v^- \rightarrow \hat{M}_v$, for all $v \in V$, and rule homomorphisms defined by arrows $(\rho_{(s,t)}, \rho_{(s,t)}, m_{(s,t)})$, for all $(s,t) \in E$. Let $\hat{G}_s$ and $\hat{G}_t$ be the result of merging corresponding to the nodes s and t. By the universal property of pushouts there exists a unique arrow $\hat{h}_{(s,t)}$ that renders Diagram 2.134 commutative. Therefore, using such objects $\hat{G}_v$ for every $v \in V$ and homomorphisms $h_{(s,t)}$ for every $(s,t) \in E$, we can construct the hierarchy $\hat{\mathcal{H}}$ corresponding to the result of canonical merging of $\mathcal{H}$ and $\mathcal{H}'$.

Non-canonical merging can be specified using a hierarchy of objects $\mathcal{M}$ defined over the same skeleton as $\mathcal{H}$, and a pair of arrows $\bar{r}_v^+ : L_v \rightarrow M_v$ and $\bar{r}_v^- : R_v \rightarrow M_v$ such that $\bar{r}_v^+ \circ r_v^- = \bar{r}_v^- \circ r_v^+$ for every $v \in V$.



(2.133)



(2.134)

2.3 The ReGraph library

`ReGraph`[1] is a Python library implementing the mathematical theory, presented in the two previous sections, instantiated in the category of simple graphs with attributes $\mathbf{SimpGrph}_{attrs}$. It allows the user to create and manipulate graph objects equipped with dictionary attributes, create rewriting rules, apply them to graphs, construct graph hierarchies and perform rewriting and propagation in these hierarchies. Moreover, it provides tools for audit of updates performed in individual graph objects as well as hierarchies of graphs.

`ReGraph` provides the above-mentioned functionality based on two graph backends: in-memory graph objects provided by the `NetworkX`[2] library and persistent graphs provided by the `Neo4j`[3] graph database. Moreover, it is designed in a way that facilitates the addition of a new backend (for example, another graph database technology such as RDF-based Apache Jena[4], Blazegraph[5], Virtuoso[6] and so on).

[1] `https://github.com/Kappa-Dev/ReGraph`
[2] `https://networkx.github.io/`
[3] `https://neo4j.com/`
[4] `https://jena.apache.org`
[5] `https://blazegraph.com/`
[6] `https://virtuoso.openlinksw.com/`

ReGraph consists of the following principal components:

- Module `regraph.graphs` contains data structures for graphs. In particular, provides the abstract class `Graph` defining the interface for graph objects in `ReGraph` and implementing the graph transformation primitives of interest.
- Module `regraph.hierarchies` contains data structures for hierarchies. In particular, provides the abstract class `Hierarchy` defining the interface for hierarchy objects in `ReGraph` and implementing rewriting and propagation in hierarchies.
- Package `regraph.backends.networkx` provides a set of utilities for working with the `NetworkX`'s graph objects. It contains the `NXGraph` and `NXHierarchy` classes for in-memory representation of simple graphs with attributes and their hierarchies.
- Package `regraph.backends.neo4j` provides a set of utilities for working with the `Neo4j` graph database. It implements the `Neo4jGraph` class for persistent representation of simple graphs with attributes and the `Neo4jHierarchy` class for persistent representation of hierarchies of simple graphs.
- Module `rules` implements the `Rule` class for representation of SqPO rewriting rules in `ReGraph`.
- Module `audit` contains a set of data structures for audit of updates in simple graphs and graph hierarchies.

Dependencies of these components are schematically illustrated in Figure 2.17. To add a new backend for representation of graphs to `ReGraph`, one needs to implement corresponding concrete classes for graph and hierarchy objects that would inherit the abstract `Graph` and `Hierarchy` classes. The design of `ReGraph` ensures that the functionality required to implement such concrete classes is reduced to a relatively small set of basic primitive operations on graphs and graph hierarchies, while the high-level logic of rewriting, propagation and audit stays generic and is implemented by the existing `ReGraph` modules.

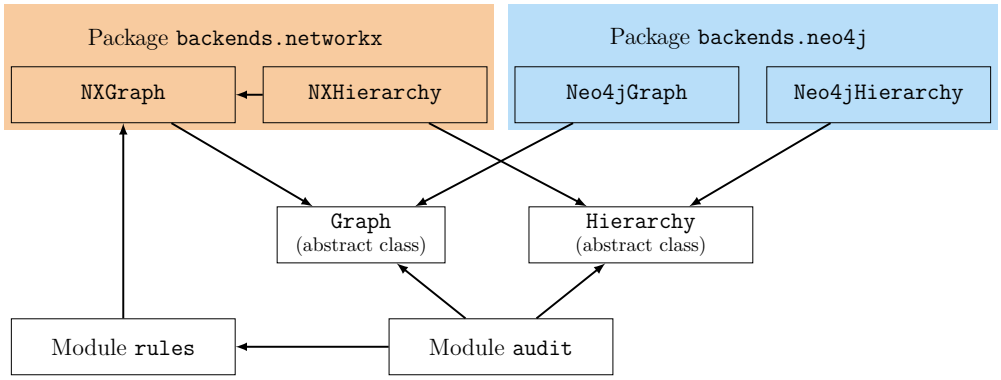


Figure 2.17: Dependencies between the main components of `ReGraph`.

The rest of this subsection provides some implementation and design details for different data structures and modules implemented in `ReGraph`.

2.3.1 Graphs and graph transformation in ReGraph

As we have previously mentioned, ReGraph allows us to create and manipulate simple graphs with attributes based on two graph backends: in-memory graphs provided by NetworkX and persistent Neo4j property graphs. Graph nodes possess unique identifiers that can be used to access them, create and manipulate edges defined by pairs of nodes, specify graph homomorphisms using maps of node identifiers. Any hashable Python object can be used as a node identifier.

In-memory graphs in ReGraph are instances of the NXGraph class. This class provides a set of methods for basic graph operations, e.g. addition, removal of nodes and edges, cloning and merging of nodes, etc. It also allows to store built-in attribute dictionary objects.

Similarly, the Neo4jGraph class provides a set of methods for basic graph operations on property graphs stored in an instance of the Neo4j database. To access the underlying database ReGraph uses the Python driver for Neo4j, namely the class neo4j.GraphDatabase. Every object of the Neo4jGraph class is equipped with the attribute Neo4jGraph._driver providing an 'access point' to the database. The current Neo4j implementation does not allow to store different graphs in a single database, therefore, every database instance is a single PG. This means that two different Neo4jGraph objects accessing the same database instance operate on the same property graph. However, it is essential for us to be able to accommodate multiple graphs, and to do so, we exploit the PG data model. Recall that this model allows the storing of data as a collection of nodes and relationships. Nodes of PGs can be assigned with labels (allowing, for example, to group nodes into different sets) and every relationship in a property graph is assigned with exactly one type (more detailed discussion of PGs can be found in Chapter 3). ReGraph uses PG node labels for defining disjoint *namespaces* for identifiers of graph elements, i.e. every graph object has two attributes, Neo4jGraph._node_label and Neo4jGraph._edge_label, defining a subset of nodes and edges considered as nodes and edges of the current graph object. All the interface methods execute respective Cypher queries on the database instance accessed by the Neo4jGraph._driver attribute.

Example 2.3.1. *The following two listings illustrate how simple graphs can be created in ReGraph. The code on the left creates an empty in-memory graph object, adds some nodes and edges to it and performs cloning of a node. The code on the right shows how the same thing can be done using persistent graph objects implemented in the Neo4jGraph class. The graph created in the corresponding Neo4j database manipulates nodes with the label graphNode and relationships of the type graphEdge.*

```
from regraph import NXGraph

graph = NXGraph()
graph.add_nodes_from(
    ["a", "b", "c"])
graph.add_edges_from([
    ("a", "b"),
    ("b", "c")])
graph.clone_node("a", "a_clone")
```

```
from regraph import Neo4jGraph

graph = Neo4jGraph(
    uri="bolt://localhost:7687",
    user="neo4j", password="neo4j",
    node_label="graphNode",
    edge_label="graphEdge")
graph.add_nodes_from(["a", "b", "c"])
graph.add_edges_from([
    ("a", "b"), ("b", "c")])
graph.clone_node("a", "a_clone")
```

Encoding attribute sets

We would like to equip both our in-memory and persistent graphs with node/edge attributes. By definition, the attributes are structured as dictionaries (formally defined in A.1), i.e. key-value pairs with set-like values. The module `regraph.attribute_sets` provides a set of classes implementing attribute values. The `AttributeSet` class represents an interface for any such class of attribute values. It specifies a collection of elementary set-theoretic operations that can be performed on sets, e.g. union, intersection, difference, inclusion and equality tests, etc. It also manages the export of attribute values from their JSON-representation. The package contains the following structures for attribute sets inheriting the `AttributeSet` class:

- `FiniteSet`, a wrapper around the standard Python `set` datatype;
- `RegexSet`, a class implementing a set of strings recognized by an encapsulated regular expression;
- `IntegerSet` a class implementing a possibly infinite set of integers defined by a sequence of integer intervals;
- `EmptySet`, a class representing an empty set;
- `UniversalSet`, a class representing the universal set, any instance of `AttributeSet` is a subset of the universal set.

As we have previously mentioned `NetworkX` allows us to store arbitrary objects on the nodes or edges of a graph object. Therefore, our in-memory graphs with attributes (instances of `NXGraph`) are simple wrappers around `NetworkX.DiGraph` with Python dictionaries attached to the nodes and edges; the values of these dictionaries are instances of `AttributeSet`.

Implementation of set attributes for persistent graphs, on the other hand, requires an encoding effort. According to the property graph data model, both nodes and relationships can be equipped with properties, an internal structure allowing to represent a set of key-value pairs. `Neo4j` graphs allow to store as property values numbers (integers, floats), strings, booleans as well as some specialized types for storing spacial and temporal data (Point, Date, Time). Moreover, homogeneous lists of simple types can be accommodated on nodes and relationships of property graphs. We use these lists to represent sets of values of our dictionary attributes This, of course, constraints our `Neo4j`-based implementation to the use of homogeneous sets of values, therefore, if heterogeneous values are encountered in a given set, `ReGraph` automatically casts all the values of the set into strings. We also reserve some string literals as key-words that symbolically represent some infinite sets, e.g. `IntegerSet` for the set of all integers, `StringSet` for the set of all strings, `UniversalSet`, etc. Currently, the `Neo4j`-backend of `ReGraph` does not allow one to create regex-defined sets of strings or integer sets defined by intervals.

Example 2.3.2. *The following listing illustrates how the interface of `NXGraph` and `Neo4jGraph` can be used to add attributes to nodes and edges in ReGraph. Observe that the attribute with the key age of the node a is a symbolic set representing an interval from 18 to infinity. Adding attributes to elements of Neo4jGraph objects can be performed in a similar way. Note, however, that interval-defined sets of integers are not implemented for Neo4jGraph.*

```
import regraph.attribute_sets as
                              ats
from math import inf
from regraph import NXGraph

graph = NXGraph()
graph.add_node(
  "a", {
      "name": {"Bob"},
      "age": ats.IntegerSet(
          {(18, inf)})
   })
graph.add_node("b")
graph.add_node_attrs(
    "b", {
      "name": {"Alice"},
      "age": {19}
   })
graph.add_edge(
    "a", "b",
    {"type": {"friends"}})
```

```
from math import inf
from regraph import primitives
from regraph import Neo4jGraph

graph = Neo4jGraph(
  uri="bolt://localhost:7687",
    user="neo4j",
    password="neo4j")
graph.add_node(
  "a", {
      "name": {"Bob"},
      "age": {18})
   })
graph.add_node("b")
graph.add_node_attrs(
    "b", {
      "name": {"Alice"},
      "age": {19}
   })
graph.add_edge(
    "a", "b",
    {"type": {"friends"}})
```

2.3.2 Rewriting rules in ReGraph

SqPO rewriting rules in `ReGraph` can be expressed declaratively, similarly to their mathematical definition presented previously in this section, using the class `regraph.Rule` (in the rest of the section—`Rule`). Essentially, `Rule` encapsulates three small instances of `NXGraph`: `Rule.lhs` representing the left-hand side L, `Rule.p`—the interface P and `Rule.rhs`—the right-hand side R of a rule $L \xleftarrow{r^-} P \xrightarrow{r^+} R$. The two homomorphisms r^- and r^+ are represented with the attributes `Rule.p_lhs` and `Rule.p_rhs` respectively.

Example 2.3.3. *The following listing illustrates how SqPO rewriting rules can be created in `ReGraph`. The created rule clones the node `1` from the left-hand side (the node `1_clone` corresponds to the created clone in the interface), removes the edge from `2` to `3`, adds a new node (identified by `new_node` in the right-hand side) and connects it with an edge to the node `1`.*

```
from regraph import Rule, NXGraph

# Define the left-hand side of the rule
lhs = NXGraph()
lhs.add_nodes_from([1, 2, 3])
lhs.add_edges_from([(1, 2), (2, 3)])
# Define the interface of the rule
p = NXGraph()
```

```
p.add_nodes_from([1, "1_clone", 2, 3])
p.add_edges_from([(1, 2), ("1_clone", 2)])
# Define the right-hand side of the rule
rhs = NXGraph()
rhs.add_nodes_from([1, "1_clone", 2, 3, "new_node"])
rhs.add_edges_from([(1, 2), ("1_clone", 2), ("new_node", 1)])
# Define rule homomorphisms
p_lhs = {1: 1, "1_clone": 1, 2: 2, 3: 3}
p_rhs = {1: 1, "1_clone": "1_clone", 2: 2, 3: 3}

rule = Rule(p, lhs, rhs, p_lhs)
```

Rules in ReGraph can also be created using the `Rule.from_transform` method. This method allows the initialization of an identity rule from a pattern (a rule that does not specify any changes). Such an identity rule can be then modified procedurally, i.e. by specifying a sequence of primitive graph operations on the left-hand side of the rule (see Example 2.3.4).

Example 2.3.4. *The following listing illustrates how the rule from Example 2.3.3 can be created by initializing a rule object from a pattern and injecting a set of primitive operations.*

```
pattern = NXGraph()
pattern.add_nodes_from([1, 2, 3])
pattern.add_edges_from([(1, 2), (2, 3)])

rule = Rule.from_transform(pattern)
rule.inject_clone_node(1)
rule.inject_remove_edge(2, 3)
rule.inject_add_node("new_node")
rule.inject_add_edge("new_node", 1)
```

ReGraph provides means for finding matches of a pattern in a graph using the `find_matching` method of `NXGraph` and `Neo4jGraph`. As input, this function takes a graph object and a pattern graph. Optionally, the user can provide a collection of nodes specifying the subgraph of the original graph, where the search should be performed. ReGraph finds all matches of the pattern by solving the *subgraph matching problem*. The function returns a list of all such matches defined by maps from the nodes of the pattern to the nodes of the input graph such that (1) edges are preserved and (2) the attribute dictionary of a pattern node is a subdictionary of its image in the graph. For in-memory graphs ReGraph uses the `networkx.isomorphism.DiGraphMatcher` class, which provides a method for finding subgraph isomorphisms based on the VF2 algorithm [18]. In the case of persistent graphs such a pattern matching task corresponds to an ordinary match query on the respective database. Note that, in our setting, a matching is always given by an injective map of nodes which means that the nodes and edges of our patterns are always distinct graph objects. Such semantics is also known as isomorphism-based semantics in the literature of graph query languages [4], where both node and edge variables must be mapped one-to-one.

An instance of the `Rule` class can be applied to a graph using the `rewrite` method of

NXGraph and Neo4jGraph. This method takes as input a rule object and a dictionary specifying a match of the left-hand side of the rule in the graph object.

Example 2.3.5. *The following code illustrates the use of* ***primitives.find_matching*** *combined with the* ***Rule.apply_to*** *method for performing graph rewriting. We assume here that the objects* ***rule*** *and* ***graph*** *have been initialized as above, and that the list of matches of the left-hand side of the rule given by the variable* ***instances*** *is non-empty.*

```
instances = graph.find_matching(rule.lhs)
rhs_instance = graph.rewrite(rule, instances[0])
```

ReGraph also provides the method Rule.refine that allows the refinement of an arbitrary SqPO rewriting rule to its reversible version, given some graph and an instance of the rule in this graph. This method implements the procedure described in 2.1.2.

2.3.3 Hierarchies in ReGraph

ReGraph provides two data structures for working with hierarchies of simple graphs with attributes: NXHierarchy based on in-memory graphs and Neo4jHierarchy based on persistent Neo4j PGs; both inheriting the generic Hierarchy class.

Graph nodes of hierarchies possess unique identifiers (as before, any hashable Python objects) and encapsulate graph objects. Moreover, they can be equipped with attribute dictionaries (useful when associating some meta-data with a graph in a hierarchy). Several methods for adding new graph objects to the hierarchy are available in the hierarchy interface, e.g. add_graph adds to the hierarchy the graph object received as input, add_graph_from_data adds a graph with provided nodes and edges, add_graph_from_json adds the graph from its JSON-representation, add_empty_graph creates a new empty graph object and adds it. New graph homomorphisms can be added using the method add_typing, which verifies that the addition of the edge between the source and the target does not create a cycle or produce paths that do not commute with some already existing paths, i.e. that acyclicity and the consistency properties of graph hierarchies are preserved.

Example 2.3.6. *The following listing illustrates how in-memory graph hierarchies can be created in* ***ReGraph****. The created hierarchy consists of two graphs G and T related by a homomorphism.*

```
from regraph import NXGraph, NXHierarchy

# Create graph objects
t = NXGraph()
t.add_nodes_from(["Person", "City"])
t.add_edges_from(
  [("Person", "City"),
   ("Person", "Person")])
g = NXGraph()
g.add_nodes_from(
  ["Alice", "Bob", "Lyon", "Paris"])
g.add_edges_from(
```

```
    [("Alice", "Bob"),
     ("Alice", "Lyon"),
     ("Bob", "Paris")])

# Create a graph hierarchy
hierarchy = NXHierarchy()
hierarchy.add_graph(
    "T", t, attrs={
        "desc": "Meta-model"
    })
hierarchy.add_graph(
    "G", g, attrs={
        "desc": "People living in cities database"
    })
hierarchy.add_typing(
    "G", "T", {
      "Alice": "Person",
      "Bob": "Person",
      "Lyon": "City",
      "Paris": "City"
    },
    attrs={
        "desc": "Typing of entities in the meta-model"
     })
```

A set of methods of the `Hierarchy` class provides means for rewriting and propagation in the hierarchy. The method `find_matching` allows to find matches of a pattern in a graph of the hierarchy. The method `rewrite` applies a rule through the specified instance in a graph of the hierarchy. It takes as input two additional parameters `p_typing` and `rhs_typing` corresponding to the binary relations for controlling propagation from 2.2.4 and performs all the necessary propagations. These parameters are optional and, if not specified, the rewriting is propagated canonically to all the ancestors and descendants of the rewritten graph, i.e. all the instances of the cloned nodes are cloned and all the newly added nodes acquire new typing nodes.

Example 2.3.7. *The following listing illustrates how in-memory graph hierarchies can be rewritten in* `ReGraph`*. We create a rewriting rule that adds a new node* `Eric` *and connects it with an edge to* `Alice` *in* G *in the hierarchy previously defined in Example 2.3.6. The propagation of this rewriting is controlled by the parameter* `rhs_typing` *that specifies that the type of the newly added node* `Eric` *in* T *is* `Person`*.*

```
from regraph import Rule, NXGraph

# Create a rewriting rule
pattern = NXGraph()
pattern.add_nodes_from(["x", "y"])
pattern.add_edges_from([("x", "y")])

```

```
rule = Rule.from_transform(pattern)
rule.inject_add_node("Eric")
rule.inject_add_edge("Eric", "x")

# Apply the rule to the graph G
instance = {"x": "Alice", "y": "Bob"}
hierarchy.rewrite(
  "G", rule, instance,
  rhs_typing={"T": {"Eric": "Person"}})
```

The method `get_rule_hierarchy` allows the user to find a rule hierarchy corresponding to the application of a rule to a particular graph in the hierarchy (recall the discussion on rule hierarchies in 2.2.7) together with the instances of the rules in this hierarchy. Rule hierarchies are represented with Python dictionaries containing two keys, `rules` and `rule_homomorphisms`. The value of `rules` is a dictionary with identifiers of the hierarchy graphs as keys and computed propagation rules as values (for the origin of rewriting this values is the initial rewriting rule). The value of `rule_homomorphisms` is a dictionary with hierarchy edges as keys and a triple of homomorphisms as values. These triples represent homomorphisms for the left-hand side, the interface and the right-hand side of the respective rules in the rule hierarchy. A rule hierarchy can be applied to the hierarchy using the `apply_rule_hierarchy` method. We can also refine rule hierarchies to their reversible versions using `refine_rule_hierarchy` method.

Example 2.3.8. *The following listing illustrates how, instead of directly applying a rule in a hierarchy and performing all the necessary propagations, we can first compute the rule hierarchy corresponding to the rewriting and propagations and then apply it. This is useful if we need to examine the rule hierarchy before applying it, for example, when using audit trails.*

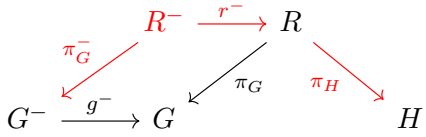
```
from regraph import Rule, NXGraph

# Create a rewriting rule
pattern = NXGraph()
pattern.add_nodes_from(
   ["x", "y"])
pattern.add_edges_from(
   [("x", "y")])
rule = Rule.from_transform(pattern)
rule.inject_add_node("Eric")
rule.inject_add_edge("Eric", "x")

# Get the rule hierarchy
instance = {"x": "Alice", "y": "Bob"}
rule_hierarchy = hierarchy.get_rule_hierarchy(
  "G", rule, instance,
  rhs_typing={"T": {"Eric": "Person"}})
# Apply the rule hierarchy
hierarchy.apply_rule_hierarchy(rule_hierarchy, instances)
```

Relations as undirected edges in a hierarchy

As well as graph homomorphisms, our library is able to represent binary relations on graphs in a hierarchy. This functionality of `ReGraph` goes slightly beyond the theory described in this section; however, the idea behind it is very simple. We accommodate relations between graphs as separate *undirected* edges of the hierarchy. We do not impose any conditions on these edges, except that only one relation between a given pair of nodes is allowed.

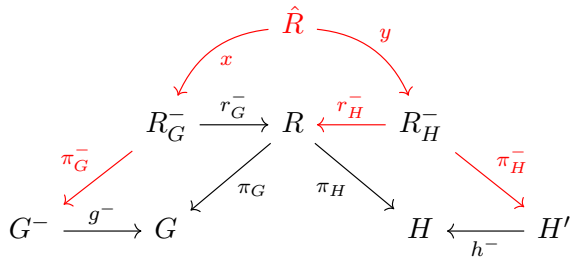
Given two graphs G and H, a binary relation between G and H is defined as a set of pairs of nodes $R \subseteq V_G \times V_H$. Let π_G and π_H be two projection functions that project elements in R into V_G and V_H respectively. It is useful to see a relation R as a graph with no edges. Then we can construct a span $G \leftarrow \pi_G - R - \pi_H \rightarrow H$ in the category $\mathbf{SimpGrph}_{attrs}$.

Our question is now how do we propagate the updates made to the objects G and H to the relation R? Seeing relations between graphs as spans allows us to apply the previously presented propagation framework without any modifications. In the current implementation of `ReGraph`, we cannot control propagation to such objects R, therefore the propagation is always performed canonically. For example, if a graph G is transformed with a restrictive update $g^- : G \leftarrow G^-$ (as the origin of rewriting or as the result of a propagation) we can construct the pullback from g^- and π_G and obtain the span $G^- \leftarrow \pi_G^- - R^- - r^- \rightarrow R$ as in Diagram 2.135. Then we can set our relation R to the set of pairs defined by the nodes of R^- with the projection functions π_G^- and $\pi_H \circ r^-$ projecting onto the nodes of G^- and H respectively (denoted with red in the diagram).



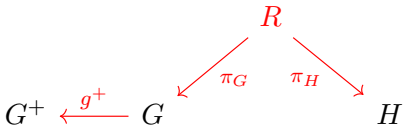
(2.135)

In the case when both G and H were updated with restrictive updates $g^- : G \leftarrow G^-$ and $h^- : H \leftarrow H^-$ respectively, we perform a similar procedure. We first find the pullbacks from the pair of g^- and π_G, and the pair of h^- and π_H and obtain two spans $G^- \leftarrow \pi_G^- - R_H^- - r_G^- \rightarrow \rightarrow R$ and $H^- \leftarrow \pi_H^- - R_H^- \overset{r_H^-}{\rightarrow} R$ respectively, as in Diagram 2.136. We then construct the pullback from r_G^- and r_H^-, and obtain the span $R_G^- \leftarrow x - \hat{R} \overset{y}{\rightarrow} R_H^-$. The updated relation corresponds to the set of pairs defined by the nodes of $\hat{R}$ with the projection functions $\pi_G^- \circ x$ and $\pi_H^- \circ y$, projecting onto the nodes of G^- and H^- respectively (denoted with red in the diagram).



(2.136)

Finally, if a graph G is updated with an expansive update $g^+ : G \rightarrow G^+$, we can modify R according to the updated projection onto G^+ given by $g^+ \circ \pi_G$ (denoted in Diagram 2.137 with red).



(2.137)

ReGraph stores graph relations as a separate type of edges and provides a basic interface for access and modification of these edges. Similarly to homomorphism edges, relation edges can be equipped with attributes.

Example 2.3.9. *The following listing illustrates how relations between graphs can be added in ReGraph. In this example we use the hierarchy defined in Example 2.3.6. We first add another graph H that contains knowledge of some online accounts and, second, relate its nodes with the nodes of G containing knowledge of people living in cities.*

```
# Add a new graph H to the hierarchy
h = NXGraph()
h.add_nodes_from(
    [
       "alice93@gmail.com",
       "a.dubuffet",
       "bob.evans" ])
hierarchy.add_graph("H", h)

# Add relation between G and H
hierarchy.add_relation(
    "G", "H", {
        "Alice": {"alice93@gmail.com", "a.dubuffet"},
        "Bob": {"bob.evans"}
     },
     attrs={
        "desc": "Relation between people and their accounts"})
```

In-memory graph hierarchies

The class NXHierarchy implements in-memory hierarchies in ReGraph. Apart from the generic Hierarchy class, it also inherits the NXGraph class, i.e. every instance of such a graph hierarchy is a graph and has the interface of directed graphs implemented in NXGraph. Hierarchy nodes are represented with Python dictionaries of a fixed shape, i.e. they have two records: the graph object is associated to the key graph and the attributes of the node with the key attrs. Edges of the hierarchy encapsulate maps of nodes encoding graph homomorphisms and some associated attributes. Similarly to hierarchy nodes, edges are dictionaries with two records: mapping is associated to a Python dictionary encoding a mapping of nodes from the source graph to the target's nodes attrs dictionary containing attributes of the new edge.

Another additional feature of ReGraph, not described in the theoretical part of this chapter and implemented for in-memory NXHierarchy objects, allows the user to add rewriting rules to the hierarchy. Similarly to graphs, rules are stored as nodes of the hierarchy and can be

equipped with attributes. Accommodation of rewriting rules in hierarchies is handy when the co-evolution of these rules and the graphs that type them is desirable.

Rule nodes in ReGraph can be equipped only with out-going edges and target nodes of these edges are required to represent graphs, i.e. we can only type rules by some graphs in our hierarchy. The rule typing edges encapsulate two homomorphisms: typing of the left-hand side and the right-hand side of the source rule by the target graph of the edge (typing of the interface is implicitly given by the two above-mentioned homomorphisms). Propagation of rewriting to rules is performed according to our backward propagation framework described in the previous parts of this chapter.

Example 2.3.10. *The following listing illustrates how rules can be added to ReGraph hierarchies.*

```
# Create a rewriting rule
pattern = NXGraph()
pattern.add_nodes_from(
  ["x", "y"])
pattern.add_edges_from(
   [("x", "y")])
rule = Rule.from_transform(pattern)
rule.inject_add_edge("y", "x")

# Add the rule to the hierarchy and type it by T
hierarchy.add_rule(
  "R", rule,
  {"desc": "Make a friendship bidirectional"})
hierarchy.add_rule_typing(
  "R", "T", {
    "lhs_typing": {"x": "Person", "y": "Person"},
    "rhs_typing": {"x": "Person", "y": "Person"}
  })
```

Persistent graph hierarchies

Persistent graph hierarchies are implemented in the Neo4jHierarchy class. Because both NXHierarchy and Neo4jHierarchy inherit the same abstract class Hierarchy, their interfaces are mostly shared (currently, the functionality related to the accommodation of rewriting rules is not implemented for persistent graphs). However, interesting points of dissimilarity between the persistent and the in-memory implementation of hierarchies in ReGraph consists in the way the skeleton of a hierarchy is accommodated within the underlying graph database together with the content of the hierarchy.

As we have previously mentioned, Neo4j does not allow to store multiple property graphs in the same database, therefore ReGraph implements a mechanism that stores both hierarchy skeleton, its graphs, homomorphisms and relations within a single property graph. Therefore, apart from the credentials necessary to connect to the database, the constructor of the Neo4jHierarchy class takes as the input the following arguments:

- `graph_label` defines the label used to denote the nodes representing the nodes of the hierarchy skeleton;
- `typing_label` defines the relationship type used to denote the relationships representing the homomorphism (typing) edges of the hierarchy skeleton;
- `relation_label`—the relationship type representing the binary relation edges of the hierarchy skeleton;
- `graph_edge_label`—the relationship type representing the edges inside of the graphs belonging to the hierarchy;
- `graph_typing_label`—the relationship type for edges encoding maps of nodes that define homomorphisms in the hierarchy;
- `graph_relation_label`—the relationship type for edges encoding relations between nodes that define binary relations in the hierarchy;

Then, the skeleton graph of the hierarchy is given by the set of nodes labeled with the value of `graph_label` and the set of relationships with the types `typing_label` and `relation_label` in the graph database. Every individual graph G is then defined by the set of nodes labeled with the identifier of this graph in the hierarchy, while the edges are defined by all the database relationships with the label given by the value of `graph_edge_label` that connect nodes from the corresponding set of graph nodes V_G. A homomorphisms $h : G \to T$ situated in such a graph hierarchy is then defined by all the database relationships labeled by the value `graph_typing_label` and whose source node is labeled with the identifier of G and target the identifier of T. The same applies for binary graph relations accommodated in the hierarchy: they are obtained by the set of relationships labeled with `graph_relation_label`.

This encoding (see schematic example in Figure 2.18) does not only allow to store persistently the structure of the hierarchy, the hierarchy's homomorphisms and its relations, it also allows to exploit the querying capabilities of the graph database for performing propagation in such hierarchies. Having accommodated homomorphisms and binary relations with native relationships in the database allows us to design concise and intuitive queries performing propagation of rewriting. A more detailed discussion of these queries can be found in [10].

2.3.4 Audit trails in `ReGraph`

To design and implement an audit trail for the presented KR system we have adopted an approach similar to modern VC systems and adapted the main notions of VC to our formalism of graphs with attributes and hierarchies of graphs.

VC views an update of the controlled object as an atomic operation called *commit*. Commits are stored in a structure usually called a *revision graph*, whose nodes are commits and whose edges connect successive commits. The states of the object at the times of different commits are stored using delta compression. A delta is a symbolic representation of the change in the object state from one commit to its successor. Delta compression allows the system to store only the current version of the object, while all previous versions can be computed using the deltas encapsulated within the revision graph. The commit that has produced the current state of the object is usually called the *head commit*. Parallel versions of an object are maintained

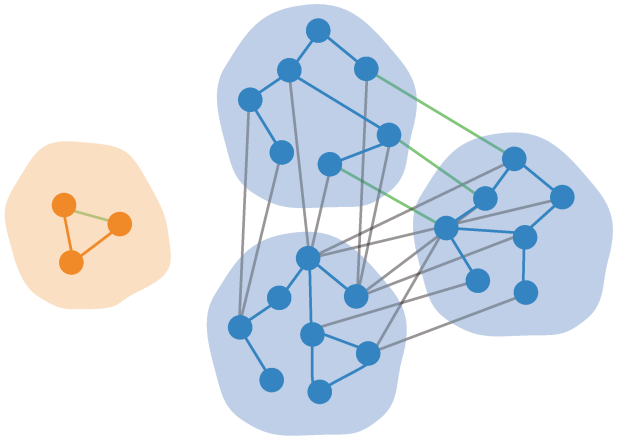


Figure 2.18: Schematic illustration of the hierarchy encoding in the Neo4j database. Orange area represents the skeleton of the hierarchy, blue areas represent the individual graphs situated in the hierarchy. Gray edges between the nodes of the individual graphs encode homomorphisms between these graphs, while green edges encode relations between the nodes of different graphs.

using a branching system, i.e. updates of these versions are represented by 'parallel' branches of the revision graph.

The `regraph.audit` module implements the audit trail related functionality in `ReGraph`. It provides several classes for maintaining audit trail structures for both graphs and graph hierarchies. The class `Versioning` represents an abstract class for a generic audit trail. `ReGraph` represents the *revision graph* as an object of the `networkx.DiGraph` class (corresponding to the attribute `Versioning._revision_graph`). Every node of the revision graph corresponds to a commit and is identified by a unique identifier (these identifiers are generated automatically using the `uuid`[7] Python library). In addition, the nodes of the revision graph are equipped with the following attributes: `branch` specifies the name of the branch where the commit was performed, `time` provides the time of the commit and `message` contains the user-provided commit message. The edges of the revision graph connect pairs of successive commits and encapsulate the deltas corresponding to these commits. Moreover, the attribute `Versioning._heads` stores a dictionary whose keys are names of the branches and whose values are corresponding head commits (recall that head commits define parallel versions of the audited object). To store the current head, `Vesioning` uses its attribute `_current_branch`, the current head commit is then obtained by finding the commit corresponding to the key `_current_branch` in the dictionary `_heads`. To facilitate switching between different branches `ReGraph` stores the pre-computed deltas between different heads (more precisely, the directed deltas from the current head to all other heads).

The `VersionedGraph` and `VersionedHierarchy` are concrete implementations of the audit trail for simple graphs with attributes and hierarchies of simple graphs with attributes. They inherit `Versioning` and are additionally able to encapsulate graph objects and hierarchy objects respectively. In addition, these classes provide the `rewrite` method, through which all the updates of the underlying objects should be performed in order to be a part of our audit trail.

Every SqPO rewrite of a graph encapsulated in an instance of `VersionedGraph` is expressed as a delta that consists of the applied rule together with the left-hand side and the right-hand

[7] `https://docs.python.org/3/library/uuid.html`

side instances of this rule. The instances are represented with simple Python dictionaries, whose keys are node identifiers for the nodes of the corresponding patterns and whose values are node identifiers of the target graphs at the time of rewriting. Values of these dictionaries are used to compute overlaps between consecutively applied rules (as in Subsection 2.1.3).

SqPO rewriting in a hierarchy encapsulated in an instance of `VersionedHierarchy` is expressed as a delta that consists of the applied rule hierarchy together with the left-hand side and the right-hand side instances of this hierarchy. The instances are represented with Python dictionaries whose keys are identifiers of graphs in the hierarchy and whose values are dictionaries representing instances of individual rules in the corresponding graphs. Values of these dictionaries are used to compute hierarchy overlaps between consecutively applied rule hierarchies (as in Subsection 2.2.9).

The `Versioning` class implements the methods corresponding to typical VC operations: it provides methods for the operations of commit (`commit`), branch switching (`switch_branch`), branching (`branch`), merge of branches (`merge`) and rollback (`rollback`). It also allows the user to access to the list of branches (`branches`) and to the current branch (`current_branch`) corresponding to the current version of the audited object. The VC operations are implemented as follows:

Commit. A commit operation corresponding to some transformation adds a new node to the revision graph, connects it with an edge to the current head, adds a delta corresponding to a transformation to the newly created edge. Example of a commit operation is given in Figure 2.19.

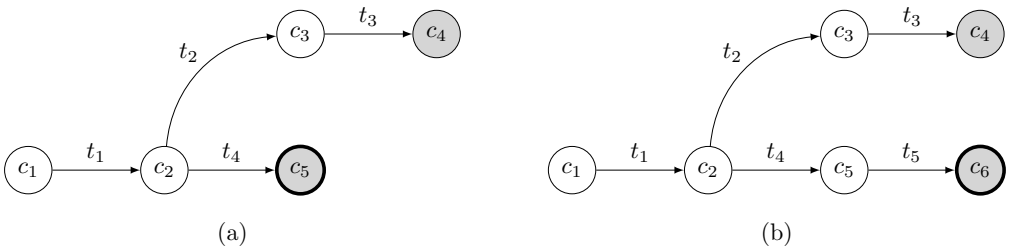


Figure 2.19: Example of an audit trail before (a) and after (b) the commit operation corresponding to some transformation whose delta is denoted with t_5. Head commits are represented with grey nodes, the current head is highlighted with a bold node.

Switch branch. The operation of branch switching applies the delta specifying the transformation of the current version of the object to the branched version (to which the switching is performed). Then the value of the current head is updated accordingly. Figure 2.20 presents a small example of branch switching.

Branch. To branch from the current head we need to perform a commit of the identity transformation specified by some delta and add this commit to the set of heads while preserving the current head. An example of this operation can be found in Figure 2.21.

Merge. Merging of the branch defined by some head (distinct from the current head) into the current branch can be performed by specifying two merging deltas: one that defines a merging

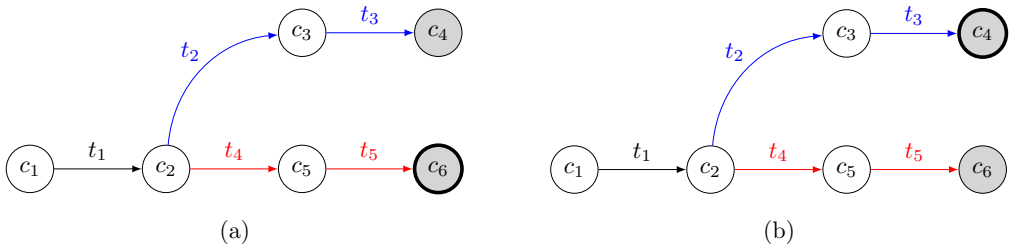


Figure 2.20: Example of audit trails before (a) and after (b) branch switching. Head commits are depicted with gray nodes and current heads are highlighted with a bold node. Different parallel branches are denoted by red and blue edge colors.

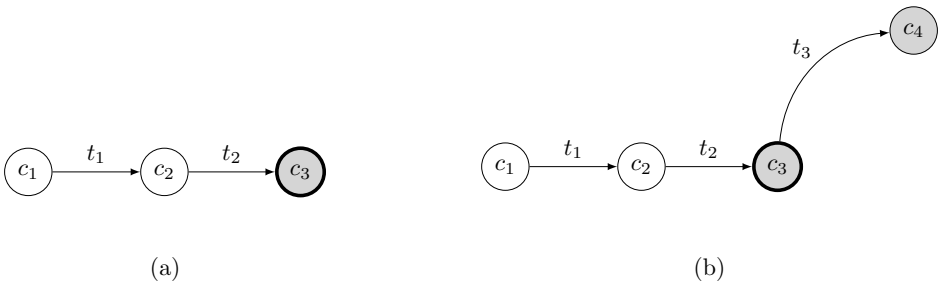


Figure 2.21: Example of audit trails before (a) and after (b) branching. Head commits are depicted with gray nodes and current heads are highlighted with a bold node. The newly added delta correpospodning to the identity tranformation of the underlying object is denoted by t_3.

transformation for the current version of the object and one that defines such a transformation for the other branch. This operation adds a new commit to the revision graph, connects it with edges to the current and the other head and associates the two merging deltas to these edges. Figure 2.22 presents a small example of branch merging.

Rollback. The operation of rollback computes the composition of deltas along the path from the current head to the rollback commit in the revision graph. Then, the update of the revision graph and the set of heads is performed as follows. Let c be a rollback commit and $\mathcal{P}(c, H)$ be the set of all paths from the commit c to every head commit $h \in H$. These paths define the set of nodes and edges to remove from the revision graph. Any head whose commit is associated with a removed node is, thus, removed. Then, all the commits whose successor nodes are removed become new heads. Let us consider a small example of a rollback operation in Figure 2.23

The implemented VC operations described above rely on a set of *abstract* methods that, therefore, make our class abstract and that must be implemented in every *concrete* audit trail. These methods include:

- `_invert_delta`, a static method that inverts the input delta;
- `_apply_delta` applies the input delta to the current object;
- `_compose_deltas`, a static method that finds the delta corresponding to the composition of the two input deltas;

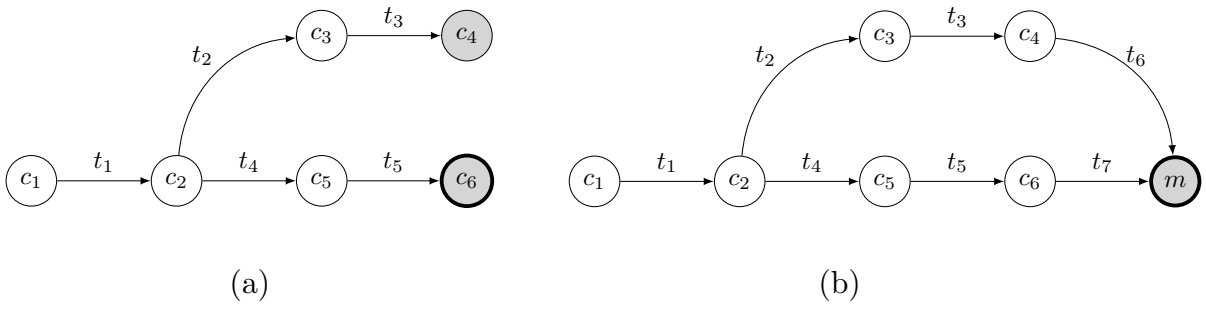


Figure 2.22: Example of an audit trail before (a) and after (b) the merge of branches defined by heads c_4 and c_6. Head commits are depicted with gray nodes and current heads are highlighted with bold node lines. The two merging deltas are denoted by t_6 and t_7.

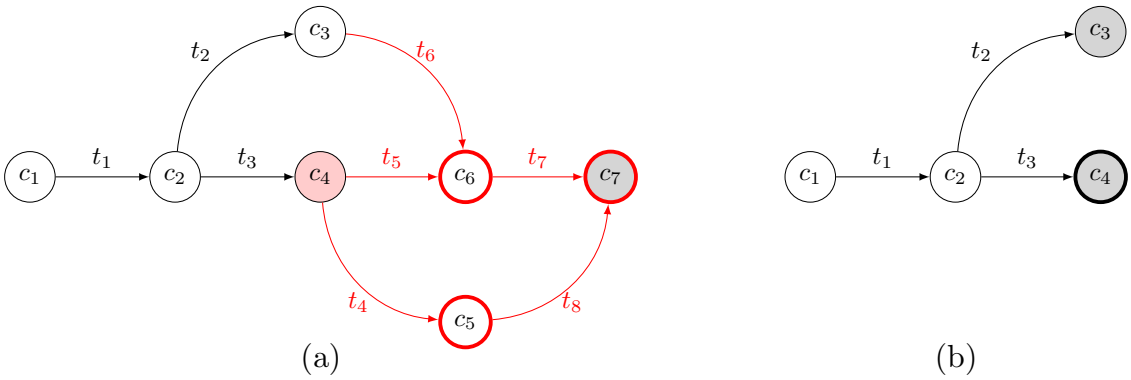


Figure 2.23: Example of an audit trail before (a) and after (b) a rollback operation. Head commits are represented with grey nodes, the rollback commit is denoted by a light red node, the removed nodes and edges of the revision graph are highlighted with red.

- `_merge_into_current_branch` computes merging deltas for the current head and the head of the input branch, then applies this delta to the current object;
- `_delta_to_json`, a static method that converts the input delta to its JSON representation;
- `_delta_from_json`, a static method that converts the input JSON representation to the corresponding delta object.

Example 2.3.11. *The following listing illustrates how a graph audit trail can be created in `ReGraph`. This example also illustrates how we can branch, rewrite objects through the interfaces of the audit trail and switch branches.*

```
1 from regraph import NXGraph, VersionedGraph
2
3 # Create a graph object
4 graph = NXGraph()
5 graph.add_nodes_from(["circle", "square"])
6 graph.add_edge("circle", "square")
```

```
7
8  # Create an audit trail for this object, initial branch
9  # is called "master"
10 g = VersionedGraph(graph, init_branch="master")
11
12 # Create the new branch "test"
13 g.branch("test")
14
15 # Create a rewriting rule
16 pattern = NXGraph()
17 pattern.add_node("square")
18 rule = Rule.from_transform(pattern)
19 rule.inject_clone_node("square")
20
21 # Apply this rule through the interface of VersionedGraph
22 g.rewrite(
23   rule,
24   instance={"square": "square"},
25   message="Clone the square node")
26
27 # Switch back to "master", after switching we obtain
28 # the version of the graph before cloning of the square
29 g.switch_branch("master")
```

Finally, note that our audit trail objects are not persistent, but *in-memory*. `Versioning` provides two methods, `to_json` and `from_json`, that can be used for the (de-)serialization of audit trails in the JSON format.

2.4 Discussion and conclusion

In this chapter, we have presented a KR system based on hierarchies of graphs and their homomorphisms. The described hierarchical structure can be, in general, instantiated in any category with appropriate structure (with pushouts and pullback complements over monos). We have developed a framework which provides means for updating individual objects situated in a hierarchy and propagating necessary changes to other objects.

In particular, we have described the frameworks for backward and forward propagation that apply in the situation when one of the two homomorphic objects in $G -h\to T$ is updated. Restrictive updates of T (given by removal and cloning of elements) induce backward propagation that may result in an update of G. Inversely, expansive updates of G (given by addition and merging) induce forward propagation that may perform an update of T. Such propagation varies over a range of updates that make it possible to unambiguously restore the homomorphism between the two updated objects. These updates are controlled by means of rule factorizations and clean-up arrows. Interestingly, rule factorizations allow controlling propagation by splitting the original rewriting into two phases, the first of which is called strict phase and performs a portion of updates that does not 'break' the original homomorphism. The second phase is called

canonical propagation and performs necessary propagation updates in some canonical fashion, i.e. using categorical constructions with universal properties.

The type respecting rewriting and canonical coupled transformations presented in the introductory chapter (Section 1.1) fall into the range of updates specified by our propagation framework. For example, given some forward rule factorization $L \to L' \to L^+$ for G, $L' \cong L^+$ gives us a type respecting rewriting $G \rightsquigarrow G'$ with $G' \to T$ as in [19]. On the other hand, in a backward factorization $L \leftarrow L' \leftarrow L^-$ or a forward factorization $L \to L' \to L^+$, $L \cong L'$ gives us a canonical coupled transformation of G and T similar to the one described in [66] for the DPO approach.

We have further used our backward and forward propagation framework to derive a procedure that allows rewriting individual objects in a general hierarchy with an arbitrary SqPO rule in a way that preserves hierarchy's structure and consistency. We have also introduced the notion of a rule hierarchy that generalizes SqPO rewriting to hierarchies of objects and studied the conditions under which an arbitrary rule hierarchy can be applied to the corresponding hierarchy of objects. We have presented the construction of a rule hierarchy that performs rewriting and propagation in hierarchies given by the above-mentioned procedure.

In this thesis, we have presented a model of an audit trail for updates in individual objects and hierarchies. Such a trail allows maintaining the history of transformations and provides means for reverting sequences of transformations. Moreover, it enables accommodation of multiple versions of the same object diverged as the result of conflicting rewrites. To design this model we have investigated the questions of reversibility and composition of SqPO rewriting for individual objects and hierarchies. In particular, we have introduced the construction that allows composing consecutive SqPO rewrites, where the first rewrite is reversible. We have also described the conditions for rewriting in hierarchies to be reversible and the construction for composing consecutive applications of two rule hierarchies.

The presented framework is implemented in the Python library `ReGraph` for the category of simple graphs with attributes. The library allows creating and transforming individual graphs and hierarchies using two backends: in-memory graphs and persistent graphs stored in the Neo4j database.

The presented KR framework provides means for expressing corpora of fragmented knowledge on different abstraction levels and relating them with homomorphisms. It supports updates of individual fragments that can be dynamically propagated to other parts of the corpus. Such propagation guarantees the consistency of knowledge at all times. The system can be used for modelling and curation of knowledge on entities and their relations in any domain. The two large use-cases of the presented system are discussed in the chapters that follow, namely a model for schema-aware property graphs and a bio-curation system for cellular signalling knowledge.

2.4.1 Future work

The presented generalization of SqPO rewriting to hierarchies of objects gives rise to a number of interesting questions. Many classical topics in graph transformation can be investigated with respect to rewriting in hierarchies, such as concurrency and parallelism of such rewriting [36], negative [46] or even nested application conditions [35] and so on. In the rest of this section, however, we will focus on some concrete future directions relevant to the KR and curation.

The first such direction could consist in adding the *undo* operation to the audit trail for individual objects and hierarchies. For this to be done, we need to investigate how an arbitrary transformation in a sequence of rewrites can be undone. For instance, in the case of rewriting

performed on individual objects, this question can be formulated as follows. For two conecutive rule applications, how the rewriting specified by the first rule can be 'undone' in a way that makes the application of the second rule (or its modified version) possible.

This question is directly related to the notions of *sequential independence* of two consecutive SqPO rewrites studied in [29], as well as enablement and prevention from [22]. We can investigate the sequential independence of the rewrites by constructing *witnesses of independence* as described in [29]. Roughly speaking, the existence of a witness of independence indicates that the transformations specified by first rule can be performed before the application of the second rule. In case when the two rewrites are not sequentially independent, we can try to find a 'subrule' of the second rule that can be applied even when the first rule was not applied. A similar question can be asked for rewriting and propagation in hierarchies of objects or, in general, for consecutive applications of rule hierarchies.

Another interesting feature of a KR system based on hierarchies of graphs would be to incorporate some calculus for expressing structural constraints on nodes and edges. Such constraints could, for example, restrict the number of outgoing edges from a node, i.e. for a homomorphic pair of graphs $G \to T$ we could state 'all the nodes in G that map to some node in T can have at most one outgoing edge'. Then, the interplay between rewriting, backward/forward propagation and such constraints could be investigated.

Some future work remains to be done in the `ReGraph` library. For example, some features of in-memory graphs and hierarchies are not implemented for the Noe4j-based backend due to limitations of the underlying database technologies, e.g. accommodation of rules as hierarchy nodes, the implementation of sets of attributes defined by regular expressions or integer intervals. Moreover, at the current stage of `ReGraph`'s development, audit trails are represented as in-memory objects whose lifetime is limited by the lifetime of the application. Therefore, it would be interesting to implement persistent encoding of audit trails using, for example, the Neo4j database.

Chapter 3

Schema validation and evolution for graph databases

Property graph (PG) databases are widely used to represent complex data stored as a collection of nodes, edges and their properties. Various database management technologies based on PGs have emerged in the recent decade, e.g. Neo4j, Oracle Spatial and Graph, SAP HANA, Redis Graph, TigerGraph and so on. Alongside these technologies a number of query and graph traversal languages have been developed including Cypher, GraphQL, Gremlin, etc. However, due to the fact that the PG data model has been originally conceived as *schema-free*, these technologies lack the support for PG schemas, i.e. schema specification and schema modification capabilities.

Node labels and relationship types in PG data model are often used to represent types of entities and relations that exist in a given database. Looking at the set of node labels and relationship types can, therefore, be seen as the first step towards what could be a schema for the underlying property graph. Moreover, the Neo4j database, for example, provides some primitive aspects of schemas via the use of *constraints*. These constraints include property *uniqueness* and *existence* constraints for nodes and relations. Neo4j also allows to create node keys: the user can specify that a node with particular label has a set of properties and their combined value is required to be unique. However, the use of node labels, relationship types and constraints does not allow for more advanced schema-related features, such as specifying a set of allowed relationship types between different node labels or a set of allowed properties and their types for a given node label or relationship type. Currently, no standardized data definition language (DDL) for PGs exists, therefore schemas seen as sets of node labels and relationship types stay *descriptive* in the sense that they only reflect the data, but do not define data specification in a *prescriptive* way. Moreover, there exist no tools that would allow PGs schema modification or provide mechanisms for data/schema co-evolution.

This chapter provides a proposal for a *PG schema model* that allows for a full range of schema-related functionality: prescriptive schema definition, schema validation and schema evolution. More precisely, the contributions as a part of this thesis include:

- a schema model specifying labels and properties for nodes and edges together with a concise schema DDL following intuitive ASCII-art syntax inspired by Cypher;
- a mathematical framework for schema validation allowing us to construct both data graph and schema as PGs and to enforce schema validation through a homomorphism from data

to schema;

- application of the mathematical framework presented in Chapter 2 that allows to define data and schema updates using SqPO rewriting approach and provide mechanisms for data/schema co-evolution;
- a prototype implementation of schema-aware Neo4j-based PGs equipped with capabilities for data/schema co-evolution (implemented as a part of the `ReGraph` library).

The contributions on schemas for property graphs presented in this chapter were communicated as a conference paper in [11] and as a longer version in [10].

3.1 Schemas for property graphs

Let us start this section by providing a traditional formal definition of property graphs [9], i.e. the definition of *labeled* property graphs. First of all, let us fix the following sets: a set of *objects* $\mathcal{O}$, a finite set of *labels* $\mathcal{L}$, a set of *property keys* $\mathcal{K}$ and a set of *values* $\mathcal{V}$.

Definition 3.1.1. A *property graph (PG)* is given by a tuple $(N, E, \eta, \lambda, P, \nu)$, where:

- $N \subseteq \mathcal{O}$ is a finite set of *nodes*;
- $E \subseteq \mathcal{O}$ is a finite set of *relationships* (or *edges*);
- $\eta : E \to N \times N$ is a function that assigns an ordered pair of nodes to every relationship;
- $\lambda : N \cup E \to \mathcal{P}(\mathcal{L})$ is a function that assigns a finite set of labels to every node and relationship (here $\mathcal{P}(\mathcal{L})$ denotes the power set of $\mathcal{L}$);
- $P \subseteq (N \cup E) \times \mathcal{K}$ is a finite set of *properties*;
- $\nu : P \to \mathcal{V}$ is a partial function that assigns property values to properties.

such that N and E are disjoint.

Remark 3.1.2. Note that in the graph database community the edges of underlying graphs are called both edges and relationships interchangeably.

Remark 3.1.3. Note also that Definition 3.1.1 treats previously mentioned *relationship types* as labels assigned to relationships and allows for a single relationship to have *multiple* such labels. However, some database management technologies (among which is `Neo4j` used as the main DB technology in this thesis) only allow each relationship to have exactly one type. In the rest of this section we assume that relationships can have multiple such labels and will call them labels rather than types.

An example of a PG is illustrated in Figure 3.1. As we have previously mentioned, sets of node and relationship labels can be used to group nodes and relations and represent types of entities and relations that exist in the database. Therefore, we can say that the node n_1 from Figure 3.1 is typed as `Person` and n_2 as `Message`.

To define PG schemas and to make an analogy to graphs with attributes (defined in Appendix A.1), in this thesis we will use a slightly modified definition of PGs given in Definition 3.1.4 (similarly to [10]). Such a modified definition, in particular, allows us to interpret the DDL specification that we design as a PG.

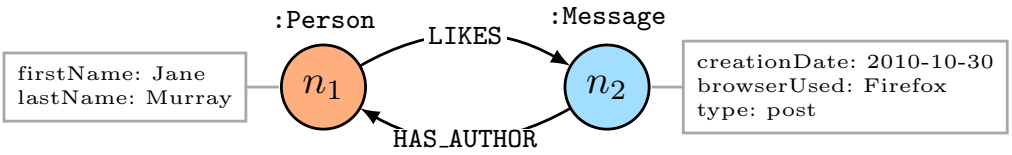


Figure 3.1: Example of a PG. Node labels are represented with small text labels starting with a column and situated above the nodes. Relationship labels correspond to text labels attached to arrows. Finally, properties are given in gray boxes attached to nodes or relationships.

Definition 3.1.4. A *property graph (PG)* is given by a tuple (N, E, η, P, ν), where:

- $N \subseteq \mathcal{O}$ is a finite set of *nodes*;
- $E \subseteq \mathcal{O}$ is a finite set of *relationships* (or *edges*);
- $\eta : E \to N \times N$ is a function that assigns an ordered pair of nodes to every relationship;
- $P \subseteq (N \cup E) \times \mathcal{K}$ is a finite set of *properties*;
- $\nu \subseteq P \times \mathcal{V}$ is a finite relation that that assigns *sets* of values to properties;

such that N and E are disjoint.

Remark 3.1.5. First of all, we have removed labels from the definition of property graph above. Later in this section we will see how, for a given PG, node and relationship labels can be brought back by establishing a homomorphism to a schema PG (label sets can be seen as nodes and relationships of this schema PG).

Remark 3.1.6. Contrary to the previous definition of PGs, properties are associated with *sets* of values and not single values from $\mathcal{V}$. This allows us to easily translate Definition 3.1.4 in terms of graphs with attributes (where labels can be encoded with special types of attributes). In particular, it allows us to define the constructions of pullback, pushout and final pullback complement on property graphs similarly to the ones for graphs with attributes.

We first provide a DDL for specifying PG schemas and then show how the schema defined using this DDL can be seen as a PG from Definition 3.1.4.

3.1.1 Data definition language for property graphs

Traditional schema definitions, for example, in relational databases, allow defining not only sets of attributes (columns) constituting our data, but also the data types for the values of these data attributes. Therefore, before we can give any adequate proposal for a PG DDL, we need to discuss the question of data types and mechanisms through which these data types interact with property values.

Data types as property values. Let $\mathcal{T}$ be a finite set of *data types* (for instance, in Neo4j such data types include `STRING`, `INTEGER`, `DATETIME` etc.). Let us assume that every data type $t \in \mathcal{T}$ defines a subset of values from $\mathcal{V}$, i.e. $t \subseteq \mathcal{V}$. For example, we will say that the data type `INTEGER` in Neo4j defines the set of all integer numbers and the data type `STRING` defines the set of all strings, etc. Let us also assume that the set of data types $\mathcal{T}$ itself is a

subset of $\mathcal{V}$, i.e. $\mathcal{T} \subseteq \mathcal{V}$. Having defined data types in such a way, the task of testing if a given value $v \in \mathcal{V}$ is of a specific type $t \in \mathcal{T}$ reduces to the testing if $v \in t$, i.e. if v is in the set of values defined by the data type t. In the sections on schema evolution that follow we will perform different set-theoretic operations on subsets of $\mathcal{V}$. For instance, as part of the construction of final pullback complements and pushouts, we will perform set difference and union. However, put in a practical implementation context, the results of some of these differences and unions should be clarified. For example, in the Neo4j type system, we would need to clarify what is the result of `INTEGER` minus $\{1, 3\}$, `INTEGER` minus $(-\infty, -1]$ or even `STRING` minus `INTEGER`. For example, the library `ReGraph` (discussed in Section 2.3), provides a module `regraph.attribute_sets` that handles the data types and mixed operations between data types and values of these data types. This allows to symbolically represent sets of integers defined by a sequence of intervals or sets of strings recognized by a regular expression. It handles operations like "`INTEGER` minus $\{1, 3\}$" (by constructing the corresponding set of integer intervals) and "`STRING` minus `INTEGER`" (by performing the casting of integers to strings and constructing a regular expression that discards all the string representation of integers). In this chapter we will omit these implementation details and assume that for any two sets of values $V_1, V_2 \subseteq \mathcal{V}$, we can construct $V_1 \setminus V_2, V_1 \cup V_2 \subseteq \mathcal{V}$, etc.

To define our DDL we will use node and relationship labels as type identifiers. The basic components of a schema definition include:

- *Property type*, a pair $(k, t) \in \mathcal{K} \times \mathcal{T}$, where k is the property key and t is its data type. For example, "`content: STRING`" declares the property type (`content`, `STRING`).

- *Element type* $b \in \mathcal{BT}$ is a tuple (l, P, B), where $l \in \mathcal{L}$ is a label, P is a set of property types, and $B \subseteq \mathcal{BT}$ is the set of element types that b inherits. An element type can inherit multiple other element types, but must not inherit itself either directly or indirectly. For example, "`Message {content: STRING?, length: INTEGER}`" is a declaration of the element type $m = (\texttt{Message}, \{pt_1, pt_2\}, \emptyset)$, where $pt_1 = (\texttt{content}, \texttt{STRING})$ and $p_2 = (\texttt{length}, \texttt{INTEGER})$. On the other hand "`Post :: Message {language: STRING?}`" declares the element type $p = (\texttt{Post}, \{pt_3 = (\texttt{language}, \texttt{STRING})\}, \{m\})$. In this example, the element type associated to the label `Post` inherits the one associated to `Message`. The set of property types of an element type $b = (l, P, E)$ is defined as $\texttt{prop}(b) := P \cup \bigcup_{b' \in B} \texttt{prop}(b')$, i.e. all the property types that b possesses, either directly or through inheritance. Similarly, we define $\texttt{labels}(b)$ to be the set of labels of b. For instance, for element type p, $\texttt{prop}(p) = \{pt_1, pt_2, pt_3\}$ and $\texttt{labels}(p) = \{\texttt{Post}, \texttt{Message}\}$. From now on, for the sake of conciseness, we will often refer to element types by their associated labels.

- *Node type* $nt \in \mathcal{NT}$ is a 1-tuple (b) where $b \in \mathcal{BT}$ is an element type, for example, "`(Post)`" declares the node type $p' = (\texttt{Post})$. For a node type $nt = (b)$, we define $\texttt{prop}(nt) = \texttt{prop}(b)$ and $\texttt{labels}(nt) = \texttt{labels}(b)$. In the rest of this section we will refer to node types by the labels associated to their element types.

- *Relationship type* $et \in \mathcal{ET}$ is a triple (s, b, t), where s, b, and t are element types. For instance, "`(Comment)-[REPLY_OF]->(Message)`" declares the edge type (`Comment`, `REPLY_OF`, `Message`). Note that s and t need not be node types. This allows defining a single edge type between multiple pairs of node types inheriting s and t. In the rest of this section we will refer to relationship types by the labels associated to their element types.

- *Property graph type*, a triple $(\mathcal{BT}, \mathcal{NT}, \mathcal{ET})$, where $\mathcal{BT}$ is a set of element types, $\mathcal{NT}$ is a set of node types and $\mathcal{ET}$ is a set of relationship types. A property graph type provides the schema for a PG.

Remark 3.1.7. For an element type $b \in \mathcal{BT}$ to be valid, $\texttt{prop}(b)$ can have exactly one property type with the same key, i.e. all property types of an element type are uniquely determined by their key.

Example 3.1.1. *The following example of the proposed PG schema DDL creates a property graph type that captures a fragment of the LDBC SNB[39] schema.*

```
CREATE GRAPH TYPE snb (
  // element types
  Person {
    firstName : STRING, lastName : STRING
  },
  Message {
    creationDate : TIMESTAMP, browserUsed : STRING
  },
  Comment <: Message {},
  Post <: Message {
    imageFile : STRING
  },
  KNOWS {creationDate : TIMESTAMP},
  LIKES {creationDate : TIMESTAMP},
  HAS_CREATOR {creationDate : TIMESTAMP},
  REPLY_OF {},
  // node types
  (Person), (Post), (Comment),
  // edge types
  (Person)-[KNOWS]->(Person),
  (Person)-[LIKES]->(Message),
  (Message)-[HAS_CREATOR]->(Person),
  (Comment)-[REPLY_OF]->(Message)
)
```

3.1.2 Schema validation

Definition 3.1.8. Given a property graph type $(\mathcal{BT}, \mathcal{NT}, \mathcal{ET})$. The corresponding schema PG S is given by a tuple $(N_S, E_S, \eta_S, P_S, \nu_S)$, where:

- N_S is given by the set of node types $\mathcal{NT}$;
- E_S is given by the set of relationship types $\mathcal{ET}$;
- η_S associates the elements from $\mathcal{ET}$ to pairs of elements from $\mathcal{NT}$ in such a way that $\eta_S(e) = (n_1, n_2)$ for every $e \in \mathcal{ET}$ if and only if $e = (n_1, b, n_2)$ for some element type $b \in \mathcal{BT}$ and node types $n_1, n_2 \in \mathcal{NT}$;
- $P_S \subseteq (\mathcal{NT} \cup \mathcal{ET}) \times \mathcal{K}$ is the set of properties defined in such a way that for all $e \in \mathcal{NT} \cup \mathcal{ET}$ and $k \in \mathcal{K}$, $(e, k) \in P_S$ if and only if $k \in \{k' \in K \mid (k', t) \in \texttt{props}(e)\}$;

- $\nu_S : P_S \to \mathcal{T}$ is a function that assigns properties from P_S to data types defined in the way that $\nu(p) = t$ for some $p = (e, k) \in P_S$, $e \in \mathcal{NT} \cup \mathcal{ET}$, $k \in \mathcal{K}$ and $t \in \mathcal{T}$ if and only if there exists $b \in \mathcal{BT}$ such that $(k, t) \in \mathit{props}(b)$.

Example 3.1.2. *Figure 3.2 illustrates a PG constructed from the property graph type* `snb` *defined in Example 3.1.1.*

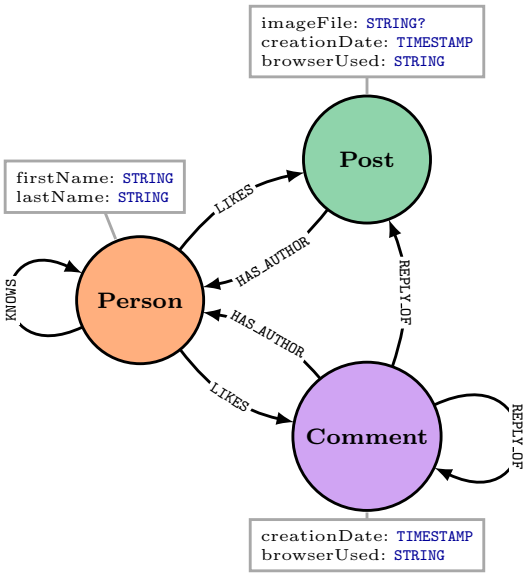


Figure 3.2: Example of a schema PG

Definition 3.1.9. Let G and S be two property graphs with $N_G \cup E_G$ and $N_S \cup E_S$ being disjoint. A *homomorphism* $h : G \to S$ is given by two functions $h_\mathcal{N} : N_G \to N_S$ and $h_\mathcal{E} : E_G \to E_S$, mapping the nodes and edges of G to nodes and edges of S, such that

- $\eta_S \circ h_\mathcal{E} = (h_\mathcal{N} \times h_\mathcal{N}) \circ \eta_G$
- and for $h := h_\mathcal{N} \cup h_\mathcal{E}$
  - if $(e, k) \in P_G$, then $(h(e), k) \in P_S$;
  - if $((e, k), v) \in \nu_G$ and $((h(e), k), t) \in \nu_S$, then $v \in t$.

Remark 3.1.10. The definition above states that a homomorphism $h : G \to S$ maps nodes and relationships of G to nodes and relationships of S is such a way that: (1) each relationship of G with source and target nodes n_1 and n_2 is mapped to a relationship in S with source and target nodes $h_\mathcal{N}(n_1)$ and $h_\mathcal{N}(n_2)$, (2) all properties in G are instances of properties in S, and (3) each property in G is associated with a subset of the values whose elements are of the data type associated to the corresponding property in S.

Remark 3.1.11. A PG homomorphism $h = (h_\mathcal{N}, h_\mathcal{E})$ is monic if both $h_\mathcal{N}$ and $h_\mathcal{E}$ are injective.

We can now view a homomorphism $h : G \to S$ as a formalization of the notion of *schema validation*, where G is a data PG and S is a schema PG. In other words, G respects the schema S when

- each node/relationship e in G is an instance of the schema node/relationship $h(e)$;
- edges in S define the set of allowed edges in G between different types of nodes;
- each node/relationship e in G is associated with a subset of properties corresponding to the element $h(e)$ in S;
- values of the properties in G are of the data types associated to the respective properties in S.

The described approach represents a fairly simple notion of schema validation, e.g. it does not allow specifying the number of in-/outgoing edges, impose uniqueness constraints.

Example 3.1.3. *Figure 3.3 illustrates an example of a PG that complies with the schema defined in Example 3.1.1. The depicted PG is homomorphic to the PG from Figure 3.2, the respective homomorphism is colour-coded, i.e. all green nodes in Figure 3.3 map to the* `Post` *node in Figure 3.2, etc.*

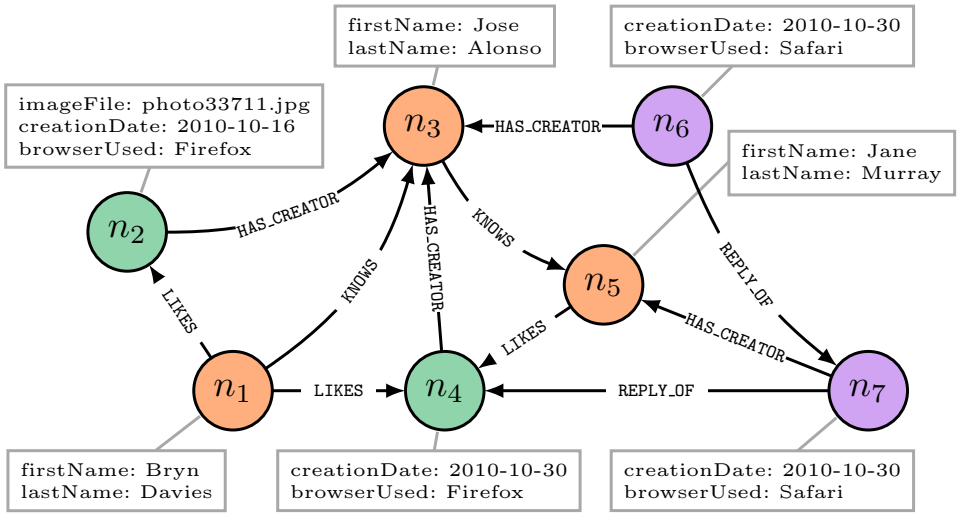


Figure 3.3: Example of a PG complying with the schema PG from Figure 3.2

Remark 3.1.12. Recall that in the previously presented DDL every element type was associated to a unique label. It means that we can encode a homomorphism $h : G \to S$ using node/relationship labels attached to the elements of G, where G is a PG from Definition 3.1.1. And, vice versa, having a property graph from Definition 3.1.4 and a homomorphism h, for every element e from G we can construct its label set by looking at the corresponding element $h(e)$ in S. Every such element in S is associated to an element type in the underlying property graph type providing the corresponding set of labels.

3.2 Data and schema co-evolution

As we have mentioned before, sets of labels on nodes and edges of traditional PGs can be seen as *descriptive* schemas that reflect the shape of data. At the same time, in the previous section we have seen how *prescriptive* schemas for PGs can be specified and validated through

a homomorphism from data instances. While in the first setting a schema does not impose any constraints on a data instance (but simply reflects it), in the later one the data instance is *required* to comply with the schema, i.e. the schema imposes constrains on properties and relationships allowed between different types of nodes.

In the first scenario evolution of the instance is implicitly reflected in the schema, and in the second case such evolution is required to always respect the schema. On the other hand, because schemas of the first kind do not exist explicitly (for example, specified using a DDL or as a PG), we do not have any means for specifying their evolution. Having explicit prescriptive schemas (for example, as schema PGs from the presented data model) allows us to design mechanisms for their evolution. In this section we will study how the SqPO rewriting approach described in Section 2.1 can be used as an update semantics for performing the evolution of both data instance and schema PGs.

Moreover, what we would like to design here is a way to combine different evolution scenarios, we would like to provide means for:

1. performing schema evolution and specifying co-evolution of its instance to perform *prescriptive updates* propagated from the schema to the data;
2. reflecting the evolution of an instance in its schema to perform what we call *descriptive updates* propagated from the data to the schema.

We have previously advocated the necessity and benefits of prescriptive schemas for the PG data model, e.g. explicit prescriptive schemas can be modified when an application undergoes a change. In this context, the first scenario of prescriptive updates is a classical data-schema co-evolution problem which consists in adjusting the data instance to a modified version of its schema. However, assuming that a 'perfect' prescriptive schema can be designed and maintained throughout the entire life-cycle of an application is not realistic as data undergoes constant changes. Therefore, providing means for handling the second kind of co-evolution scenarios, the one of descriptive updates propagated from the data to the schema, is indispensable to make the development process of our application (at least on in its early stages) rigorous yet flexible. In this section, we provide mechanisms for realizing both update scenarios while keeping our data compliant with the schema at all times. We will apply the mathematical framework for rewriting and propagation from Section 2.2 in the hierarchy consisting of two PGs, the data graph G and the schema graph S, and the homomorphism $h : G \to S$, through which schema validation is performed.

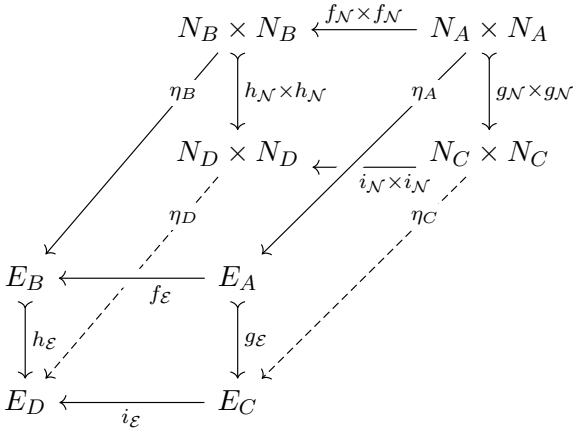
3.2.1 SqPO rewriting and propagation for PGs

To describe the effect of SqPO rewriting applied to PGs let us first define the constructions of a final pullback complement and a pushout for PGs.

Proposition 3.2.1. Given three PGs $A = (N_A, E_A, \eta_A, P_A, \nu_A)$, $B = (N_B, E_B, \eta_B, P_B, \nu_B)$ and $D = (N_D, E_D, \eta_D, P_D, \nu_D)$ and two homomorphisms $f = (f_{\mathcal{N}} : N_A \to N_B, f_{\mathcal{E}} : E_A \to E_B)$ and $h = (h_{\mathcal{N}} : N_B \to N_D, h_{\mathcal{E}} : E_B \to E_D)$ with h being a mono. The *final pullback complement* to $A \xrightarrow{f} B \overset{h}{\rightarrowtail} D$ is given by the PG $C = (N_C, E_C, \eta_C, P_C, \nu_C)$ and two homomorphisms $g : A \to C$ and $i : C \to D$, where:

- N_C, E_C, g and i are given by constructing the respective final pullback complements to $N_A \xrightarrow{f_{\mathcal{N}}} N_B \overset{h_{\mathcal{N}}}{\rightarrowtail} N_D$ and $E_A \xrightarrow{f_{\mathcal{E}}} E_B \overset{h_{\mathcal{E}}}{\rightarrowtail} E_D$ in the category **Set** (see Appendix A.5.2);

- η_C is the unique homomorphism determined by the universal property of the final pullback complement that constructed E_C as in the following diagram:

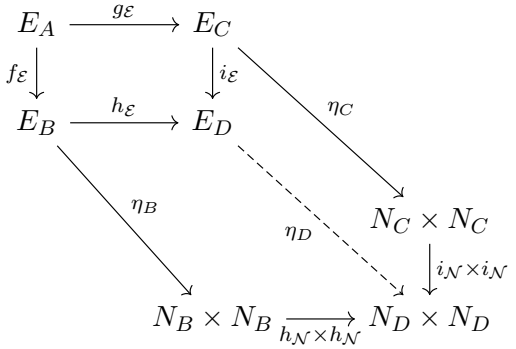


(3.1)

- $P_C \subseteq (N_C \cup E_C) \times \mathcal{K}$ is defined in a way that
  - for every $e \in N_A \cup E_A$ and $k \in \mathcal{K}$, $(g(e), k) \in P_C$ if $(e, k) \in P_A$;
  - for every $e \in N_D \cup E_D$ and $k \in \mathcal{K}$, $(i^{-1}(e), k) \subseteq P_C$ if there does not exist $e' \in N_B \times E_B$ such that $h(e') = e$ and $(e', k) \in P_B$;
- ν_C is given by the final pullback complement in $\mathbf{Set}_{fin}$ (see Appendix A.5.1);

Proposition 3.2.2. Given three PGs $A = (N_A, E_A, \eta_A, P_A, \nu_A)$, $B = (N_B, E_B, \eta_B, P_B, \nu_B)$ and $C = (N_C, E_C, \eta_C, P_C, \nu_C)$ and two homomorphisms $f = (f_\mathcal{N} : N_A \to N_B, f_\mathcal{E} : E_A \to E_B)$ and $g = (g_\mathcal{N} : N_A \to N_C, g_\mathcal{E} : E_A \to E_C)$, the *pushout* from $B \overset{f}{\leftarrow} A \overset{g}{\rightarrow} C$ is given by the PG $D = (N_D, E_D, \eta_D, P_D, \nu_D)$ and two homomorphisms $h : B \to D$ and $i : C \to D$, where:

- N_D, E_D, h and i are given by constructing the respective pushouts from $N_B \overset{f_\mathcal{N}}{\leftarrow} N_A \overset{g_\mathcal{N}}{\rightarrow} N_C$ and $E_B \overset{f_\mathcal{E}}{\leftarrow} E_A \overset{g_\mathcal{E}}{\rightarrowtail} E_C$ in the category **Set** (see Appendix A.4.2);
- η_D is the unique homomorphism determined by the universal property of the pushout that constructed E_D as in the following diagram:



(3.2)

- $P_D \subseteq (N_D \cup E_D) \times \mathcal{K}$ is defined is a way that

  - for every $e \in N_B \cup E_B$ and $k \in \mathcal{K}$, $(h(e), k) \in P_D$ if $(e, k) \in P_B$;
  - for every $e \in N_C \cup E_C$ and $k \in \mathcal{K}$, $(i(e), k) \in P_D$ if $(e, k) \in P_C$;

- ν_D is given by the pushout in $\mathbf{Set}_{fin}$ (see Appendix A.4.1)

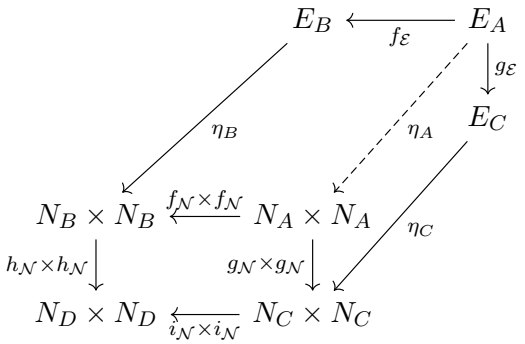
Now an update of an arbitrary PG following Definition 3.1.4 can be performed according to the semantics of the SqPO rewriting.

Given a data graph G complying with a schema graph S through a homomorphism $f : G \to S$, the objects G, S, f form a hierarchy from Section 2.2 with two hierarchy nodes corresponding to G and S and one hierarchy edge corresponding to f. To describe schema/data co-evolution mechanisms, we can use backward and forward propagation of SqPO rewriting. Before we can do that, we need to define the constructions of a pullback and image factorization for PGs.

Proposition 3.2.3. Given three PGs $B = (N_B, E_B, \eta_B, P_B, \nu_B)$, $C = (N_C, E_C, \eta_C, P_C, \nu_C)$ and $D = (N_D, E_D, \eta_D, P_D, \nu_D)$ and two homomorphisms $h = (h_{\mathcal{N}} : N_B \to N_D, h_{\mathcal{E}} : E_B \to E_D)$ and $i = (i_{\mathcal{N}} : N_C \to N_D, i_{\mathcal{E}} : E_C \to E_D)$. The *pullback* from $B \overset{h}{\to} D \overset{i}{\leftarrow} C$ is given by the PG $A = (N_A, E_A, \eta_A, P_A, \nu_A)$ and two homomorphisms $f : A \to B$ and $g : A \to C$, where:

- N_A, E_A, f and g are given by constructing the respective pullbacks from $N_B \overset{h_{\mathcal{N}}}{\to} N_D \overset{i_{\mathcal{N}}}{\leftarrow} N_C$ and $E_B \overset{h_{\mathcal{E}}}{\to} E_D \overset{i_{\mathcal{E}}}{\leftarrow} E_C$ in the category **Set** (see Appendix A.3.2);

- η_A is the unique homomorphism determined by the universal property of the pullback that constructed $N_A \times N_A$ as in the following diagram:



(3.3)

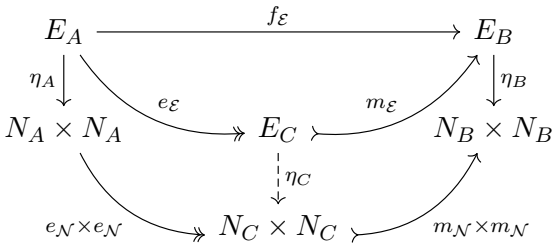
- $P_A \subseteq (N_A \cup E_A) \times \mathcal{K}$ is defined in a way that, for every $e \in N_A \cup E_A$ and $k \in \mathcal{K}$, $(e, k) \in P_A$ if $(f(e), k) \in P_B$ and $(g(e), k) \in P_C$.

- ν_A is given by the pullback in $\mathbf{Set}_{fin}$ (see Appendix A.3.1);

Proposition 3.2.4. Given two PGs $A = (N_A, E_A, \eta_A, P_A, \nu_A)$, $B = (N_B, E_B, \eta_B, P_B, \nu_B)$ and a homomorphism $f = (f_{\mathcal{N}} : N_A \to N_B, f_{\mathcal{E}} : E_A \to E_B)$, the *image factorization* of $A \overset{f}{\to} B$ is given by the PG $C = (N_C, E_C, \eta_C, P_C, \nu_C)$ and two homomorphisms $e : A \to C$ and $m : C \to B$, where:

- N_C, E_C, e and m are given by constructing the respective image factorizations of $N_A \overset{f_{\mathcal{N}}}{\to} N_B$ and $E_A \overset{f_{\mathcal{E}}}{\to} E_B$ in the category **Set** (see Appendix A.7.2);

- η_C is the unique homomorphism that can be constructed applying Lemma A.7.4 to the following diagram:



(3.4)

- $P_C \subseteq (N_C \cup E_C) \times \mathcal{K}$ is defined in a way that for every $x \in N_A \cup E_A$ and $k \in \mathcal{K}$, $(e(x), k) \in P_C$;
- ν_C is given by the image factorization in $\mathbf{Set}_{fin}$ (see Appendix A.7.1);

3.2.2 Rewriting and propagation scenarios

We are now able to formulate SqPO rewriting and propagation in the graph hierarchy consisting of the data graph G, the schema graph S and the schema-validating homomorphism f exactly the way they were formulated in Section 2.2. What we would like to do in the rest of this section, however, is to elaborate a couple of database application-driven scenarios, mentioned in the introduction to this section, in which such rewriting and propagation can be useful. These scenarios describe situations when an update of a given PG graph (the schema or the data graph) requires the co-evolution of its counterpart. Namely, we distinguish two such scenarios: a *prescriptive* update of the schema that propagates to the data instance and a *descriptive* update of the data that propagates to the schema.

Prescriptive updates through backward propagation. Prescriptive updates correspond to transformations of the schema PG that should be propagated to the instance for it to stay compliant. From Section 2.2 we know that such transformations correspond to restrictive updates (removes and clones of graph elements or properties), i.e. given a data-schema hierarchy composed of G, S and $f : G \to S$, restrictive updates of S induce backward propagation to G described in Section 2.2.2. In the context of schema evolution, removal of schema elements (or properties) models the removal of concepts (or properties of concepts) from the universe of discourse. Such removal induces the 'clean-up' of all the instances of these removed concepts (or properties). The operation of cloning of schema elements corresponds to *concept refinement*, i.e. as the result of cloning the original coarse-grained concept is split into more fine-grained concepts. As we know from Section 2.2.2, cloning of S can have a range of effects on the data instance G, i.e. instances of the cloned elements can be canonically cloned or retyped preserving the consistency of the hierarchy. Example 3.2.1 provides an example of such concept refinement and a prescriptive update it triggers.

Example 3.2.1. *Consider Figure 3.4 illustrating an example of a restrictive rule $L \leftarrow L^-$ applied to a schema PG S. Application of this rule matches the schema node* `Message` *in the original schema S and clones it into two nodes,* `Post` *and* `Comment`*, producing a new schema PG*

S^-. *In this example the rule performs concept refinement in which from a more coarse-grained concept of a message we create two more fine-grained concepts, of a post and a comment.*

Having applied a restrictive rule to the schema, to restore our data-schema hierarchy, we need to perform backward propagation to the instance. Consider the data instance G compliant with the initial schema S depicted in Figure 3.5. The homomorphism $f : G \to S$ is encoded with node colors (i.e. orange nodes represent persons and blue nodes represent messages). In this example, we will use the values of the property `type` *attached to the message nodes of our instance G to perform backward propagation. Namely, we will use these values to construct a controlling relation R from 2.2.4 as $R = \{(n_2, \texttt{Post}), (n_6, \texttt{Comment})\}$. We depict the established relation using orange dashed lines in Figure 3.5. As it is highlighted in the figure, the message node n_8 does not have a property* `type`*, therefore we do not include this node in the controlling relation R.*

As a result of the backward propagation controlled by R, we obtain the PG depicted in Figure 3.6. The nodes n_2 and n_6 are retyped by `Post` *and* `Comment` *respectively, while for the initial node n_8 the canonical propagation is performed, i.e. two instances n_8 and n_8' of* `Post` *and* `Comment` *respectively are produced.*

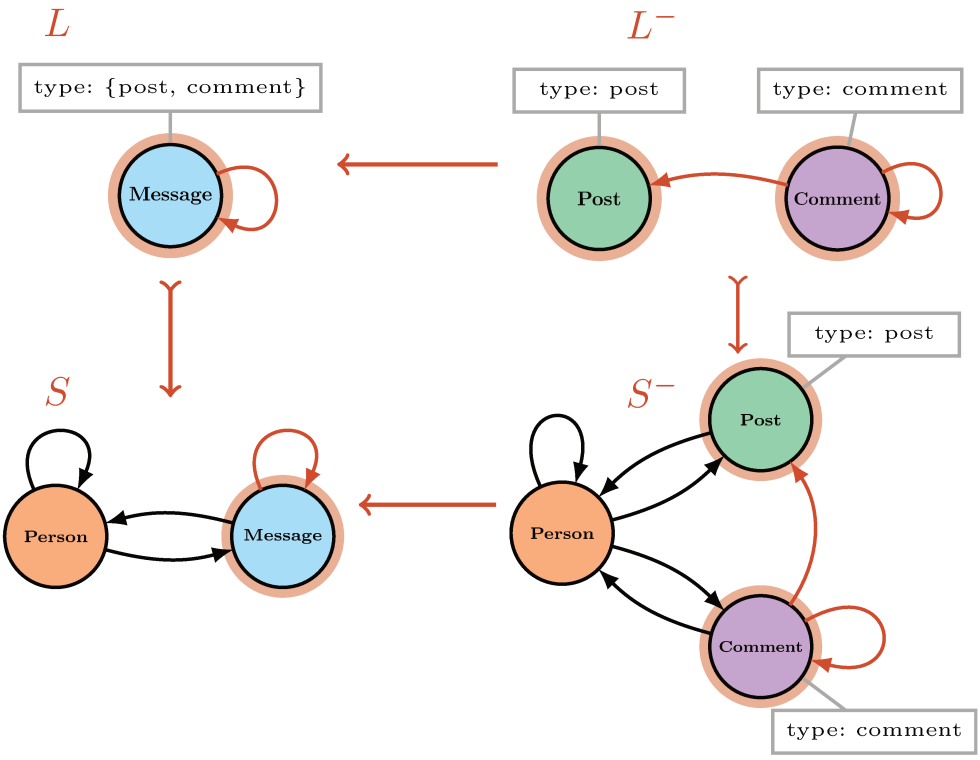


Figure 3.4: Example of a restrictive rewrite of a schema PG.

Descriptive updates through forward propagation. *Descriptive updates correspond to transformations of the data instance to be reflected in schema. From Section 2.2 we know that such transformations correspond to expansive updates (additions and merges of graph elements or properties), i.e. given a data-schema hierarchy composed of G, S and $f : G \to S$, expansive updates of G induce forward propagation to the schema S described in Section 2.2.3.*

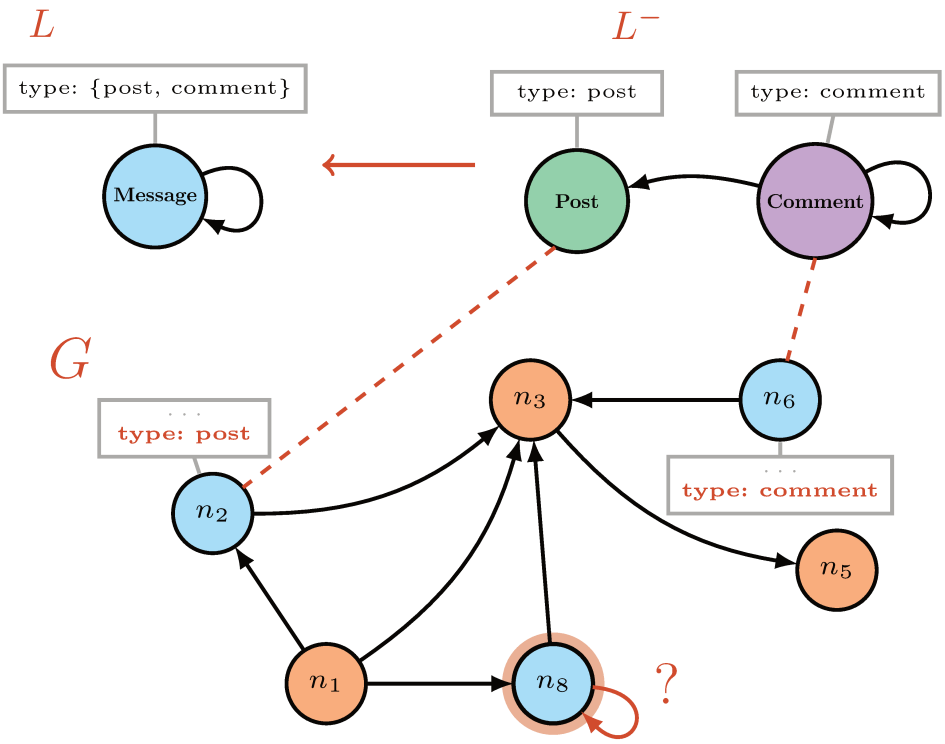


Figure 3.5: Example of a constrolling relation for backward propagation to the data instance corresponding to the schema and the rule from Figure 3.4.

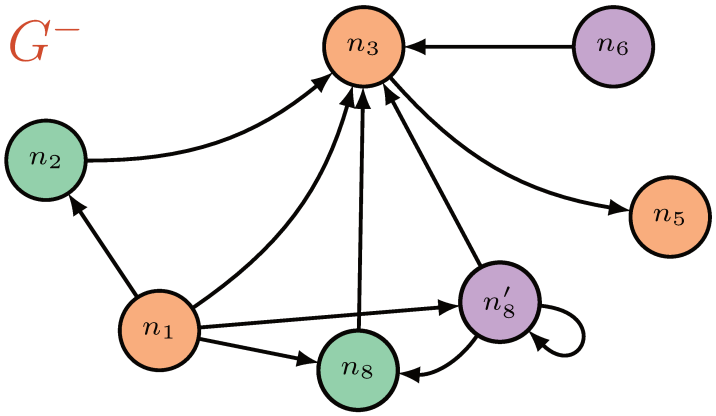


Figure 3.6: Result of the backward propagation specified by the controlling relation from Figure 3.5.

In the context of data evolution, the merge of data elements that correspond to different schema elements (that are instances of different schema concepts) induces the merge of these schema elements (join of the corresponding concepts). On the other hand, addition of new data elements can have a range of effects on the schema S, e.g. the newly added elements can be typed by already existing schema nodes or the new typing elements can be created. Example 3.2.2 provides an example of a descriptive update in which addition of new data elements is propagated to the schema.

Example 3.2.2. *Consider Figure 3.7 illustrating an example of an expansive rule $L \to L^+$ applied to a PG instance G. The rule matches the pattern highlighted with orange in G, adds a new node (corresponding to the node z in L^+), connects it with edges to the matched nodes and, finally, adds a new property* `city: Lyon` *to the node corresponding to y in the rule instance (the node n_6). Now, Figure 3.8 depicts a schema PG S typing G (as before, node colors represent the schema-validating homomorphism $f : G \to S$). To be able to perform a schema-respecting (strict) rewrite of G we need to provide a homomorphism $L^+ \to S$, i.e. we need to type the right-hand side of our rule by the schema. In this example, taking into account the initial matching of L into G, we obtain a map of two nodes in L^+ to S, namely the node x is typed as* `Person` *and the node y as* `Message`*. To type the node z, however, we will create a new schema element* `Checkin` *that introduces a new concept into the universe of our discourse, namely a check-in that can be related to persons and messages in the specified by L^+ way. Moreover, even though y is mapped to* `Message`*, this map cannot be used for constructing a homomorphism $L^+ \to S$ as it does not preserve properties. Therefore, using forward propagation induced by the original rewriting of G we would also like to add a new attribute* `city: STRING` *to the schema node* `Message`*. Figure 3.9 illustrates the updated instance G^+ and the new schema S^+ typing this instance.*

3.2.3 Expressing schema rewriting with schema modification operations

We have described how the update semantics given by the SqPO rewriting can be used to perform both data and schema rewriting. Recall how in Section 2.1 we interpreted the effect of an SqPO rewrite in terms of primitive graph transformations, i.e. deletions, clones, additions and merges of graph elements. Such transformations of the data PG are fully supported by the update semantics of modern PG query languages (addition and deletion are supported natively, while cloning and merging can be performed formulating slightly more complex queries as in Appendices D.1 and D.2). We would like to design a set of schema modification operations (SMOs) that allow to express an arbitrary SqPO rewrite of a schema PG. Similarly, to reflect the evolution of our schema PG $S = (N, E, \eta, P, \nu)$, we will interpret an arbitrary rewriting as a sequence of the following SMOs.

- *Split* a schema element $x \in N \cup E$ into elements $x_1, x_2, \ldots, x_k$ (note that we assume that $x_1, x_2, \ldots, x_k \notin N \cup E$). For example, "`SPLIT (Message) INTO (Post), (Comment)`" clones the schema node (`Message`) into two nodes (`Post`) and (`Comment`).
- *Drop a schema node $n \in N$.* For example "`DROP (Message)`" deletes the node (`Message`) from the schema. Note that as a side-effect of the drop operation all the schema edges incident to the removed node are removed.
- *Drop a schema* relationship $e \in E$. For example "`DROP (Person)-[KNOWS] ->(Person)`" deletes the loop relationship `KNOWS` of the node (`Person`) from the schema.

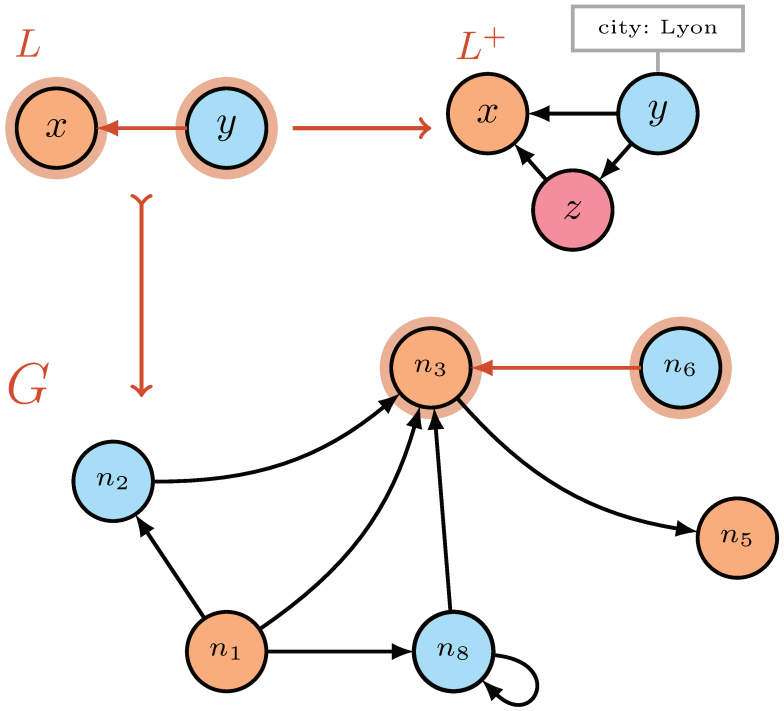


Figure 3.7: Example of an expansive update of a PG instance G.

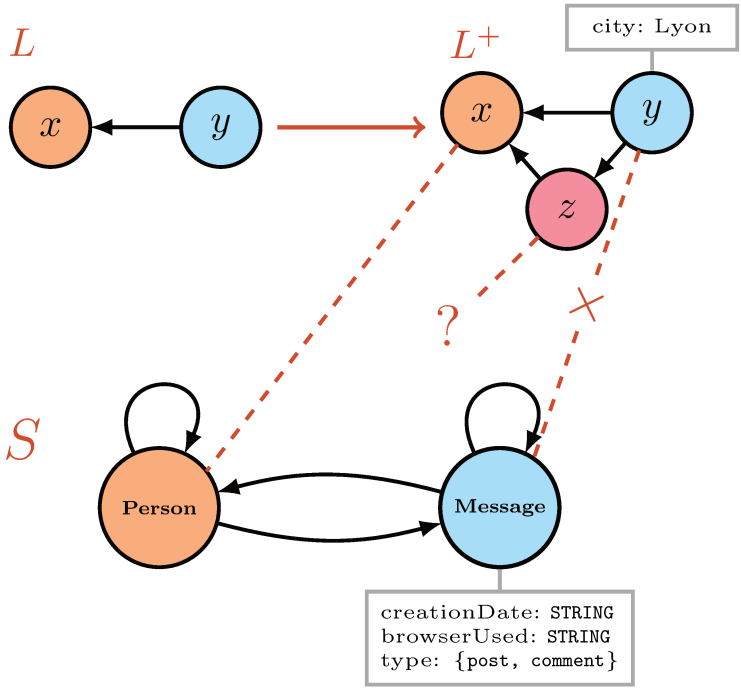


Figure 3.8: Example of a PG schema S corresponding to the instance from Figure 3.7 (the node colors of G encode a map to schema nodes). The right-hand side of the rule from Figure 3.7 does not comply with the schema (failure to establish a homomorphism from $L^+ \to S$).

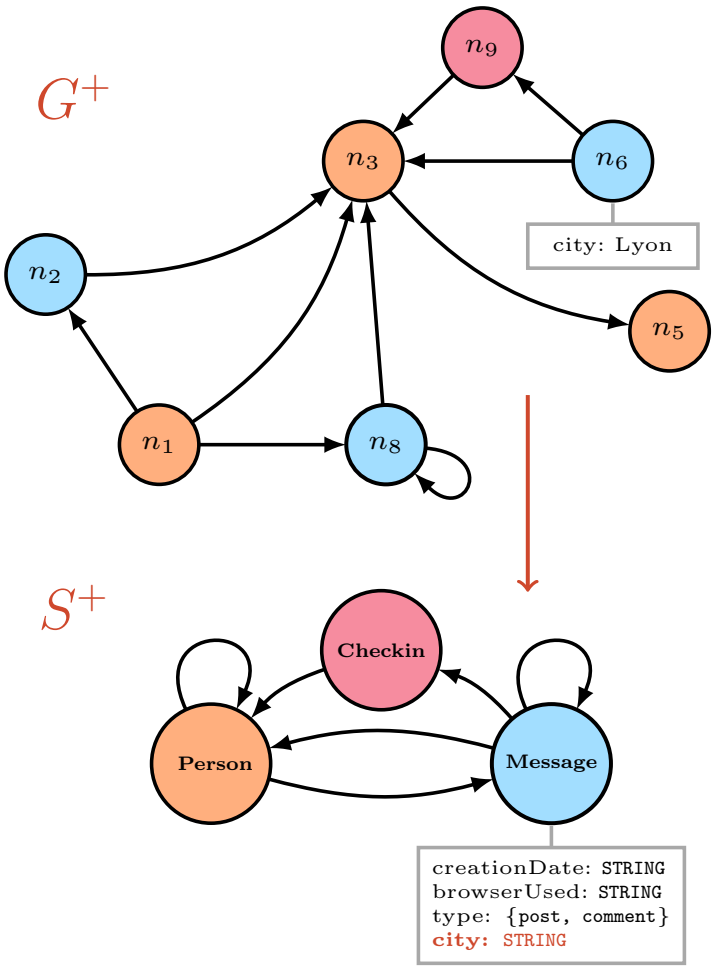


Figure 3.9: The result of rewriting $G \rightsquigarrow G^+$ from Figure 3.7 and the updated schema S^+ after forward propagation of the rewrite.

- *Drop a property* p from $x \in N \cup E$. For example, "`DROP PROPERTY browserUsed FROM Message`" removes the property `browserUsed` from the schema node `Message`.

- *Join* schema elements $x_1, x_2, \ldots, x_n \in N \cup E$ into x. For example "`MERGE Post, Comment INTO Message`" merges the schema nodes `Post` and `Comment` into a new schema node `Message`. Note that elements $x_1, x_2, \ldots, x_n$ are required to be of the same type, i.e. all elements are either nodes or edges of the schema. In addition, if the merged elements are edges they are all required to have the same source and target nodes.

- *Create a new schema node* n with the possibility to specify its properties and inherited element types. For example, "`CREATE (Checkin)`" adds a new node `Checkin` to the schema, "`CREATE (Checkin) WITH { location: STRING, creationDate: TIMESTAMP }`" adds a new node `Checkin` and associates the specified properties to this node. The SMO "`CREATE (Story) WITH {timeout: INTEGER} INHERIT Message`" adds a new node `Story` to the schema which inherits the existing element type `Message` and extends it with the specified properties.

- *Create a new schema relationship* e from a schema node $s \in N$ to a schema node $t \in N$ with the possibility to specify its properties and inherited element types. For example, "`CREATE RELATIONSHIP HAS_LOCATION FROM Message TO Checkin WITH { type: STRING }`" adds a new relationship `HAS_LOCATION` from the node `Message` to the node `Checkin` and associates the specified properties to this relationship.

- *Create property* p for a schema element $x \in N \cup E$. For example, "`CREATE PROPERTY device : STRING for Message`" adds the specified property and its value datatype to the node `Message` of the schema.

3.2.4 Reflecting schema evolution in PG types

In this subsection we would like to discuss how a rewritten schema can be translated back to a PG type expressed in the DDL presented in Subsection 3.1.1. Let $(\mathcal{BT}, \mathcal{NT}, \mathcal{ET})$ be the initial PG type corresponding to a schema graph $S = (N, E, \eta, P, \nu)$. Given a schema transformation $S \rightsquigarrow S'$ we would like to compute the updated PG type that corresponds to the updated schema graph S'. Recall that, by construction of S, $N := \mathcal{NT}$ and $E := \mathcal{ET}$. Recall also that an arbitrary SqPO rewrite of the schema can be seen as a sequence of SMOs. Therefore, to construct the updated PG type it is enough to describe how every individual SMO defined in the previous subsection updates the original PG type. Let the tuple $(\mathcal{BT}', \mathcal{NT}', \mathcal{ET}')$ define the PG type after performing an SMO. For the respective operations the updated sets of element types $\mathcal{BT}'$, node types $\mathcal{NT}'$ and edge types $\mathcal{ET}'$ are defined as follows.

Split. Consider an operation of splitting an element $e \in N \cup E$ into elements $e_1, e_2, \ldots, e_k$. Let e be associated with an element type $b = (l, P, B) \in \mathcal{BT}$ and let $l_1, l_2, \ldots, l_k$ be a sequence of new labels corresponding to the elements $e_1, e_2, \ldots, e_k$ respectively (for example, specified by the user or generated automatically). We create a new element type $b_i = (l_i, \emptyset, \{b\})$ inheriting the original element type b for every $i \in [1 \ldots k]$. The new set of element types is then constructed as follows $\mathcal{BT}' := \mathcal{BT} \cup \{b_1, b_2, \ldots, b_k\}$..

If our original element is a node, i.e. $e \in N$, we obtain the updated set of node types as $\mathcal{NT}' := \mathcal{NT} \cup \{(b_1), (b_2), \ldots, (b_k)\}$ and keep the set of relationship types unchanged, i.e

$\mathcal{ET}' := \mathcal{ET}$. On the other hand, if $e \in E$, we set $\mathcal{NT}' := \mathcal{NT}$ and obtain the set of updated relationship types as $\mathcal{ET}' := \mathcal{ET} \cup \{(s, b_1, t), (s, b_2, t), \ldots, (s, b_k, t)\}$, where s and t are element types corresponding to the source and the target node of the original relationship e.

Drop type. The operation of dropping a schema element operates exclusively on the set of node and relationship types and does not affect the set of element types $\mathcal{BT}$. More precisely, if we drop a node type $n \in N$ corresponding to an element type $b \in \mathcal{BT}$ we construct the updated node and relationship type sets as $\mathcal{NT}' := \mathcal{NT} \setminus \{n\}$ and $\mathcal{ET}' := \mathcal{ET} \setminus \{(s, b, t) \in \mathcal{ET} \mid s = b \text{ or } t = b\}$ respectively. If we drop a schema relationship $e \in E$, we update only the set of relationship types and set $\mathcal{ET}' := \mathcal{ET} \setminus \{e\}$.

Drop property. The operation of dropping a property from a node or a relationship type in slightly more elaborated due to the fact that this operation needs to be propagated to the set of all inherited element types associated to the corresponding affected element type. In general, the operation of dropping a property p from some element type $b = (l, P, B) \in \mathcal{BT}$ constructs the following sets of updated properties $P' := P \setminus \{p\}$.

However, apart from the element type b itself, the property p needs to be dropped from the set of *all* (directly and indirectly) inherited element types, i.e. from all the elements of the following set:

$$\hat{B} := \{\hat{b} \in \mathcal{BT} \mid \hat{b} \neq b \text{ and the label defining } \hat{b} \text{ is in } \texttt{labels}(b)\}.$$

We further observe that having dropped p from all the elements in $\{b\} \cup \hat{B}$ we have also dropped p from every element type that inherits one of the elements in $\{b\} \cup \hat{B}$ as a side-effect, i.e. we have inadvertently dropped p from the elements of the following set:

$$\check{B} := \{\check{b} = (\check{l}, \check{P}, \check{B}) \in \mathcal{BT} \mid \check{b} \notin \{b\} \cup \hat{B} \text{ and } \exists \hat{b} \in \check{B} \text{ such that } \hat{b} \in \{b\} \cup \hat{B}\}.$$

To fix this undesirable side-effect, for every element $\check{b} \in \check{B}$, we need to add the property p to the set of its properties. Therefore, to obtain the updated set $\mathcal{BT}'$, we need to update the properties of the element types from $\mathcal{BT}$ in the following way:

- we drop p from all the elements in $\{b\} \cup \hat{B}$ and
- we add p to all the elements in $\check{B}$.

We then update our node/relationship types from $\mathcal{NT} \cup \mathcal{ET}$ by associating the corresponding updated element types.

Join. Consider the operation of joining schema elements $e_1, e_2, \ldots, e_k \in N \cup E$ into a new element e. Let $b_1, b_2, \ldots, b_k \in \mathcal{BT}$ be the element types associated to the schema elements $e_1, e_2, \ldots, e_k$ and let l be a new label for the element e resulting from the merge operation. The new element type b' corresponding to e simply inherits the element types $b_1, b_2, \ldots, b_k$, i.e. b' is given by a tuple $(l, \emptyset, \emptyset, \{b_1, b_2, \ldots, b_k\})$. The updated set of element types is defined as $\mathcal{BT}' := \mathcal{BT} \cup \{b'\}$ and the new node/relationship type associated to b' is added to $\mathcal{NT}$ or $\mathcal{ET}$.

Create type. Consider the operation of creating a new schema element e with property types $PT \subseteq \mathcal{K} \times \mathcal{T}$ inheriting a set of existing element types $B \subseteq \mathcal{BT}$. Let l be a new label associated with e, then the updated set of element types is obtained by simply adding a new element type $b' = (l, PT, B)$. The new node or relationship type associated to b' is added to $\mathcal{NT}$ or $\mathcal{ET}$ respectively.

Create property. Creation of a property type $pt \in \mathcal{K} \times \mathcal{T}$ for a schema element $e \in N \cup E$ simply adds this property type to the element type $b \in \mathcal{BT}$ associated with e.

Example 3.2.3. *Consider the property graph type given by the tuple* $(\mathcal{BT}, \mathcal{NT}, \mathcal{ET})$*, where*

$$
\begin{aligned}
\mathcal{BT} := \{ & \\
& (\texttt{Person}, \{(\texttt{firstName}, \texttt{STRING}), (\texttt{lastName}, \texttt{STRING})\}, \emptyset) \\
& (\texttt{Message}, \{(\texttt{creationDate}, \texttt{TIMESTAMP}), (\texttt{browserUsed}, \texttt{STRING})\}, \emptyset), \\
& (\texttt{KNOWS}, \{(\texttt{creationDate}, \texttt{TIMESTAMP})\}, \emptyset), \\
& (\texttt{LIKES}, \{(\texttt{creationDate}, \texttt{TIMESTAMP})\}, \emptyset), \\
& (\texttt{HAS\_CREATOR}, \{(\texttt{creationDate}, \texttt{TIMESTAMP})\}, \emptyset), \\
& (\texttt{REPLY\_OF}, \emptyset, \emptyset)\}, \\
\mathcal{NT} := \{ & (\texttt{Person}), (\texttt{Message})\}, \\
\mathcal{ET} := \{ & \\
& (\texttt{Person}, \texttt{KNOWS}, \texttt{Person}), \\
& (\texttt{Person}, \texttt{LIKES}, \texttt{Message}), \\
& (\texttt{Message}, \texttt{HAS\_CREATOR}, \texttt{Person}), \\
& (\texttt{Message}, \texttt{REPLY\_OF}, \texttt{Message})\},
\end{aligned}
$$

defining the schema graph S from Figure 3.4. Let us apply the following split operation "`SPLIT (Message) INTO (Post), (Comment)`*". As the result of this operation, we obtain the updated property graph type given by the tuple* $(\mathcal{BT}', \mathcal{NT}', \mathcal{ET}')$*, where*

$$
\begin{aligned}
\mathcal{BT}' &:= \mathcal{BT} \cup \{(\texttt{Post}, \emptyset, \{\texttt{Message}\}), (\texttt{Comment}, \emptyset, \{\texttt{Message}\})\}, \\
\mathcal{NT}' &:= \{(\texttt{Person}), (\texttt{Post}), (\texttt{Comment})\}, \\
\mathcal{ET}' &:= \mathcal{ET}.
\end{aligned}
$$

The types `Post` *and* `Comment` *are added to the set of element types together with the corresponding node types.*

Now, let us perform another schema update given by the following SMO "`DROP PROPERTY browserUsed FROM Comment`*". The updated property graph type is given by the tuple* $(\mathcal{BT}'', \mathcal{NT}'', \mathcal{ET}'')$*, where*

$$
\begin{aligned}
\mathcal{BT}'' := \{ & \\
& (\texttt{Person}, \{(\texttt{firstName}, \texttt{STRING}), (\texttt{lastName}, \texttt{STRING})\}, \emptyset) \\
& (\texttt{Message}, \{(\texttt{creationDate}, \texttt{TIMESTAMP})\}, \emptyset), \\
& (\texttt{Post}, \{(\texttt{browserUsed}, \texttt{STRING})\}, \{\texttt{Message}\}), \\
& (\texttt{Comment}, \emptyset, \{\texttt{Message}\}), \\
& (\texttt{KNOWS}, \{(\texttt{creationDate}, \texttt{TIMESTAMP})\}, \emptyset), \\
& (\texttt{LIKES}, \{(\texttt{creationDate}, \texttt{TIMESTAMP})\}, \emptyset), \\
& (\texttt{HAS\_CREATOR}, \{(\texttt{creationDate}, \texttt{TIMESTAMP})\}, \emptyset), \\
& (\texttt{REPLY\_OF}, \emptyset, \emptyset)\}, \\
\mathcal{NT}'' := \mathcal{NT}', & \\
\mathcal{ET}'' := \mathcal{ET}. & \}
\end{aligned}
$$

This operation removes the property `browserUsed` *from the node type* `Comment`*. As a result, this property is also dropped from the element type* `Message`*. In order to preserve this property for the element type* `Post`*, it is explicitly added to the set of its properties.*

3.3 Discussions and conclusions

In this chapter we have formulated the notion of a prescriptive schema for the PG data model. Such a schema allows defining types for nodes and relationships of a PG. Moreover, it allows specifying sets of allowed properties for different graph elements. Schemas for PGs are formulated as PG and are related to instance graphs by homomorphisms that preserve edges and properties.

We have also shown how both data and schema PGs can be updated using the SqPO rewriting approach, and illustrated how the techniques of backward and forward propagation in hierarchies of graphs can be used for data/schema co-evolution.

In addition, we have proposed a concise DDL that can be used to define PG schema and a syntax for various SMO operations, such as creation of new element types, joining or splitting of element types, etc.

The two database management scenarios have been described, for performing prescriptive and descriptive updates. The first scenario arises when the user is interested in performing a schema evolution that would automatically propagate to the data instances, while the second one—when the user performs some update of the data that should be reflected in the underlying schema.

The prototype system for creating and manipulating schema-aware PGs based on the Neo4j database is implemented as a part of the `ReGraph` Python library (its API is provided by the `TypedNeo4jGraph` data structure).

3.3.1 Future work

The current definition a PG schema and schema validation allows us to express only *optional* properties of graph elements. However, in a lot of traditional data modelling scenarios it is desirable to define *mandatory* properties. The possible future directions for this line of work could,

therefore, include incorporating the notion of such mandatory properties into PG schemas. This would allow us to fix a set of properties that various graph elements are required to have.

Moreover, it would be interesting to combine graph constraints provided in, for example, Neo4j with the designed notion of the schema and the mechanisms for schema validation, as well as integrate the respective language for their definition into the presented DDL.

Another interesting direction for future work could consist in using three-level hierarchies of the form $G \to V \to S$ for representing modifiable graph *views*, virtual database objects defined by queries and used for summarizing data, combining and formulating complex queries.

Due to the fact that PGs were originally designed to be schema-free, a great number of datasets have been collected and stored in a schema-less representation. Therefore, automated *schema inference* techniques could be an indispensable tool for rendering such datasets schema-aware. One possible approach to such inference would be to start from a trivial schema consisting of a single node type together with a loop edge connected to this node, and evolve this schema by consecutively splitting its nodes and edges into more and more refined concepts. Another possible orthogonal solution would be to start from a schema that trivially reflects the data (i.e. is isomorphic to it) and gradually identify and join the concepts of the same kind.

Finally, even though a prototype system for schema-aware PG has been implemented, native support for schemas, SMO and schema-data co/evolution remains to be implemented in the PG database technologies, such as Neo4j.

Chapter 4

The bio-curation framework KAMI

The bio-curation framework KAMI, developed as part of this thesis, aims to decouple the process of knowledge curation from model building. Such decoupling has proven itself indispensable for building models of complex systems of cellular signalling. The framework provides means for semi-automatic aggregation of knowledge corpora from individual PPIs, reuse of these corpora in different cellular contexts (such as different cell types, mutant cells, etc.) and automated generation of executable dynamic rule-based models.

In Section 4.1 we first present the developed de-contextualized KR given by an instance of a fixed hierarchy of simple graphs with attributes discussed in Chapter 2. We then describe, in Section 4.2, the mechanism of knowledge aggregation that assembles individual PPIs into coherent knowledge corpora which exploits the technique of forward propagation in graph hierarchies. Next, in Section 4.3 we discuss the question of knowledge instantiation, i.e. how aggregated knowledge can be reused for building different signalling models, based on backward propagation. Finally, in Section 4.4, we present the technique for generation of executable rule-based models incorporated as a part of KAMI.

The described framework is implemented in the `KAMI` Python library (see Section 4.5) and a standalone bio-curation environment `KAMIStudio` (see Section 4.6).

The bio-curation framework KAMI described in this chapter was presented as a conference paper in [50] and as a longer journal paper in [49]. Moreover, the bio-curation environment `KAMIStudio` was communicated as a separate tool paper in [51].

4.1 Knowledge representation

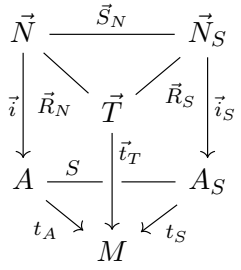
The KR provided by KAMI makes an attempt to *de-contextualize* knowledge about PPIs taking part in cellular signalling. Such de-contextualization consists in seeking to represent not the actual interactions occurring between different concrete molecules, but rather the *minimal requirements* for various interaction mechanisms to be realized. Such minimal requirements vary from purely structural, such as presence or absence of specific protein domains or key residues, to phenomenological, such as activation of proteins or their functional sites.

This de-contextualization can be achieved by abstracting from the notion of a protein to the notion of a *protoform* as the agent of a PPI. A protoform does not represent a concrete molecule, but a set of all product molecules that can be realized from a particular gene (as the result of translation and various PTMs). Therefore, an agent of interaction in KAMI represents *constraints* on a neighbourhood in the sequence space of a gene (e.g. splice variants

and mutants) together with all the combinations of PTMs (e.g. phosphorylation of residues) and phenomenological states (e.g. activity). This implies that KAMI represents knowledge on *potential* individual PPIs that can be realized (or not) in different cellular contexts and allows KAMI to reuse the same knowledge corpus for generation of models for these different contexts. The question of such reuse, or instantiation, is addressed in Section 4.3.

Thus, KAMI distinguishes two types of knowledge bodies: a knowledge corpus and a model. Corpora contain de-contextualized knowledge: agents of interactions are protoforms and the regions, residues and states associated to protoforms define their feasible sets of variants. Interactions in a corpus represent potential interactions and the necessary conditions for them to occur. Models, on the other hand, contain knowledge instantiated in given contexts: agents are concrete proteins and interactions describe rules for concrete PPIs.

A knowledge corpus in KAMI is defined as a hierarchy of graphs with attributes that has the shape of the following diagram, where graph homomorphisms are represented with arrows and graph relations (discussioned in 2.3.3) with dashes:



(4.1)

Objects and arrows of this hierarchy represent the following knowledge components:

- **Built-in components** that include:
  - The *meta-model* graph M defines the kinds of entities that can exist in a system.
  - The collection of graphs $\vec{T}$ represents *interaction templates*, their nodes define the roles of entities and actions in PPIs, e.g. enzymes, substrates, binding sites. Nodes of a template graph are typed by nodes in the meta-model M through the homomorphisms $\vec{t_T} : \vec{T} \to M$.
- **User-defined components** that include:
  - The graph A, called *action graph*, represents a global roadmap containing the 'anatomy' of protoforms, their states, PTMs and all potential interactions present in the knowledge corpus. Every node of the action graph is typed by a node in the meta-model M through the homomorphism $t_A : A \to M$.
  - The collection of graphs $\vec{N}$, called *nuggets*, encodes rules for PPIs, it specifies the necessary conditions for interactions between different protoforms. All the nugget graphs are mapped to the action graph through the collection of homomorphisms $\vec{i} : \vec{N} \to A$. These homomorphisms identify entities and actions represented in different nuggets with entities and interactions in the action graph. The collection of relations $\vec{R}_N$ between nuggets and templates assigns the roles of entities and actions in the PPIs expressed with nuggets.

- **Background knowledge components** inluding:

  - The graph A_S, called *semantic action graph*, represents a roadmap containing background knowledge on kinds of conserved protein domains and their generic interaction mechanisms. As in the case of the action graph, every node of A_S is typed by the meta-model through the homomorphism $t_S : A_S \to M$. The relation between A and A_S given by S associates entities and actions present in the action graph with their semantics in A_S.
  - The collection of graphs $\vec{N}_S$, called *semantic nuggets*, encodes individual semantic PPI mechanisms of conserved protein regions. The associated collection of arrows $\vec{i}_S : \vec{N}_S \to A_S$ identifies entities and actions from different semantic nuggets inside the semantic action graph. Moreover, every node and action of a semantic nugget is assigned with a role in a PPI with the set of relations $\vec{R}_S$. Finally, the collection of relations $\vec{S}_N$ associates entities and actions in nuggets to their semantics.

KAMI allows the curator to accommodate knowledge about different variants of proteins (for instance, slice variants or mutants)—*protein definitions*. Such protein definitions can be used to specify the 'anatomy' of variants, for example the loss of functional sites or amino acid replacements. Protein definitions are used in the process of instantiation of *concrete signalling models* from a knowledge corpus. As the result of such instantiation, some of the potential PPIs present in the corpus are not realized. An instantiated KAMI model is a graph hierarchy $I[\vec{N}] \to I[\vec{i}] \to I[A] \to I[t] \to I[M]$, where $I[M]$ is the instantiated meta-model, $I[A]$ is the instantiated action graph and $I[\vec{N}]$ is the collection of instantiated nuggets.

In the rest of this section we discuss in more detail the presented components of KAMI's corpora and protein definitions. The main constituents of instantiated models in KAMI, i.e. instantiated action graph and instantiated nuggets, are discussed in Section 4.3.

4.1.1 Meta-model

The meta-model defines the *kinds* of entities and actions that can be represented in KAMI's corpora (Figure 4.1). It is designed in a way that allows the expression of a wide spectrum of mechanistic details on PPIs and the conceptualization of the notion of an interaction mechanism. The meta-model defines a domain-specific 'syntax' for graphs representing knowledge in KAMI, i.e. all graphs are required to be *homomorphic* to the meta-model. Effectively, it defines the set of all allowed relationships between different kinds of entities, as well as the attributes and the attribute values for entities and relationships.

As we have previously discussed, *protoform* entities are meant for the representation of molecules in neighbourhoods of the sequence spaces of different genes. Protoform nodes can be equipped with attributes providing meta-data for the reference genes (such as UniProt Accession numbers[1], HGNC symbols[2]). Such meta-data helps to identify and disambiguate the agents of PPIs during the aggregation process (see Section 4.2). KAMI further allows to associate various structural and functional elements to protoforms, for example, regions, functional sites, residues and states. Elements associated to a protoform define constraints on its sequence space or its PTMs.

[1] `https://www.uniprot.org/help/accession_numbers`
[2] `https://www.genenames.org/`

Region nodes in KAMI can be used to express requirements on the presence of particular conserved protein domains of protoforms. Such nodes can be equipped with attributes expressing meta-data associated to the reference protein domain (e.g. `name` for the name of the domain, `interproid` for the InterPro identifier[3], etc.). The location of a region in the gene sequence of the associated protoform can be encoded within the edge from the region to the protoform, i.e. the modeller can specify the integer-valued attributes `start` and `end` of the sequence interval. In a similar way, KAMI allows to represent requirements on the presence of small functional sites using *site* nodes.

Protoform, region and site nodes can be equipped with key residue constraints expressed with *residue* nodes connected with an edge. The attributes of residue nodes allow to set the associated amino acid (the attribute `aa` taking values in the set of single-letter amino acid codes) and the boolean-valued attribute `test` that specifies whether the residue constraint tests the presence or absence of the key residue. The location of the residue in the gene sequence can be specified using the corresponding edge attribute `loc`.

The *state* nodes in the meta-model allow to attach modifiable on/off states to protoforms and their structural elements. Such states may represent, for example, physical states (PTMs such as phosphorylation, methylation, etc.) or phenomenological states (such as activity). State nodes are equipped with the attribute `name` that allows for identification of the state, and the boolean-valued `test` indicating whether the associated constraint expressed by attaching the state to the corresponding entity is required to be on or off. For example, we can express the requirement for a structural element to be unphosphorylated by attaching a state node with the attributes {`name: phosphorylation, test: False`}.

The meta-model of KAMI allows the accommodation of two kinds of *action* nodes: *binding* (BND) and *modification* (MOD) nodes. These nodes are designed to represent explicitly and, therefore, conceptualize the notion of an *interaction mechanism* (see a more detailed discussion in Subsection 4.1.3). Actors of interactions are connected to the corresponding action nodes with edges. For example, the meta-model allows for protoform, region and site nodes to be actors of a binding interaction (see the edges from the protoform, region and site nodes to the BND node in Figure 4.1). On the other hand, only protoform and region nodes are allowed to be used as actors of a modification interaction. Moreover, MOD nodes are connected with an outgoing edge to the target of modification—a state node they act upon (whose value they change). Both interaction nodes can be equipped with the float-valued attribute `rate` representing the *interaction rate constant*[4].

BND nodes can be used to express both binding and unbinding interactions. Namely, the boolean-valued attribute `test` specifies whether the node represents a binding or an unbinding. Furthermore, BND nodes in KAMI can be used not only to represent an action performed by some agent, they can also express a constraint on agents to *be* bound. The attribute `type` can take the values `do` or `be`, representing an action and a constraint respectively. On the other hand, MOD nodes always represent an action and are equipped with the boolean-valued attribute `value`, which specifies the value of the modifiable state it targets.

[3] `https://www.ebi.ac.uk/interpro/`

[4] Rates in KAMI correspond to rate constants in the Kappa language [26], see *P. Boutillier, J. Feret, J. Krivine, and W. Fontana. The Kappa Language and Tools* (version of November 27, 2019, `kappalanguage.org`) for more discussion on how such rate constants are computed.

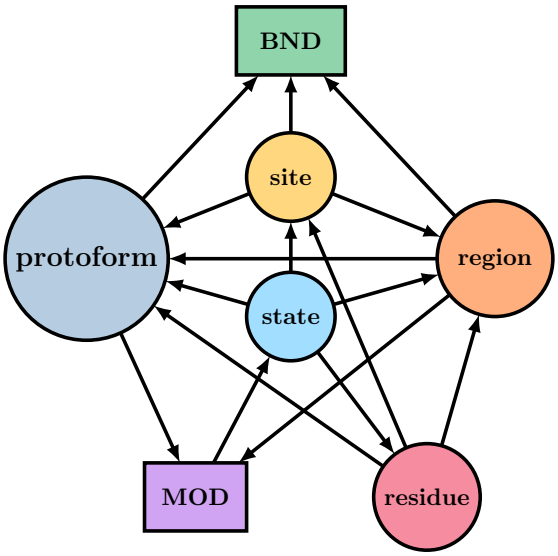


Figure 4.1: The meta-model of KAMI. Nodes representing actions are denoted with rectangles, attributes on nodes and edges are omitted.

4.1.2 Interaction templates

Interaction templates represent small built-in graphs that define the roles of entities and actions in PPIs. KAMI contains two interaction templates corresponding to binding (see Figure 4.2) and modification (see Figure 4.3) interactions. The binding template is essentially symmetric and specifies the roles of actors of a binding interaction. For example, a binding can be performed directly by a protoform or through one of its regions or sites. The second KAMI template specified the roles of actors in a modification interaction. An action can be performed directly by a protoform or one of its enzymatic regions. The MOD node always targets a state node that can belong to a protoform or any of its structural elements. Establishing relations between nuggets (graphs representing PPIs) and the template graphs, KAMI assigns the above-mentioned roles (see a more detailed discussion in the following subsection).

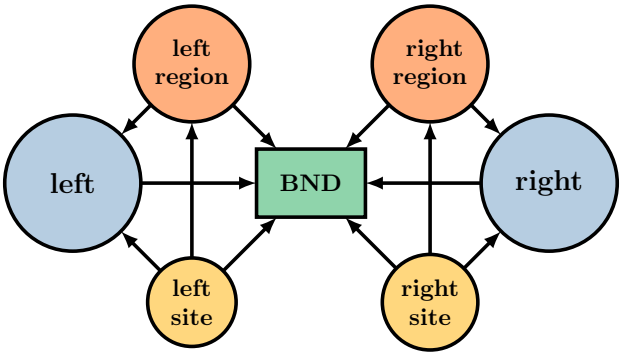


Figure 4.2: The binding template. The mapping specifying the types of nodes in the meta-model is encoded using node colors from Figure 4.1, e.g. regions are orange, residues are light-red.

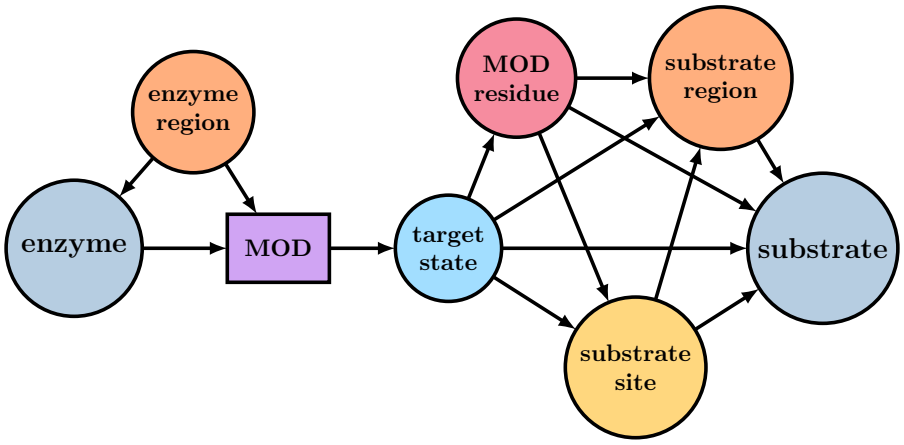


Figure 4.3: The modification template. The mapping specifying the types of nodes in the meta-model is encoded using node colors from Figure 4.1, e.g. regions are orange, residues are light-red.

4.1.3 Nuggets and the action graph

Up until now we have seen two built-in components of KAMI: its meta-model and its interaction templates. In this subsection we describe how nugget graphs and the action graph are used for representation of user-provided knowledge in KAMI corpora and models.

As we have previously described, the *action graph* specifies a global roadmap of all *actual* entities and actions that are present in a KAMI corpora (the same holds for the instantiated action graph of a KAMI model). More precisely, it represents all the protoforms, components and interaction mechanisms mentioned in a given corpora. The action graph is homomorphic to the meta-model, i.e. every node represents an instance of an entity or an action defined by the meta-model, all the edges and attributes in the action graph are preserved in the meta-model.

Example 4.1.1. *Figure 4.4 illustrates an example of action graph. As before, the mapping specifying the types of nodes in the meta-model is encoded using node colors from Figure 4.1. Nodes are labeled for the sake of readability and the labels do not carry any semantics (semantics is encoded within node and edge attributes). The underlying corpus contains knowledge about three different protoforms corresponding to the genes EGFR, SHC1 and GRB2. The graph contains the 'anatomy' of these protoforms, e.g. some of their conserved domains, sites and residues, as well as some binding and modification actions that these protoforms can perform.*

The actual rules for PPIs in KAMI are represented using *nuggets*, small graphs homomorphic to the action graph. Essentially, the action graph specifies an *evolvable schema* for nuggets, while the meta-model defines a *fixed schema* for all the graphs in a KAMI corpus or a model [49]. The homomorphisms mapping different nuggets to the action graph encode relations between these nuggets. More precisely, if two different entities or actions from different (or the same) nuggets map to the same node in the action graph, they are considered to represent "the same" entity or action. It is important to note that two action nodes mapping to the same node in the action graph are identified as the same interaction mechanism in KAMI. By allowing such identification, KAMI conceptualizes interaction mechanisms. Then, PPIs representing instances of the same interaction mechanism are considered to be *conflicting*, i.e. they cannot happen at the same time as, supposedly, they use the same resource needed for the interaction mechanism

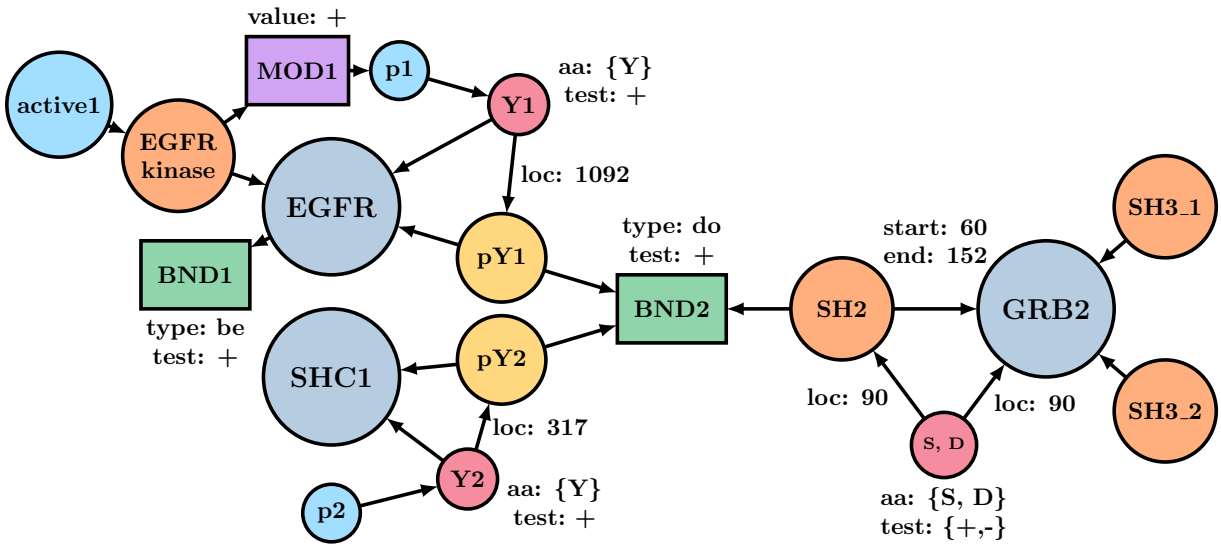


Figure 4.4: Example of an action graph.

to be realized (such as presence of a free binding site or a conserved domain). Example 4.1.2 provides some nuggets and illustrates how they can be used to encode rules for PPIs and how their homomorphisms to the action graph provide the identification relation between them.

Example 4.1.2. *Consider graphs in Figures 4.5a, 4.5b and 4.5c. The mapping of nodes to the meta-model (factoring through the action graph) is encoded using node colors from Figure 4.1. As in the previous example, nodes are labeled for the sake of readability. Moreover, in this example, they encode the mapping to the nodes of the action graph from Figure 4.4 (nodes from the nuggets map to the action graph nodes with the same label).*

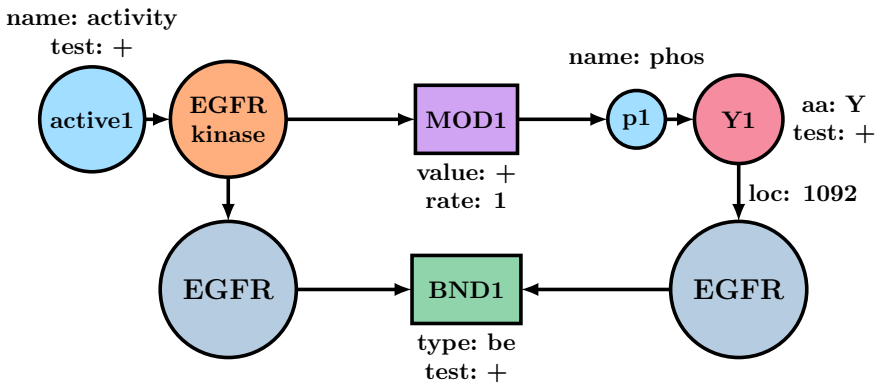
The illustrated nuggets represent the following three respective statements:

1. *"A protein product of EGFR can phosphorylate residue Y1092 of another EGFR molecule through its active kinase domain, when the two molecules are bound."*

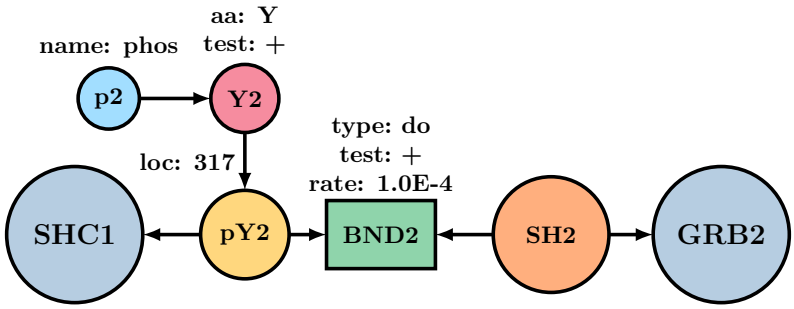
2. *"A protein product of SHC1 can bind to the SH2 domain of a GRB2 protein through its site pY having the residue Y317 phosphorylated."*

3. *"A protein product of EGFR can bind to the SH2 domain of a GRB2 protein through its site pY having the residue Y1092 phosphorylated. This interaction happens when the SH2 domain of GRB2 has the key residue S90, but not D90."*

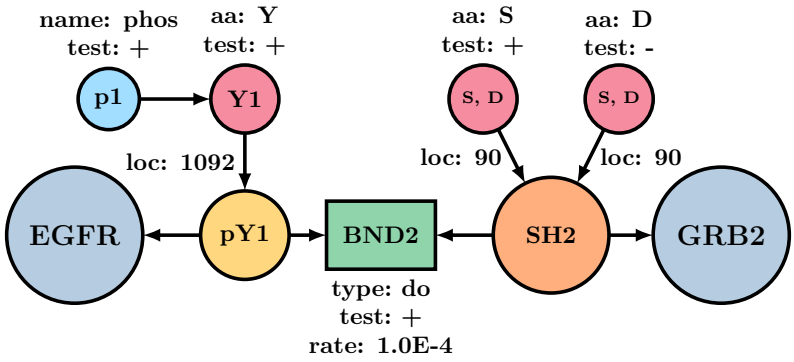
Mapping of the nugget nodes to the action graph allows us to identify the entities and actions represented by nuggets. For example, the two protoforms taking part in the interaction described in the first nugget are instances of the same reference protoform corresponding to the EGFR gene. Likewise, one of the protoforms performing the binding described in the third nugget is also an instance of the same reference EGFR protoform. Interestingly, the two residue nodes attached to the SH2 domain in the third nugget map to the same action graph node representing a key residue of GRB2 at the location 90, but have different values of amino acid and different test semantics. The nugget represents knowledge on a positive requirement *for the amino acid S to be present at the location 90, i.e. we* know *that, if it is the case, the described interaction*



(a) Nugget 1.



(b) Nugget 2.



(c) Nugget 3.

Figure 4.5: Example of nuggets.

can appear. It also represents knowledge on a negative requirement *for the amino acid D to be present at this location, i.e. we* know *that, if it is the case, the interaction cannot appear. As for the rest of the possible amino acids at this location, we* do not know *if they enable or prevent this interaction from happening.*

Note that the BND nodes in the second and the third nuggets map to the same node in the action graph. This indicates that the two interactions described by these nuggets represent instances of the same interaction mechanism*. Namely, the binding mechanism of the SH2 domain of GRB2. This also implies that the two interactions are* conflicting*, i.e. when one of the interactions appears, the second one cannot be realized as the resources needed (a free SH2 domain) for such an interaction are no longer available.*

Every nugget in a KAMI corpus (or a model) is related to at least one of the interaction templates, depending on the kind of interaction it represents. As we have previously mentioned, such relations allow us to assign roles to nugget nodes in the interaction. For example, the nugget in Figure 4.5a is related to both binding and modification interaction templates. The two EGFR nodes are associated with the roles `left` and `right` in the binding template, then the left EGFR node is also associated with the `enzyme` node in the modification template, while the right EGFR—with the `substrate` node.

4.1.4 Semantic background knowledge

So far, in the examples we have presented, we have referred to some states as 'phosphorylation' or 'activity', their representation is purely syntactic and does not carry any *semantics*. It was up to us to interpret the state node with the attributes {`name: phos`} or {`name: activity`} as phosphorylation or activity respectively. Due to the fact that the main functional units of the majority of PPIs are conserved protein domains, we can also 'hard-wire' background knowledge on the semantics of generic interaction mechanisms of these domains. To assign semantics to the represented knowledge, KAMI provides two built-in components to its corpora: the *semantic action graph* (SAG) and *semantic nuggets* (SNs).

Similarly to the action graph, the SAG represents a roadmap of semantic entities and actions that can exist in a KAMI corpora and is homomorphic to the meta-model (see Figure 4.6). At the current stage KAMI is equipped with the interaction semantics corresponding to the *protein kinase*, *phosphatase* and *SH2* domains. The semantic entities encoded in the SAG include the above-mentioned domains, pY-binding sites, activity and phosphorylation states. The semantic actions described in the SAG from Figure 4.6 represent phosphorylation (PHOS), dephosphorylation (DEPHOS) and SH2-pY binding (SH2/pY BND) interaction mechanisms. KAMI assigns the described semantics to the entities and actions from the action graph by establishing a relation between the nodes of the SAG and the action graph.

For *(de-)phosphorylation semantics* KAMI adopts the following two conventions reflected in the SAG in Figure 4.6: (1) a (de-)phosphorylation is always performed by a protein kinase (phosphatase) region, (2) to be able to exhibit its enzymatic activity the protein kinase (phosphatase) region is required to be activated and (3) there can be at most one (de-)phosphorylation action associated with a single protein kinase (phosphatase) region in the action graph. The last constraint represents the fact that there is a unique mechanism through which every protein kinase (phosphatase) region performs (de-)phosphorylation, and it can be further interpreted as a conflict between every two individual phosphorylation interactions performed by the same region, i.e. in a real system a protein kinase region can perform only one phosphorylation at a time.

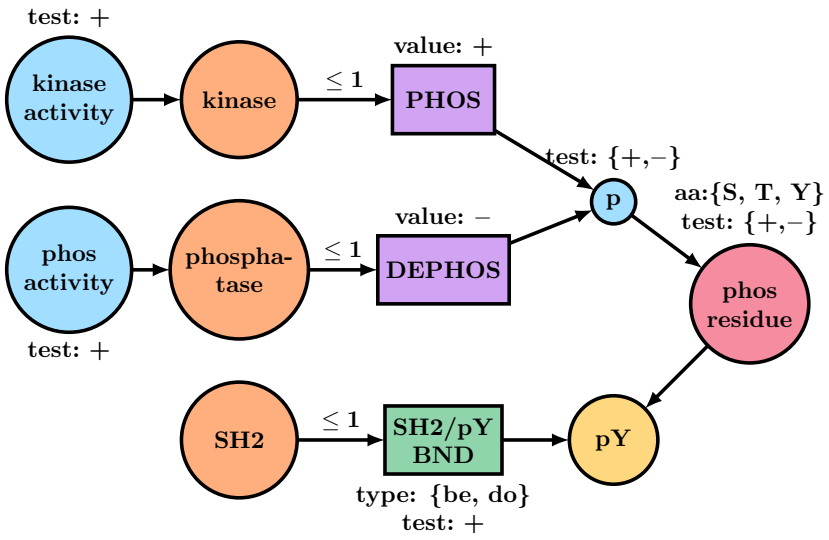


Figure 4.6: Semantic action graph

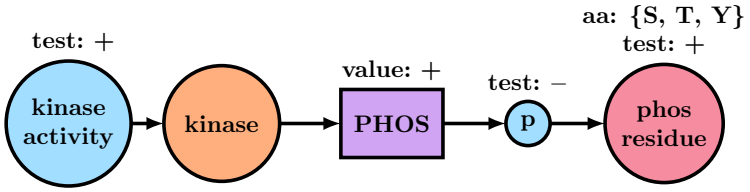
Similarly to the phosphorylation semantics we adopt some conventions concerning *SH2/pY binding semantics* reflected in the SAG in Figure 4.6: (1) an SH2 domain always binds to a small peptide motif, *pY site*, which contains a phosphorylated Y residue and (2) there can be at most one binding action associated with a single SH2 region in the action graph. Again, the last constraint represents the fact that there is a unique mechanism through which every SH2 domain binds, and it can be further interpreted as a conflict between every two individual bindings of the same SH2 domain to different pY sites, i.e. an SH2 domain binds to one pY site at the time.

SNs represent minimal requirements on generic interaction mechanisms of semantic entities to be realized (see Figure 4.7), e.g. phosphorylation of S, T or Y residues by a protein kinase domain, binding of an SH2 domain to sites containing phosphorylated Y residues. The SNs in KAMI are homomorphic to the SAG and are used to assign semantics to the potential PPIs represented with nuggets.

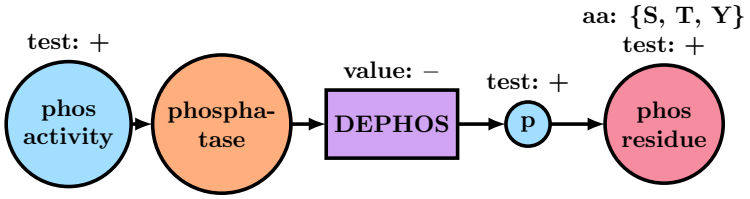
Section 4.2 describes how KAMI establishes the above-mentioned semantic relations in an automated fashion, and how this semantics helps to perform nugget autocompletion and identification of interaction mechanisms.

4.1.5 Protein definitions

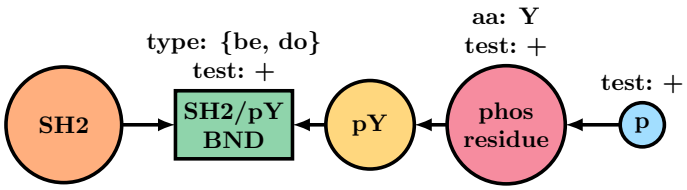
As we have previously described, the agents of potential PPIs in KAMI corpora represent neighbourhoods in the sequence spaces of different genes. KAMI allows to create protein definitions that specify how the 'anatomy' of a given protoform gives rise to different protein products (e.g. splice variants and mutants). Protein definitions in KAMI are represented with *restrictive SqPO rules* (i.e. arrows of the form $L \leftarrow P$) and their instances in the action graph. An example of a rule encoding the protein definitions for the protoform GRB2 is illustrated in Figure 4.8. A more detailed discussion on how such protein definitions can be used to perform model instantiation in KAMI can be found in Section 4.3.



(a) Phosphorylation semantics.



(b) Dephosphorylation semantics.



(c) SH2-pY binding semantics.

Figure 4.7: Semantic nuggets.

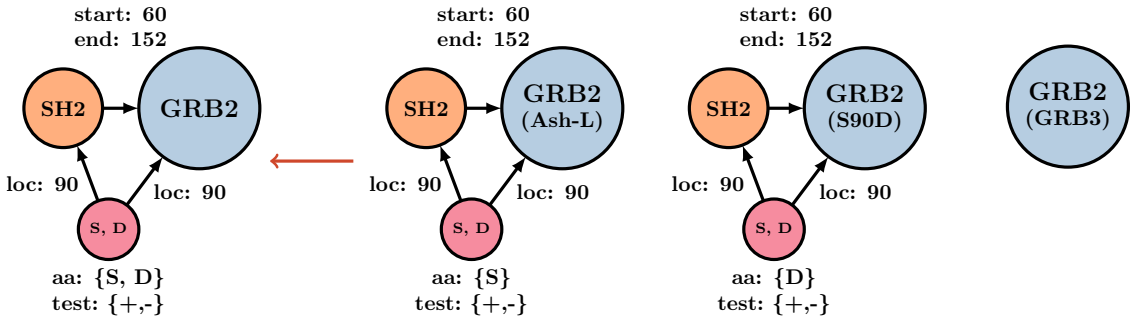


Figure 4.8: Protein definition for GRB2. GRB2 gives rise to three proteins: the wild type of GRB2 (Ash-L), the mutant with the key residue S90 replaced by D90 (S90D), the splice variant with the knock-out of the SH2 domain (Grb3). The matching of the left-hand side of the rule in the action graph is given by the correspondance between the node labels.

4.2 Knowledge aggregation

In this section we describe the mechanism for automatic aggregation of individual PPIs used in the KAMI framework. This mechanism exploits the techniques for rewriting and forward propagation in hierarchies of graphs discussed in Section 2.2. In addition, it incorporates domain-specific knowledge on generic mechanisms for PPIs that allows it to perform more sophisticated and biologically meaningful aggregation.

KAMI proposes an intermediate representation language for the input of knowledge to the aggregation process. This language provides means for representation of interactions and agents of interactions in a user-friendly and concise manner. Moreover, it frees the user from the necessity to learn the graph-based knowledge representation syntax internal to KAMI, serves the validation purpose and allows KAMI's aggregation engine to perform identification of interactions and entities on the fly during the nugget graph generation process.

Provided an individual PPI expressed in the intermediate representation language, the adopted technique aggregates new knowledge into the underlying corpus in a context-dependent fashion performing the following sequence of steps:

1. generation of a nugget graph;
2. identification of entities and actions already present in the action graph;
3. addition of the nugget to the corpus and propagation of new knowledge to the action graph;
4. bookkeeping updates such as anatomization of new genes, reconnection of spatially nested components;
5. updates specific to semantics of the provided PPI.

In the rest of this section we discuss KAMI's intermediate representation language and the aggregation steps in more detail.

4.2.1 Intermediate representation language

KAMI's intermediate language allows the representation of PPIs, their actors and actors' components (such as regions, sites, residues etc). This language coincides with the Python syntax for calls to the constructors of user-defined classes (i.e. see Section 4.5 for more details on the implementation of the Python API for KAMI).

Entities from KAMI's meta-model are represented with their corresponding classes `Protoform`, `Region`, `Site`, `Residue` and `State`. The `Protoform` class allows encapsulating collections of regions, sites, residues and states associated to the intended protoform definition. Similarly, the `Region` class encapsulates sites, residues and states; `Site`, residues and states; and `Residue`, one state. Consider the following example illustrating the use of the intermediate language for representation of a protoform.

Example 4.2.1. *The following listing represents the entity corresponding to the statement "The protoform ABL1 that has active protein kinase domain".*

```
abl1 = Protoform(
  uniprotid="P00519",
  hgnc_symbol="ABL1",
  regions=[
    Region(
      name="Protein kinase",
      interproid="IPR000719",
      start=242, end=493,
      states=[State("activity", True)])])
```

According to KAMI's meta-model, not only protoforms, but also their regions and sites, can be the actors of PPIs. To express such region and site actors explicitly, the intermediate language provides two constructions `RegionActor` and `SiteActor`. Consider the following example defining a region actor.

Example 4.2.2. *The following listing represents the actor corresponding to the statement "The active protein kinase domain of the protoform ABL1".*

```
abl1_PK = RegionActor(
  protoform=Protoform(
    uniprotid="P00519",
    hgnc_symbol="ABL1"),
  region=Region(name="Protein kinase",
    interproid="IPR000719",
    start=242, end=493,
    states=[State("activity", True)]))
```

Let us compare Examples 4.2.1 and 4.2.2. On the first sight the statements expressed in the two examples seem identical. However, there is a substantial difference in their interpretation. When used in a PPI, the protoform from Example 4.2.1 directly takes part in the interaction, and to be able to do that it is required to have the protein kinase region. There may be several reasons (though unknown) why it is indeed required: (a) the region may actually be performing the interaction itself, but it was *unknown* to the source of knowledge; (b) the region may be required for an *indirect* reason for the interaction to appear (for example, its knock out causes a conformational change that prevents the interaction). Meanwhile, the actor from the Example 4.2.2 is *known* to directly perform the interaction. The region may have some additional interaction semantics associated to it (known by the system) and this semantics can influence further interpretation of the interaction (more details will follow in Subsection 4.2.4). Moreover, if the same region is used as an actor of several distinct interactions, it may be an evidence of a conflict between the interactions. Examples of such conflicts can be often seen in the case of binding interactions, i.e. the same region has multiple binding partners, but we know that in a particular protein molecule this region can bind to exactly one partner at a time.

As a part of its intermediate representation language KAMI provides a format for expressing various types of interactions. As previously mentioned, actor entities play a role of building blocks of interactions. The interaction classes include `Modification`, `AnonymousModification`,

`SelfModification`, `Binding` and `LigandModification`. Every such interaction can be provided with an interaction rate (bi- or unimolecular).

Modification. A modification interaction in KAMI is a high-level PPI, where one protein molecule (enzyme) modifies some state of another protein molecule (substrate). It is high-level in the sense that it represents a sequence of several more basic biochemical reactions underlying it, e.g. binding of the substrate to the enzyme, a biochemical reaction as a result of which the substrate's state changes, unbinding of the enzyme from the substrate. Knowledge about a modification interaction in KAMI can be expressed by providing an enzyme, a substrate, a modification target and a modification value. The enzyme and substrate are actor entities, a modification target can be a state or a residue with a state, and finally the modification value is the one assigned to the target state as the result of modification. Consider the following example of a modification interaction.

Example 4.2.3. *The following listing represents the interaction corresponding to the statement "The active protein kinse domain of ABL1 phosphorylates the residue Y394 of PLCG1", where we reuse the actor `abl1_PK` defined in Example 4.2.2.*

```
1 Modification(
2   enzyme=abl1_PK,
3   substrate=Protoform(
4     "P00533", hgnc_symbol="EGFR"),
5   target=Residue("Y", 1173, State("phosphorylation", False)),
6   value=True,
7   rate=1.0E-4)
```

Self-modification. Self-modification in KAMI is a modification interaction where a single molecule modifies its own state (it is at the same time the enzyme and the substrate). An instance of such an interaction is defined by an enzyme actor, optionally its substrate region or site (if it is known that the modification takes place in a particular region or site), modification target and state. Consider the following example of a self-modification interaction expressed with the intermediate language.

Example 4.2.4. *The following listing represents the interaction corresponding to the statement "FGFR1 phosphorylates its residue Y583 through the protein kinase region".*

```
1 SelfModification(
2   enzyme=RegionActor(
3     protoform=Protoform("P11362", hgnc_symbol="FGFR1"),
4     region=Region(name="Protein kinase")),
5   target=Residue("Y", 583, State("phosphorylation", False)),
6   value=True,
7   rate=0.1)
8
```

Anonymous modification. Anonymous modification in KAMI is a modification where a substrate undergoes a state change without an explicitly known enzyme. An instance of such an interaction is defined by a substrate actor, a target and a value of modification as before. Anonymous modifications can be used to express more detailed mechanisms that underlie phenomenological states, for example, that a protein becomes active if some particular residues are phosrphorylated. Consider the following example of an anonymous modification interaction expressed with the intermediate language.

Example 4.2.5. *The following listing represents the interaction corresponding to the statement "RAF1 is activated when its residues S621 and T268 are phosphorylated".*

```
AnonymousModification(
  substrate=Protoform(
    "P04049", hgnc_symbol="RAF1",
    residues=[
      Residue("S", 621, State(name="phosphorylation", value=True)),
      Residue("T", 268, State(name="phosphorylation", value=True))]),
  target=State("activity", False),
  value=True,
  rate=1)
```

Binding. A binding interaction in KAMI is a PPI that arises from the formation of stable non-covalent bonds between a protein molecule and its ligand. Combinations of such individual bindings govern the assembly of large protein complexes. An instance of a binding can be expressed by defining the two actor entities (called the 'left' and the 'right' binding partner to 'break the symmetry' of the protein-ligand relation). Consider the following example of a binding interaction expressed with the intermediate language.

Example 4.2.6. *The following listing represents the interaction corresponding to the statement "A protein product of AXL can bind to the SH2 domain of a GRB2 protein through its site pY having the residue Y821 phosphorylated".*

```
axl_pY = SiteActor(
  protoform=Protoform("P30530", hgnc_symbol="AXL"),
  site=Site(
    name="pY",
    residues=[
      Residue("Y", 821, State("phosphorylation", True)])))

grb2_sh2 = RegionActor(
  protoform=Protoform("P62993", hgnc_symbol="GRB2"),
  region=Region("SH2"))

Binding(axl_pY, grb2_sh2)
```

Ligand modification. An interaction of ligand modification in KAMI is a modification that requires the enzyme to be bound to the substrate in order for the interaction to happen. As in the case of any modification interaction, ligand modification can be expressed by providing an enzyme, a substrate, a modification target and a modification value. In addition, KAMI allows the curator to specify the actors of binding necessary for the interaction to happen, i.e. regions or sites of the enzyme and the substrate protoform. Consider the following example of a ligand modification interaction.

Example 4.2.7. *The following listing represents the interaction corresponding to the statement "A protein product of EGFR can phosphorylate residue Y1092 of another EGFR molecule through its active kinase domain, when the two molecules are bound" (corresponding to the nugget graph in Figure 4.5a).*

```
egfr = Gene("P00533", hgnc_symbol="EGFR")

egfr_pk = RegionActor(
  gene=egfr,
  region=Region(name="Kinase", start=712, end=979))

mod = LigandModification(
  enzyme=egfr_pk,
  substrate=egfr,
  target=Residue(
    "Y", 1092, State("phosphorylation", False)),
  rate=1)
```

4.2.2 Nugget generation

The nugget generation phase KAMI (1) converts the input interaction expressed with the previously-described intermediary language to a nugget graph N; (2) finds a typing of the nugget by the meta-model $t_N : N \to M$; (3) finds relations of the nugget with the interaction templates, and (4) identifies the entities and actions already present in the action graph. While the first three generation steps are rather straightforward (can be directly inferred from the intermediary representation), the third one—the identification of nugget entities and actions in the action graph—is slightly more intricate.

Identification of entities is done by querying the action graph and is performed according to the following conventions we have adopted in KAMI:

1. *Protoforms* are identified by the UniProt accession numbers (AC) of the corresponding genes. The aggregation engine traverses the protoforms nodes present in the action graph and searches for the one with the same AC.

2. *Regions* and *sites* in KAMI can have specified name, InterPro identifier, sequence interval start and end (all optional). In addition, their intermediate representation is always encapsulated within a representation of the reference protoform. Therefore, identification of a region/site is performed, if and only if its protoform has been identified in the action graph. First of all, if a region to be identified has the sequence start and end specified, the

engine looks for an existing region with the same (or significantly overlapping) interval; if such a region was not found in the action graph, the engine tries to find it by resolving the information given by the name, the InterPro identifier of the input region.

Moreover, the intermediate input language in KAMI allows the curator to specify an integer-valued attribute called `order`. In the case when the protoform to be represented has multiple domains with the same name (for example, two SH2 domains: N-terminal and C-terminal), such an attribute allows us to identify the order of the domain in the gene sequence. For example, let us suppose that some protoform in the action graph has two regions, the region R_1 called "SH2" in the interval 123–217 and the region R_2 also called "SH2" in the interval 560–644. Now, suppose that an actor of the input interaction represents the same gene that has a region with the name "SH2". This information alone is not enough to decide whether it refers to the region R_1 or R_2 in the action graph, as this gene has two SH2 domains. On the other hand, even if the user does not have the information about the exact interval of the SH2 domain from the input interaction, but knows, for example, that this is the N-terminal SH2 domain, then they can specify the order of this region (in this case order being equal to 1).

3. *Residues* have specified amino acid codes and, optionally, locations. In interactions they are always encapsulated inside physical entities, so the genes to which they belong are always known at the identification stage. First of all, if the location of a residue is provided, KAMI searches for a residue with this location belonging to the respective gene in the action graph. If the location is unknown, the engine tries to find a residue from the pool of residues with no location in the action graph that has the same amino acid value.

4. *States* in KAMI are identified by their name, i.e. two states with the same name belonging to the same structural element (it can be gene, region, site or residue) are considered to represent the same state.

Let N be the nugget graph generated from the input interaction. The output of the identification is a *relation* $R_i \subseteq V_N \times V_A$, where V_N is the set of nodes of the generated nugget graph and V_A is the set of nodes of the action graph.

To add the generated nugget N to the corpus, the aggregation engine first adds an empty graph $N_\emptyset$ as the new nugget to the underlying hierarchy. It then creates a *nugget generation rule* $\emptyset \to N$, i.e. an expansive rule that adds all the generated nugget nodes and edges to an empty graph (here denoted $\emptyset$). This rule is applied to the newly added empty graph through the trivial instance $m : \emptyset \rightarrowtail N_\emptyset$. The induced forward propagation to the action graph is *controlled* by the relation R_i (see more details on controlled propagation in Subsection 2.2.4). On the other hand, this rewriting is strict with respect to the meta-model, as the typing of the generated nugget by the meta-model is given by the homomorphism t_N. Thus, the application of the generation rule transforms the empty graph $N_\emptyset$ into N, propagates the new bits of information (all unidentified entities and new actions) to the action graph and keeps the meta-model unchanged. The following example illustrates the nugget generation procedure.

Example 4.2.8. *Consider the interaction illustrated in Example 4.2.6. The nugget graph N generated from this interaction is presented in Figure 4.9. The typing of the nugget by the meta-model t_N is encoded using node colors from Figure 4.1.*

The relation with the binding interaction template is defined as follows:

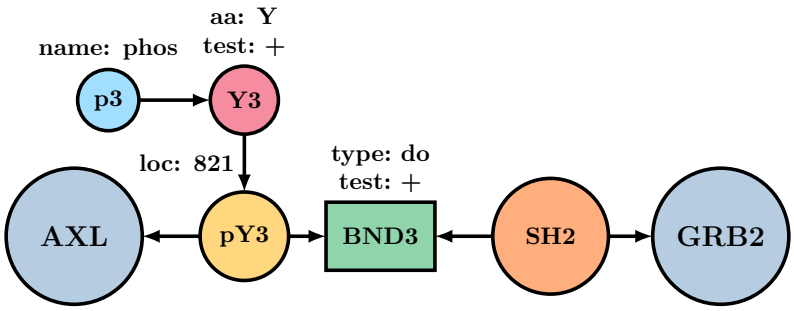


Figure 4.9: Generated nugget.

$$R_N^{BND} = \{(\texttt{AXL}, \texttt{left}), (\texttt{pY3}, \texttt{left site}), (\texttt{BND3}, \texttt{BND}), (\texttt{SH2}, \texttt{right region}), (\texttt{GRB2}, \texttt{right})\}$$

Let the action graph be given in Figure 4.4. We identify the existing entities and actions and obtain the following the identification relation $R_i = \{(\texttt{SH2}, \texttt{SH2}), (\texttt{GRB2}, \texttt{GRB2})\}$. *The updated action graph (after forward propagation of the nugget generation rule) is illustrated in Figure 4.10.*

4.2.3 Bookkeeping updates

After the update of the action graph performed in the previous aggregation stage, a series of *bookkeeping updates* is performed. Such updates include gene anatomization and reconnection of nested and transitive components. This subsection provides some details on the bookkeeping updates and the scenarios in which they are performed.

Gene anatomization. For all new protoforms added to the action graph with its update, KAMI retrieves some additional information about their reference genes. For a given gene this information includes some *meta-data* (such as HGNC symbol, synonyms, references to other databases) and the list of its *known conserved domains and functional sites* (together with their names, intervals and *InterPro* identifiers). This information is obtained from various databases including *UniProt*, *InterPro* and *Ensembl*[5]. We refer to this process as *gene anatomization*. The information about the gene's regions plays a significant role in our automated aggregation process. More specifically, the InterPro identifiers (which are often not provided by the user, but obtained at the anatomization stage) help the engine to perform semantic tagging, i.e. to identify if a given region has some known interaction semantics (e.g. protein kinase or SH2 domain semantics).

Using the information retrieved at the anatomization stage, KAMI generates an *anatomization rule*, i.e. an expansive rule, application of which augments the action graph with the new information. The rule is constructed in the following way: (1) a node corresponding to the gene is added to the left-hand side of the rule; (2) for every region found by the anatomization, if the region is identified with some existing region in the action graph, the region from the action graph is added to the left-hand side and is connected with an edge to the gene; otherwise, a

[5] `https://www.ensembl.org/`

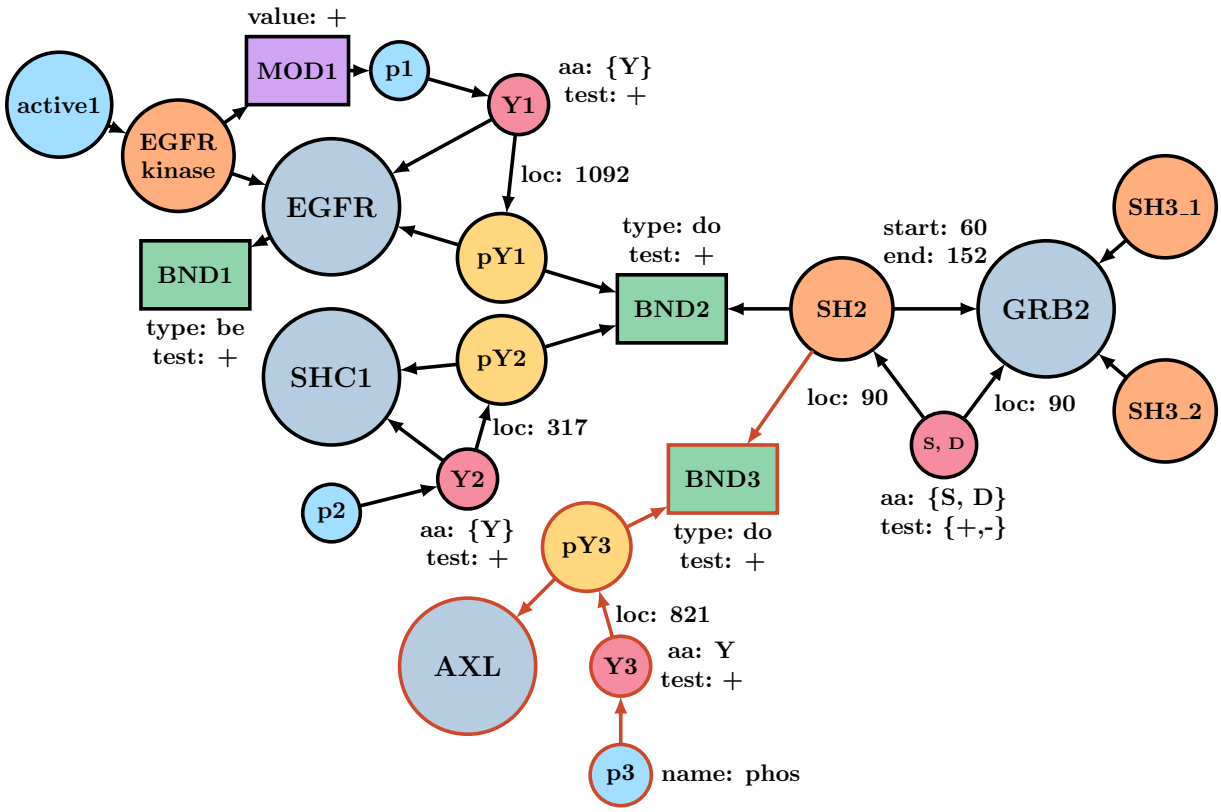


Figure 4.10: Updated action graph. New nodes and edges added to the action graph are highlighted with dark orange.

node corresponding to the region is added to the right-hand side. Example 4.2.9 illustrates an anatomization rule generated by KAMI.

Example 4.2.9. *Let us suppose that after the addition of a new nugget the protoform corresponding to the gene ABL1 was added to the action graph together with its kinase region. Figure 4.11 illustrates the gene anatomization rule generated by KAMI. This rule adds new meta-data fetched from available databases: it adds the new attribute* `Synonyms: {ABL, JTK7}` *to the protoform, adds the new attribute* `InterPro: IPR000719` *to the existing kinase region and attaches two other known conserved protein domains of ABL1.*

Reconnection of nested and transitive components. This stage performs the following updates of the action graph:

- For every new component added to the action graph (including all new regions, sites and residues), if it is equipped with information about its location in the gene's sequence (location for residues and interval for regions and sites), the engine verifies if there exists a component in the action graph which spatially includes it and, if it is the case, adds an edge between them (with an appropriate direction specified by the meta-model). For example, if the site with the interval $[n, m]$ was added and there exists a region with the interval $[k, l]$ such that $k \leq n$ and $m \leq l$, an edge between them will be added.

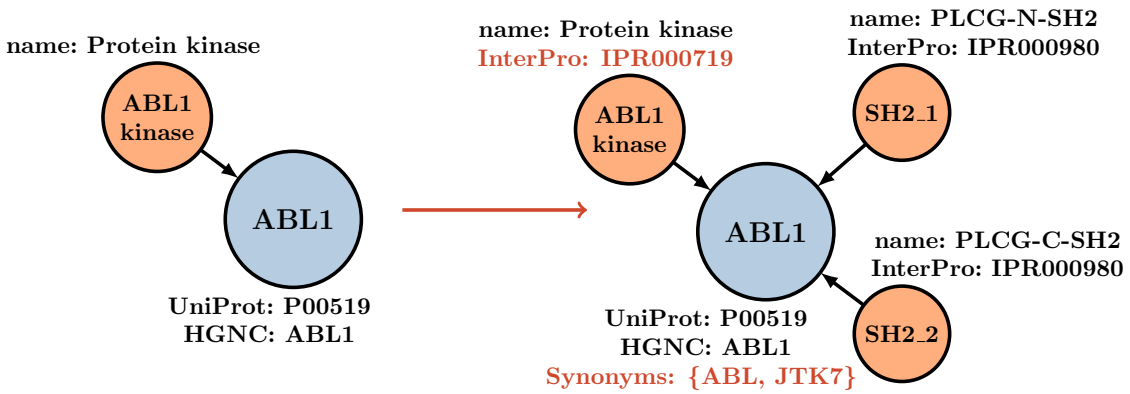


Figure 4.11: Example of a gene anatomization rule for ABL1. Added node attributes are denoted with dark orange.

- To facilitate the entity identification performed at various stages of PPI aggregation, KAMI adds explicit edges between all the structural elements which are included transitively (e.g. a residue that is included in a region of a protoform is transitively included in the protoform as well). To reconnect such components, the engine finds all matchings of specific patterns (see the figure 4.12) in the modified portion of the action graph and applies the rule that reconstructs appropriate edges.

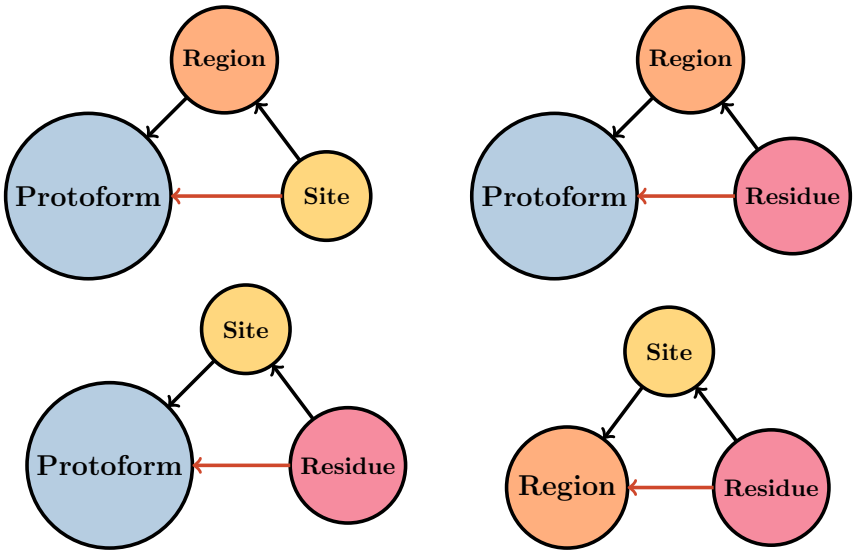


Figure 4.12: Patterns of transitively included components. The four graphs depict the patterns of transitive component inclusion, the new edge added at the bookkeeping stage is highlighted with dark orange.

4.2.4 Semantic updates

Semantic updates performed by the aggregation engine are based on generic interaction mechanisms of conserved protein domains. At the current stage of its development, KAMI is able to

perform semantic updates related to the specific interaction mechanisms of three abundant domains: the *protein kinase domain*, the *phosphatase domain* and the *SH2 domain*. This section details the updates performed by KAMI, when PPIs involving these domains are encountered. Namely, we present the semantic updates performed when a phosphorylation and an SH2/pY binding interactions are added to the corpus. The dephosphorylation semantic update is practically identical to the phosphorylation one and, therefore, omitted in the discussion.

Phosphorylation semantic update. If KAMI has identified that the newly created nugget represents a phosphorylation interaction, the corresponding semantic update of the corpus is performed. This update consists of two steps: first, the nugget is *autocompleted* with missing bits of knowledge induced by the known mechanism of phosphorylation and, if there exist multiple phosphorylation actions associated with the given protein kinase region (the one that performs the phosphorylation), these *actions are merged*. To perform the nugget autocompletion, KAMI tests if the nugget has an enzyme region specified. Depending on the outcome of this test the following updates are performed:

- If the *enzyme region is specified*: if it is typed by a node of the action graph that is semantically tagged as a protein kinase region, then the engine simply checks if the respective nugget node has the activity state node attached, and adds it if it is not the case; otherwise, KAMI warns the user that the region performing the phosphorylation is not a protein kinase region and halts the semantic update stage.

- If the *enzyme region is not specified*, KAMI searches for a unique protein kinase region of the corresponding protoform in the action graph: if it is found, then the nugget is autocompleted with a node representing this region and the requirement for this region to be active; otherwise, KAMI warns the user and halts the update.

To perform the action merge, KAMI simply looks for all modification actions associated to the protein kinase node of the action graph. If more than one action is found, all the actions are merged into a single action.

SH2/pY binding semantic update. The semantic update triggered by an identified SH2/pY binding interaction also consists of two steps: autocompletion and action merge. To perform the nugget autocompletion, KAMI tests if the SH2 binding partner has a binding site specified. If it is the case, this site is semantically tagged as a pY site, otherwise the nugget is autocompleted with pY site node. Then, multiple binding actions associated with the given SH2 domain are merged. Figure 4.13 illustrates the action graph from Figure 4.10 after the merge of the BND actions of the SH2 domain of GRB2.

4.3 Knowledge instantiation

The mechanism of knowledge instantiation allows the curator to *reuse* a knowledge corpus in different cellular contexts. Such contexts can be specified by a set of protein definitions that describe how the 'anatomy' of a given protoform gives rise to different protein products. To instantiate a concrete signalling model KAMI applies restrictive rules corresponding to different

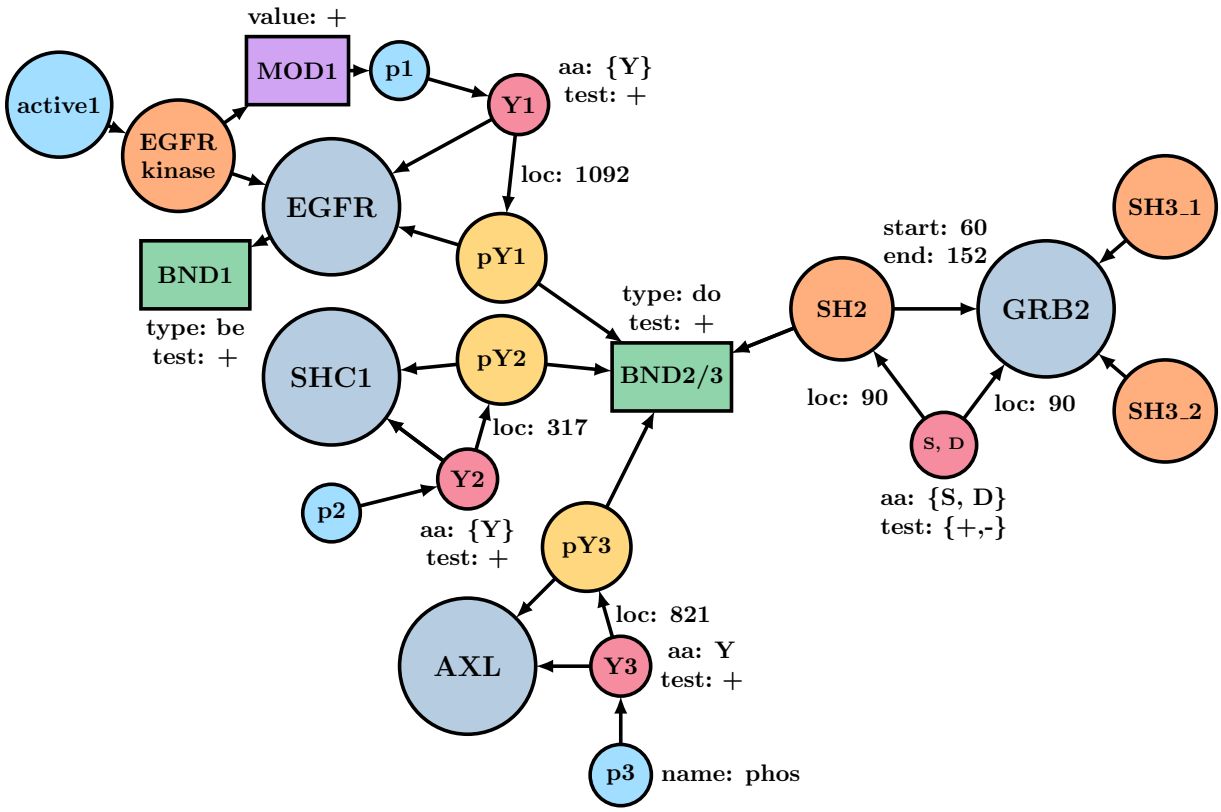


Figure 4.13: The action graph from Figure 4.10 after the semantic update. The action nodes `BND2` *and* `BND3` *from Figure 4.10 are merged into the node* `BND2/3`*.*

protein definitions to the action graph of the corresponding corpus. As the result of such application backward propagation to nuggets is performed. This backward propagation invalidates some of the nuggets rendering some of the described PPIs non-realizable.

To distinguish de-contextualized and instantiated knowledge, KAMI changes the meta-model (Figure 4.14 depicts the instantiated meta-model of KAMI $I[M]$). Now the actors of PPIs are considered to represent concrete proteins and not protoforms.

As the result of instantiation, given a set of protein defititions, we obtain a KAMI model given by a hierarchy of a form $I[\vec{N}] -I\rightarrow [A] -I\rightarrow [M]$. To understand how instantiated models are produced from knowledge corpora in KAMI, consider the following example.

Example 4.3.1. *Consider the protein definition in Figure 4.8. Application of the restrictive rule that constitutes this protein definition results in the action graph $I[A]$ illustrated in Figure 4.15. The typing of nodes is encoded with colors from Figure 4.14. Note how the former protoform nodes become typed by the protein node in $I[M]$. The names of the three clones of the protoform GRB2 produced as the result of instantiation are given in parentheses and correspond to the wild type variant Ash-L, the S90D mutant and the splice variant Grb3.*

The backward propagation to the nuggets performs cloning and removal, specified by the protein definition, for every instance of GRB2 in the nuggets, thus, producing the instantiated nuggets. Figure 4.16 illustrates some of the nuggets instantiated from our example knowledge corpus.

The three interactions depicted in the nugget graphs are performed through the SH2 domain

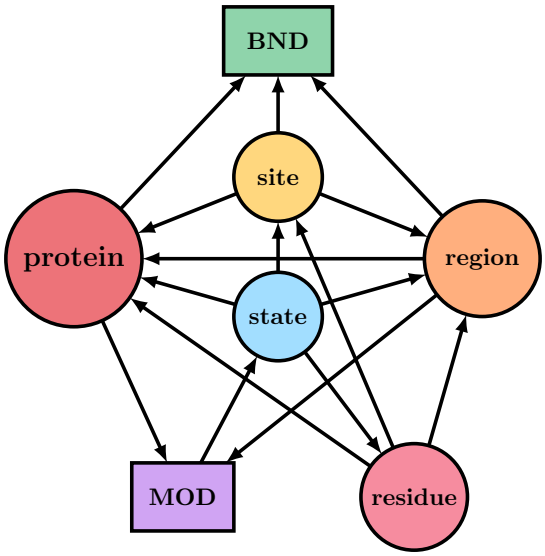


Figure 4.14: Instantiated meta-model of KAMI.

of GRB2, and are, therefore, not realizable for the splice variant Grb3 (the clones corresponding to this variant become disconnected from the action node in the nuggets and are omitted in the figure). Furthermore, note that, as a result of propagation, the attribute `aa` *of the residue node associated to the SH2 domain of the mutant S90D becomes empty. Because this node represents a positive condition on the presence of the amino acid (i.e.* `test: +`*), this condition can never be satisfied. Which implies that the described interaction mechanism is not realizable for the mutant S90D. Intuitively, this means that the wild type variant satisfies the necessary conditions (presence of the key residue S90) on the PPI to appear, while the mutant does not. On the contrary, the mutant satisfies the negative conditions (presence of the key residue D90), which means that we know that the binding does not happen.*

Therefore, the three instantiated nuggets depicted in Figure 4.16 define five distinct rules for binding interactions for: (1) products of SHC1 with Ash-L; (2) products of SHC1 with S90D; (3) products of EGFR with Ash-L; (4) products of AXL with Ash-L; and (5) products of AXL with S90D.

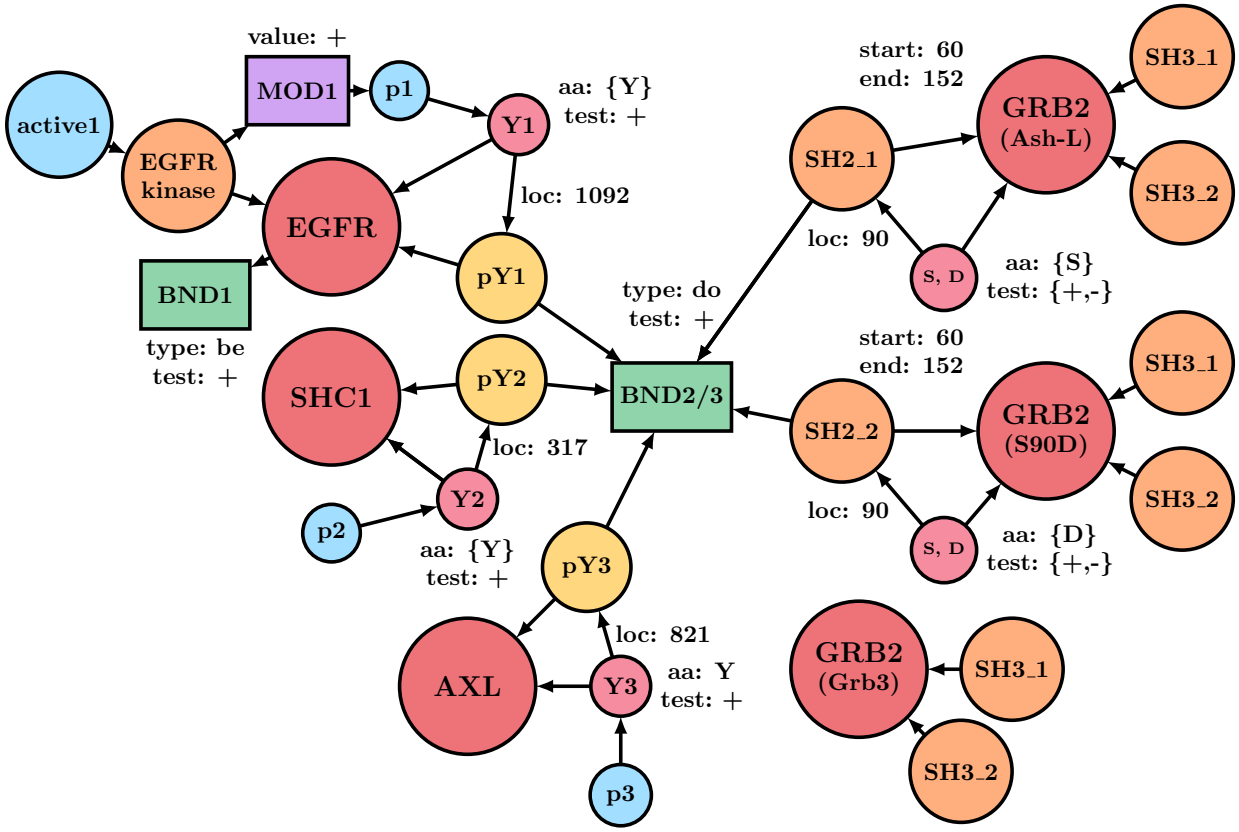
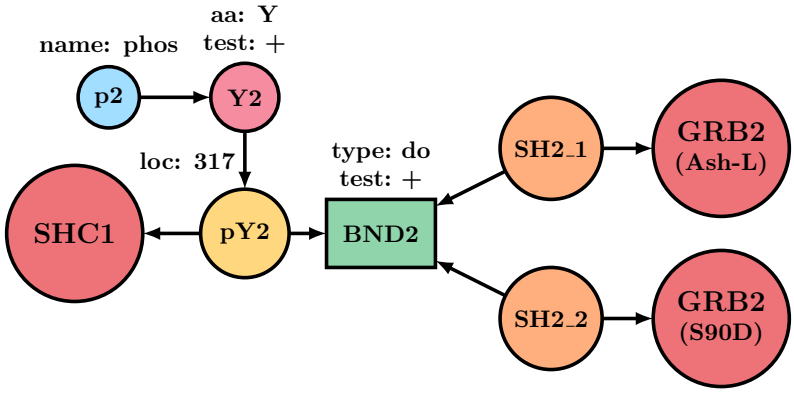
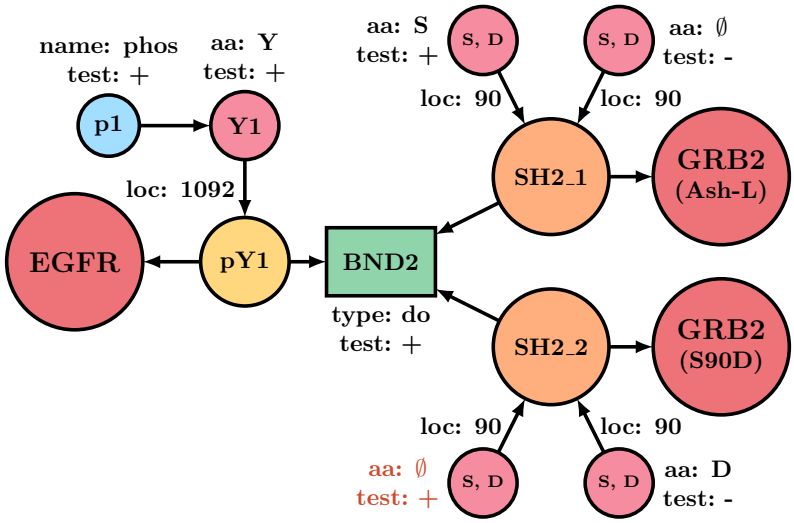


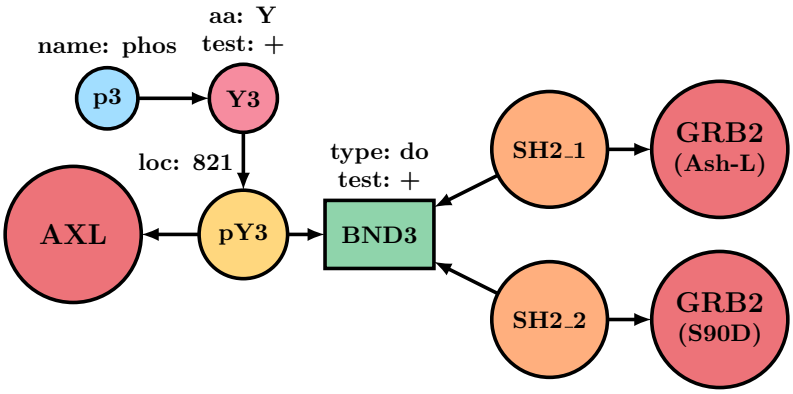
Figure 4.15: Instantiated action graph from Figure 4.13.



(a) Instantiated nugget from Figure 4.5b.



(b) Instantiated nugget from Figure 4.5c. The unsatisfiable condition is denoted with dark orange.



(c) Instantiated nugget from Figure 4.9.

Figure 4.16: Instantiated nuggets.

4.4 Generation of executable models

In this section we briefly discuss how instantiated KAMI models can be used to generate scripts written in the rule-based modelling language *Kappa* [26] (compatible with version 4 of the Kappa language and its simulator KaSim4). Given an instantiated model, KAMI generates Kappa scripts containing agent signatures, interaction rules and initial conditions. To be used for stochastic simulations, such scripts should be further augmented with observables that specify the patterns of interest (particular agents in some combination of states or bonds) whose quantitative dynamics should be tracked by the Kappa simulator[6].

Generation of agents. To generate Kappa agents, KAMI inspects the protein nodes of the instantiated action graph and generates a distinct agent per *reference protoform*. It encodes proteins derived from the same protoform with a dedicated Kappa-site called `variant` in order to optimize the simulation performance of KaSim4 [13]. Then, for each agent, it explores all the derived variants and creates a site per (not necessarily directly) adjacent state node. Because the state nodes in KAMI represent binary on/off states, every such site is of the form `site_name{on off}`. After this, KAMI adds a site per adjacent KAMI-site node and a binding node (in both cases the nodes are not required to be directly adjacent, but can be adjacent to some components of the current variant). For example, the agent signatures generated using the instantiated action graph from Figure 4.15 are given in the following listing:

```
%agent: EGFR(phos{on off}, activity{on off}, BND1_site, pY_site)
%agent: SHC1(phos{on off}, pY_site)
%agent: AXL(phos{on off}, pY_site)
%agent: GRB2(variant{AshL, S90D, Grb3}, AshL_SH2_site, S90D_SH2_site)
```

Generation of rules. To generate Kappa rules, KAMI examines instantiated nuggets together with their mapping to the instantiated action graph. As was previously mentioned, for a given agent, every adjacent binding action (therefore every binding mechanism) gives a rise to a separate Kappa site. This represents the main subtlety of the Kappa generation process, as for every binding nugget in KAMI we need to identify the site corresponding to the interaction mechanism of the binding. In KAMI's knowledge representation framework interaction rates are encoded in the interaction nodes of nuggets. However, KAMI does not enforce them to be specified, as these rates for some interactions may be unknown or depend on the context. Therefore, to generate valid Kappa, KAMI allows the user to specify default rates for binding, unbinding and modification interactions in a model; these rates are used to generate Kappa rules for nuggets whose rates are not available. The following listing illustrates the rules generated from three nuggets depicted in Figure 4.16.

```
SHC1(phos{on}, pY_site[.]), GRB2(variant{AshL}, AshL_SH2_site[.]) ->
    SHC1(phos{on}, pY_site[1]), GRB2(variant{AshL}, AshL_SH2_site[1]) @ 1.0E-4

SHC1(phos{on}, pY_site[.]), GRB2(variant{S90D}, S90D_SH2_site[.]) ->
    SHC1(phos{on}, pY_site[1]), GRB2(variant{S90D}, S90D_SH2_site[1]) @ 1.0E-4
```

[6]For the documentation refer to *P. Boutillier, J. Feret, J. Krivine, and W. Fontana. The Kappa Language and Tools* (version of November 27, 2019), `kappalanguage.org`.

```
EGFR(phos{on}, pY_site[.]), GRB2(variant{AshL}, AshL_SH2_site[.]) ->
    EGFR(phos{on}, pY_site[1]), GRB2(variant{AshL}, AshL_SH2_site[1]) @ 1.0E-4

AXL(phos{on}, pY_site[.]), GRB2(variant{AshL}, AshL_SH2_site[.]) ->
    AXL(phos{on}, pY_site[1]), GRB2(variant{AshL}, AshL_SH2_site[1])
                          @ 'default_bnd_rate'
AXL(phos{on}, pY_site[.]), GRB2(variant{S90D}, S90D_SH2_site[.]) ->
    AXL(phos{on}, pY_site[1]), GRB2(variant{S90D}, S90D_SH2_site[1])
                          @ 'default_bnd_rate'


// variables
%var: 'default_bnd_rate' 0.001
```

Generation of initial conditions. To generate initial conditions for Kappa models, KAMI allows the modeller to provide knowledge of molecular counts for different types of agents. Such molecular counts can be specified for both canonical versions of protein molecules (with no PTMs or bounds) and non-canonical ones (having a PTM, such as phosphorylation or being bound to some other agent). Section 4.5.2 illustrates how such initial concentrations can be specified using KAMI's intermediate representation language for entities. The following listing illustrates initial conditions for the previously defined Kappa agents.

```
%init: 150 EGFR() // canonical EGFR
%init: 75 EGFR(activity{on}) // active EGFR
%init: 30 EGFR(phos{on}) // phosphorylated EGFR
%init: 20 EGFR(BND1_site[BND1_site.EGFR]) // EGFR dimer
%init: 100 GRB2(variant{AshL}) // canonical Ash-L
%init: 100 GRB2(variant{S90D}) // canonical S90D
%init: 100 GRB2(variant{Grb3}) // canonical Grb3
%init: 100 SHC1() // canonical SHC1
%init: 100 AXL() // canonical AXL
```

4.5 The KAMI library

The Python library `KAMI`[7] implements the bio-curation framework presented in this chapter. It provides a programmatic API for input of individual PPIs, their aggregation into knowledge corpora, definition of protein interactions, instantiation of concrete signalling models and their export to executable Kappa scripts. It is based on the `ReGraph` library discussed in Section 2.3 and can be used with both `NetworkX` in-memory graphs and `Neo4j` persistent graphs. In this section we will write `KAMI` referring to the library and simply KAMI for the framework.

`KAMI` consists of the following principal components:

- The package `resources` provides the definitions of the built-in knowledge components of KAMI corpora and models: the meta-model, the interaction templates, the SAG and semantic nuggets.

[7] `https://github.com/Kappa-Dev/KAMI`

- The package `data_structures` provides data structures for knowledge corpora and models (the modules `corpora` and `models`). Moreover, it provides structures for knowledge input following the intermediate representation language discussed in 4.2.1 (the modules `entities` and `interactions`), as well as the data structures for programmatic input of protein definitions (the module `definitions`).

- The package `importers` provides a set of importers that convert knowledge represented using common biological formats into KAMI's intermediate representation objects (the module `biopax` for the BioPAX format, `indra` for import from INDRA statements, `intact` for import from the IntAct[8] data following PSI-MI 3.0 format).

- The package `aggregation` provides a set of utilities for automated knowledge aggregation: generation of nuggets, identification of entities and actions, bookkeeping and semantic updates of knowledge corpora.

In the rest of this section we give some details on the implemented knowledge importers and provide an example of the use of `KAMI`'s programmatic API.

4.5.1 Importers

The set of utilities implemented in the package `kami.importers` provides means for the input of PPI knowledge from several common biological formats. In this subsection we will briefly discuss the ideas behind the implemented import from BioPAX models and INDRA statement objects.

BioPAX importer. `KAMI` is able to import PPIs stored whithin BioPAX models (provided as files with the `.owl` extension). It uses the Paxtools Java utilities[9] for reading and querying BioPAX models. To collect modification interactions `KAMI` queries all the instances of the `Catalysis` class (see [32] for more details on the BioPAX ontology). Such instances are defined by the entity that controls the catalysis (controller) and a controlled reaction. For every instance of catalysis, `KAMI` selects only the instances of the `BiochemicalReaction` class as the controlled reaction (BioPAX allows the representation of other reactions such as transcription or translation, which are currently outside the scope of KAMI). The controller of catalysis is converted to an enzyme entity defining `KAMI`'s modification interaction. Then, `KAMI` examines the left- and the right hand-sides of the controlled biochemical reaction to extract the substrate entity and the target of modification. To collect binding interactions `KAMI` queries the instances of the `ComplexAssembly` class, where the left-hand side of the corresponding reaction consists of exactly two components and the right-hand side exactly one. This means that the importer is able to convert only *binary binding interactions* from BioPAX (which are exactly the binding interactions of interest). Currently, `KAMI` filters all the extracted interactions (both binding and modification) whose entities are protein complexes or families of proteins. This allows us to make sure that the imported PPIs contain knowledge on the appropriate level of mechanistic details (see the discussion on the difference between the KR level of KAMI and BioPAX in 1.3.4).

[8] `https://www.ebi.ac.uk/intact/`
[9] `https://www.biopax.org/Paxtools/`

INDRA importer. INDRA (Integrated Network and Dynamical Reasoning Assembler [45]) provides a set of tools for reading biological facts expressed with natural language and other common biological formats, and representing these facts with a set of computable statements. INDRA provides data structures for various types of PPIs including different types of modification and binding interactions[10]. The module `kami.importers.indra` implements a set of utilities for converting INDRA statements into respective `KAMI` data structures. The importer allows to extract modification interactions from `Modification`, `SelfModification`, `RegulateActivity` and `ActiveForm` statements, and binding interactions from `Complex` statements consisting of two agents (as in the case of the BioPAX importer, at the moment, KAMI focuses on binary binding interactions). Even though `KAMI`'s interactions and entities share a lot of similarities with INDRA's agents and statements, they differ in a couple of crucial points. The first important difference is that statements in INDRA cannot represent protein regions or sites involved in the interactions, whereas in KAMI, they play an important role in the identification of interaction mechanisms and their interpretation. In addition, KAMI is focused on the specific level of mechanistic details of PPIs, which, in some cases, does not coincide with the level of INDRA's knowledge representation. For example, `Complex` statements in INDRA can be used to represent a formation of protein complexes with multiple entities, `RegulateAmount`—regulation of the protein synthesis/degradation by another agent. Therefore, `KAMI` implements the import of a selected subset of INDRA statements relevant to KAMI's KR.

4.5.2 Programmatic API: example

In this subsection we provide a small example use-case for `KAMI`'s programmatic API. We illustrate how knowledge on individual PPIs from different sources can be aggregated into a knowledge corpus, how such a corpus can be instantiated into a concrete signalling model and converted to a Kappa script.

Consider the Python listing below, it illustrates how the entity and interaction data structures provided by the `KAMI` library can be used for manual input of PPIs. In addition, it shows how such objects can be serialized and de-serialized to/from the JSON format. The interaction object created in the listing corresponds to the nugget from Figure 4.5a.

```
1  from kami.data_structures.entities import *
2  from kami.data_structures.interactions import *
3
4  # Create a protoform
5  egfr = Protoform("P00533")
6
7  # Create a region actor
8  egfr_kinase = RegionActor(
9      protoform=egfr,
10     region=Region(name="Protein kinase", start=712, end=979,
11                   states=[State("activity", True)]))
12
13 # Create a ligand modification interaction
14 interaction = LigandModification(
```

[10]The full list of INDRA statements can be found at `https://indra.readthedocs.io/en/latest/modules/statements.html`.

```
    enzyme=egfr_kinase, substrate=egfr,
    target= Residue("Y", 1092, state=State("phosphorylation", False)), value=True,
    rate=1, desc="Phosphorylation of EGFR homodimer")

# Convert interaction to JSON
int_json = interaction.to_json()

# Convert JSON back to interaction
copy_interaction = Interaction.from_json(int_json)
```

The following listing shows how a `KAMI` corpus can be created. KAMI requires corpora to be assigned an identifier (e.g. `EGFR_signalling` in the listing below). An interaction object can be added to the corpus using the `add_interaction` method. As a result, a new nugget graph is generated, added to the corpus and the new bits of knowledge are propagated to the action graph (as described in Section 4.2). `KAMI` provides various tools for accessing the components of the corpus, e.g. a nugget graph object, the action graph, identification of entities and actions in the nugget by the action graph, typing of the action graph by the meta-model, etc. In addition, `KAMI` implements a set of tools for manual addition of new entities and actions to the action graph (independent from the aggregation process).

```
from kami import KamiCorpus

# Create an empty KAMI corpus based on in-memory graphs
corpus = KamiCorpus("EGFR_signalling", backend="networkx")

# Add interaction to the corpus
new_nugget_id = corpus.add_interaction(interaction)

# Access the newly created nugget graph
nugget = corpus.get_nugget(new_nugget_id)
print(corpus.get_nugget_desc(new_nugget_id))
print(nugget.nodes())

# Access the action graph
ag = corpus.action_graph
print(ag.nodes())

# Get the identification of nugget nodes in the action graph
print(corpus.get_nugget_typing(new_nugget_id))

# Get typing of the action graph by the meta-model
print(corpus.get_action_graph_typing())

# Get all the protoforms in the corpus
print(corpus.protoforms())

# Find a protoform node by the UniProt AC of its gene
```

```
egfr_protoform_node_id = corpus.get_protoform_by_uniprot("P00533")

# Manually add a new protoform and its site to the corpus
new_protoform_node = corpus.add_protoform(Protoform("P62993"))
corpus.add_site(Site("New site"), new_protoform_node)
print(corpus.get_attached_sites(new_protoform_node))
```

The following listing creates interaction objects corresponding to the nuggets from Figures 4.5b and 4.5c and adds them to the knowledge corpus.

```
grb2 = Protoform("P62993")
grb2_sh2 = RegionActor(
    protoform=grb2,
    region=Region(name="SH2"))

shc1 = Protoform("P29353")
shc1_pY = SiteActor(
    protoform=shc1,
    site=Site(
        name="pY",
        residues=[Residue("Y", 317, State("phosphorylation", True))]))
interaction1 = Binding(grb2_sh2, shc1_pY)

grb2_sh2_with_residues = RegionActor(
    protoform=grb2,
    region=Region(
        name="SH2",
        residues=[
            Residue("S", 90, test=True),
            Residue("D", 90, test=False)]))

egfr_pY = SiteActor(
    protoform=egfr,
    site=Site(
        name="pY",
        residues=[Residue("Y", 1092, State("phosphorylation", True))]))

interaction2 = Binding(grb2_sh2_with_residues, egfr_pY)

corpus.add_interactions([interaction1, interaction2])
```

As we have previously mentioned, `KAMI` provides an importer for PPIs represented using the BioPAX format. The following listing illustrates how KAMI interaction objects can be created from a BioPAX model stored in a `.owl` file[11].

[11]The dataset `PathwayCommons11.pid.BIOPAX.owl` from the listing can be found at `https://www.pathwaycommons.org/archives/PC2/v11/`.

```
from kami.importers.biopax import BioPaxImporter

# Convert BioPax model into KAMI interactions
bp_importer = BioPaxImporter()
biopax_interactions = bp_importer.import_model("PathwayCommons11.pid.BIOPAX.owl")

# Add interactions to the corpus
corpus.add_interactions(biopax_interactions)
```

Moreover, `KAMI` allows to convert INDRA statement objects into native entity and interaction objects. In the following listing a text containing a mechanistic description of PPIs is proccessed using INDRA's TRIPS processor [45] into statement objects. These objects are further converted into interactions and added to the corpus.

```
from indra.sources import trips
from kami.importers.indra import IndraImporter


text = (
  "MAP2K1 phosphorylates MAPK3 at Thr−202 and Tyr−204;"
  "ABL1 phosphorylates PLCG1 at Y394.")

# Proccess text using INDRA's TPIS processor
trips_processor = trips.process_text(text)
# Get INDRA statements
statements = trips_processor.statements

# Convert statements into KAMI interactions
indra_importer = IndraImporter()
indra_interactions = indra_importer.import_statements(statements)

# Add interactions to the corpus
corpus.add_interactions(indra_interactions)
```

`KAMI` provides the `Definition` data structure for creation of protein definitions. As input, the constructor of `Definition` takes a protoform object and a list of `Product` objects. The latter objects define which components are removed from the protoform and which amino acids are set to its key residues in particular protein products. The following listing creates the protein definition for GRB2 from Figure 4.8. Definition objects provide the `generate_rule` method that returns `ReGraph`'s rule object that corresponds to the representation of protein definitions described in Section 4.1. The created protein definition can be further used to instantiate a concrete signalling model.

```
# Create a protein definition for GRB2
protoform = Protoform(
    "P62993",
```

```
    regions=[Region(
        name="SH2",
        residues=[
            Residue("S", 90, test=True),
            Residue("D", 90, test=False)])])

ashl = Product(name="Ash−L", residues=[Residue("S", 90)])
s90d = Product(name="S90D", residues=[Residue("D", 90)])
grb3 = Product(name="Grb3", removed_components={"regions": [Region("SH2")]})

grb2_definition = Definition(protoform, products=[ashl, s90d, grb3])

# Generate the instantiation rule from the definition given the corpus
rule, instance = grb2_definition.generate_rule(
  corpus.action_graph, corpus.get_action_graph_typing())

# Instantiate a model for the corpus
grb_variants_model = corpus.instantiate("EGFR_signalling_GRB2", [grb2_definition])
```

The module `kami.exporters.kappa` provides a set of utilities for the generation of executable Kappa scripts from both instantiated models and knowledge corpora (provided protein definitions for the protoforms present in the corpora). Consider the listing below, it defines initial conditions for protein products of EGFR, GRB2 and SHC1. Such conditions specify the number of molecules for different states of the corresponding proteins in the initial mixture. For example, the listing defines the following initial concentrations of the EGFR products in the mixture:

- 150 molecules of the canonical EGFR protein (no PTMs, bounds or activity);
- 75 molecules of the EGFR protein with the active kinase domain;
- 30 molecules of the EGFR protein with the phosphorylated Y1092;
- 30 molecules of the EGFR protein with the phosphorylated Y1092 and bound to the SH2 domain of Ash-L through its pY site;
- 30 instances of the EGFR protein dimer (EGFR bound to another EGFR).

```
from kami.exporters.kappa import KappaInitialCondition


# Initial condition for EGFR
egfr_initial = KappaInitialCondition(
    canonical_protein=Protein(Protoform("P00533")),
    canonical_count=150,
    stateful_components=[
        (egfr_kinase, 75),
        (Residue("Y", 1092, state=State("phosphorylation", True)), 30),
```

```
        (Site(
            name="pY",
            residues=[Residue("Y", 1092,
                              state=State("phosphorylation", True))],
            bound_to=[
                RegionActor(
                    protoform=grb2, region=Region(name="SH2"),
                    variant_name="Ash-L")
            ]), 30)
    ],
    bonds=[
        (Protein(Protoform("P00533")), 30),
    ])

# Initial condition for Ash-L
ashl_initial = KappaInitialCondition(
    canonical_protein=Protein(Protoform("P62993"), "Ash-L"),
    canonical_count=200,
    stateful_components=[
        (Region(name="SH2", bound_to=[shc1_pY]), 40)])

# Initial condition for S90D
s90d_initial = KappaInitialCondition(
    canonical_protein=Protein(Protoform("P62993"), "S90D"),
    canonical_count=20,
    stateful_components=[
        (Region(name="SH2", bound_to=[egfr_pY]), 10)])

# Initial condition for Grb3
grb3_initial = KappaInitialCondition(
    canonical_protein=Protein(Protoform("P62993"), "Grb3"),
    canonical_count=70)

# Initial condition for SHC1
shc1_initial = KappaInitialCondition(
    canonical_protein=Protein(Protoform("P29353")),
    canonical_count=100,
    stateful_components=[
        (Residue("Y", 317, state=State("phosphorylation", True)), 30)])

initial_concentrations = [
  egfr_initial,
  ashl_initial,
  s90d_initial,
  grb3_initial,
  shc1_initial
]
```

The two listings below illustrate how the `ModelKappaGenerator` and `CorpusKappaGenerator` classes can be used to generate Kappa scripts from the previously defined model and corpus respectively. The generation adds initial conditions corresponding to the concentrations defined in the previous listing. The `default_concentation` argument is used to assign default concentration for canonical agents (with no PTMs and bounds) that are not mentioned in the `initial_concentrations` parameter.

```
from kami.exporters.kappa import
          ModelKappaGenerator

# Create a Kappa generator from a model
generator = ModelKappaGenerator(
  grb_variants_model)
# Generate Kappa with default agent
# concentration 75 molecules per agent
kappa_str = generator.generate(
    initial_concentrations,
    default_concentation=75)
```

```
from kami.exporters.kappa import
          CorpusKappaGenerator

# Create a Kappa generator from a corpus
generator = CorpusKappaGenerator(
  corpus, [grb2_definition])
# Generate Kappa with default agent
# concentration 75 molecules per agent
kappa_str = generator.generate(
    initial_concentrations,
    default_concentation=75)
```

4.6 KAMIStudio

KAMIStudio is an environment for biocuration of cellular signalling knowledge based on the KAMI framework. It provides features for semi-automatic curation of large corpora of cellular signalling knowledge including:

- interactive visualization of knowledge stored in corpora and models;
- input of individual PPIs to a corpus through intuitive forms as well as batch import from JSON-formatted interactions resulting in the automatic aggregation of new knowledge to the corpus;
- an interface for specifying protein variants; automatic instantiation of corpora into models using protein variants;
- automatic generation of Kappa scripts from models.

Interactive visualization of corpora and models in KAMIStudio allows the user to interact with graphs in various ways: click on graph elements to view (and modify) the attached meta-data, zoom, pan, drag the nodes. Moreover, using the meta-data attached to the graph elements, KAMIStudio provides cross-referencing to common databases such as UniProt and InterPro. Such interactive capabilities may provide some additional insights to knowledge, e.g. on the structure of the underlying PPI network, its connected components or its hub nodes and may also suggest manual edits necessary to make the data consistent with the modeller's viewpoint. Moreover, KAMIStudio offers an intuitive interface for the creation of protein definitions and their visualization.

Intuitive forms for the PPI input implemented in KAMIStudio are based on the intermediate representation language presented in Subsection 4.2.1. They provide a graphical interface through which the user can specify the type of interaction (e.g. modification, self-modification, binding), its actors (e.g. enzyme, substrate, binding partners), create and nest various protoform components (e.g. regions, sites, residues) and set them as actors of interactions. Given an input interaction, KAMIStudio generates a preview of the nugget graph which allows the user to intervene in the automatic entity and action identification process. In addition, graphical visualization of corpora provides means for manual intervention in the aggregation process and allows the user to select different action nodes and merge them, effectively stating *"I know that these interactions are instances of the same mechanism"*.

KAMIStudio is a web-based application: its server can be started locally and its functionality can be used in a browser via the provided client. The knowledge represenation and update-related backend is based on the Python libraries `ReGraph` and `KAMI`. To store data, KAMIStudio uses two noSQL database technologies: Neo4j and MongoDB. The full version can be installed from source and run locally (detailed installation instructions can be found in the github repository[12]). In addition, a read-only demo is available online at `http://kamistudio.ens-lyon.fr/`. The online demo contains three example corpora: *EGFR signalling* built from a subset of individual PPIs involved in the EGFR signalling pathway, *pYNET 20* and *pYNET 200* built from respectively 20 and 200 random PPIs involving tyrosine phosphorylations and bindings of SH2 domains to phosphotyrosine-containing sites. The demo also contains three models that can be used to generate Kappa scripts. The first model is an instantiation of the *EGFR signalling* corpus using splice variants and mutants of genes EGFR and GRB2. The two other models represent instantiations of *pYNET 20* and *pYNET 200* using the wild-type variants. These models are built by aggregation of independent PPIs without pre-conceived pathways in mind. A superficial look at the action graph of the *pYNET 20* model reveals a number of disconnected components most of which correspond to individual PPIs which suggests to the modeller some gaps in the collected knowledge. On the other hand, the action graph of the *pYNET 200* model starts exhibiting a large connected component, which suggests the potential emergence of pathways.

4.7 Discussions and conclusions

The bio-curation framework KAMI described in this thesis represents a novel approach to *meta-modelling* of cellular signalling that makes the first step towards decoupling the process of knowledge curation from model building. It proposes a de-contextualized KR based on hierarchies of graphs that is designed to accommodate both mechanistic and phenomenological knowledge of mechanisms of PPIs. Such a KR allows for both meaningful semi-automated aggregation of knowledge from various sources, as well as instantiation of this knowledge in different contexts. Furthermore, this KR delivers a set of transparent curation tools based on a rigorous mathematical theory for rewriting and propagation in hierarchies.

The semi-automatic knowledge aggregation mechanism of KAMI allows for 'gluing' fragments of knowledge on individual PPI into coherent corpora that can be studied as is or further reused to instantiate concrete signalling models in different contexts. This instantiation process in based on the idea that, depending on the anatomy of concrete protein products arising in different cellular contexts (mutants, splice variants, etc.), some PPI mechanisms described in

[12] `https://github.com/Kappa-Dev/KAMIStudio`

a corpora are realized and some are muted. This allows the curator to automatically generate multiple signalling models from the knowledge corpus. KAMI provides means for converting such models into executable scripts that can be used for simulation of the dynamics and further analysis of the underlying systems.

KAMI's bio-curation approach makes knowledge collation and curation semi-automatic and model building entirely automatic. This, however, does not make *the human curator* and *the human modeller* superfluous in the meta-modelling process. It simply frees the human expert from the cognitive burden of the manual collation of fragmentary knowledge and the necessity to conceive and build large complex models. It directs the efforts of the human curator towards finding high-quality pertinent knowledge sources, reviewing the aggregated corpora and finding gaps and discrepancies in the represented knowledge (and potentially resolving representation level mismatches). The expertise of the modeller, on the other hand, is necessary to *discover artifacts* of the generated models, design pertinent questions and tools for automated analysis of such models. Such an approach makes it possible to build 'models of nothing'—dynamical models built from seemingly unrelated observations without preconceived ideas on the system they model—and discover emergent phenomena in their dynamics, for example, signalling pathways.

4.7.1 Future work

Together with the `KAMI` library and the KAMIStudio environment, the framework is in an active development phase. Therefore, a significant amount of work remains to be done to make it a mature bio-curation tool. In the rest of this section we provide some ideas on possible features that need to be designed and implemented.

First of all, incorporating richer semantic background knowledge and more elaborate mechanisms for semantic tagging in KAMI would allow for better identification of action mechanisms, nugget autocompletion and semantic updates. Such background knowledge could include, for example, generic interaction mechanisms of PTB domains, ubiquitin ligases, etc.

Secondly, adapting a *version control* system based on the audit trail for hierarchies of graphs (discussed in Subsection 2.2.10) to the domain-specific purposes of KAMI would allow the expert to document the curation process, maintain different versions of knowledge corpora, merge them, and rollback to specific points in the curation history. It could enable the accommodation of different versions of individual nuggets (representing, for example, conflicting knowledge on some PPI) or different action graphs (representing different interpretations of nuggets). This version control system should go in hand with a sophisticated knowledge *annotation system*. The curator should be able to assign provenance and epistemic status (e.g. experimentally observed or inferred facts, phenomenological observations, hypotheses) to both accommodated bits of knowledge and the curation actions themselves.

Moreover, KAMI requires a language for *querying knowledge* stored in its corpora. As a corpus grows, manual reviewing and update of its knowledge becomes increasingly difficult. A specialized language would allow the curator to formulate the queries of interest and make the manual component of curation more efficient. Such queries could, for example, include finding all the mechanisms of interaction between two given protoforms, finding all the nuggets where a protoform binds through one of its conserved domains, finding all the protoforms phosphorylated by some protein kinase domain, etc.

This language could also provide means for the *static analysis* of knowledge present in a corpus: detection of 'dead' nuggets (nuggets that describe interactions whose mechanism cannot

be realized given the knowledge already present in the corpus), identify various relations between nuggets (positive or negative influence of the interactions they describe), test the reachability of a particular molecular species (some polymers or large protein complexes).

Integration of KAMI with the Kappa simulator KaSim and the causal story extractor KaSTOR would consolidate a powerful rule-based modelling platform allowing for an interesting modelling *feedback*: knowledge discovered from simulations and causal analysis of generated models could be fed back into the knowledge corpus. Moreover, the previously mentioned query language could be used to formulate *assertions* expressing, for example, 'actions at a distance', e.g. activation of A eventually leads to the activation of B. Then static analysis and causal stories extracted from simulations could be combined to test whether a given assertion is validated by the knowledge present in the corpus.

Another interesting feature of KAMI could include the accommodation of *relations between corpora* with the possibility to *transfer knowledge* between them. For example, if different corpora contain knowledge on PPIs taking part in cellular signalling of different species, the relation between these corpora could designate the orthology relation between protoforms. Then, some interaction mechanisms from one corpus could be transferred to another 'by similarity'.

The KAMI framework can be used as a basis for building an *open repository* for knowledge corpora on PPI mechanisms constituting cellular signalling, similarly to such repositories as GeneOntology[13] for functions of genes, UniProt[14] for protein sequences and functions, InterPro[15] for protein families, domains and functional sites, etc. Such a repository could be used to store and curate reviewed high-quality knowledge corpora that would be accessible to the scientific community for sharing and browsing known PPI mechanisms, as well as directly generating executable models from the corpora and their fragments.

As proof of concept for the KAMI framework several small use-cases were developed (such as the example from Subsection 4.5). In addition, the first attempt to use KAMI for building a large 'model of nothing' by aggregating knowledge on tyrosine phosphorylations and SH2/pY bindings is currently under development. Nevertheless, to explore its capabilities and shortcomings KAMI still requires some large use-cases to be implemented. Such use-cases could include well-studied and reviewed models (for example, the Wnt signaling pathway, the activation of Raf, as suggested in [48]) or large-scale 'models of nothing' that could be used to confirm already known pathways or discover completely new ones.

[13] `http://geneontology.org/`
[14] `https://www.uniprot.org/`
[15] `https://www.ebi.ac.uk/interpro/`

Chapter 5

Conclusions

In this thesis we have designed a knowledge representation and curation framework based on hierarchies of graphs—a structure consisting of graphs related with homomorphisms. It is particularly suitable for representing fragments of knowledge on complex systems and expressing various relations between these fragments, such as instantiation, identification and so on.

We have presented a mathematical theory for rewriting in hierarchies. Transformations expressed with such rewriting can be used for updating accommodated knowledge and performing an audit of updates. We have described the mechanism for rewriting individual objects situated in hierarchies that propagates corresponding transformations to other objects and homomorphisms and maintains the structure and consistency of the hierarchy. The developed theory is general and applies to hierarchies of any objects satisfying particular structural requirements (that make the SqPO rewriting possible).

We have introduced the notion of a rule hierarchy—a hierarchy of SqPO rules that can be used to rewrite entire hierarchies of objects. We have investigated the question of its applicability given a fixed instance in a hierarchy, as well as the reversiblity of rewriting it induces. Finally, we have presented a construction that allows to synthesize rule hierarchies corresponding to the composition of two successive rewrites in a given hierarchy.

The described theory is further used to design an audit trail system that records the update history of individual objects and their hierarchies. The system allows rolling back to any state of this history, as well as maintaining multiple diverged versions of the same object and merging these versions.

Two major use-cases of the introduced framework were described: the design of schemas for graph databases and the development of a framework for the curation of cellular signalling knowledge.

The first application of graph hierarchies allows us to design schemas for the property graph data model, widely used in modern graph databases. Using graph homomorphisms, such schemas constrain the shape of a database: the set of node types and allowed edge types between different nodes, the sets of properties and their types for graph elements. We have described how the developed framework for rewriting and propagation in hierarchies provides powerful tools for co-evolution of a schema and its data instance.

The second application constitutes the basic knowledge representation and curation capabilities of the bio-curation framework KAMI. This framework decouples the proccess of knowledge curation from model building and allows the building of large signalling models from knowledge. It provides tools for semi-automated aggregation of fragmented knowledge on individual

protein-protein interactions into coherent corpora. These corpora can be used for automatic generation of dynamical rule-based models of cellular signalling in different contexts.

The described theory and its applications have given rise to a set of open-source software tools and prototypes, notably, the Python libraries `ReGraph` and `KAMI` and the web-based environment `KAMIStudio`.

5.1 Future work

This work presents a powerful generalization of SqPO rewriting in hierarchies of objects. Many classical questions in graph transformation remain to be studied with respect to this generalization, such as concurrency, parallelism, various application conditions and so on. Answering these questions is important not only to provide a fully developed theory of rewriting in hierarchies but also to provide theoretical tools for immediate concrete applications, among which is equipping the hierarchy audit trails with the 'undo' operation.

Likewise, the two presented applications of graph hierarchies require some future development. For example, possible future directions in the design of schemas for PGs include incorporating the notion of mandatory properties to PG schemas (thus, integrating a notion of mandatory attributes and their rewriting into the presented mathematical framework), exploiting three-level hierarchies for representing modifiable graph views, designing techniques for automated schema inference, etc.

The KAMI framework requires the development of new features that would allow it to become a mature bio-curation tool. Among others, such features include support for richer semantic background knowledge, a version control system adapted to the domain-specific purposes, a knowledge annotation system allowing the curator to assign provenance and epistemic status to the newly added knowledge, new tools for static analysis of knowledge.

Finally, some future work remains to be done for making the software implemented as part of this thesis full-fledged tools. The `ReGraph` Python library represents a rather complete framework allowing the user to build arbitrary knowledge representations based on hierarchies of graphs. It provides features for rewriting and audit trailing for both individual simple graphs and their hierarchies and is available as part of the Python package index[1]. Additionally, the library contains a prototype system for working with schema-aware PGs. The native support for schemas, however, remains to be integrated into the PG database technologies, such as Neo4j.

The `KAMI` Python library together with the standalone bio-curation environment `KAMIStudio` are in their earlier development stage. Apart from the implementation of new features of the KAMI framework (e.g. annotation system, static analysis), they require integration with the Kappa platform [14], which would give rise to a powerful rule-based modelling platform allowing for a full spectrum of modelling features: from knowledge curation, its static analysis, to model building, simulation and analysis.

[1] `https://pypi.org/project/regraph/`

Appendix A

Mathematical background

In this appendix we provide some mathematical background for the rest of the thesis and, in particular, Chapter 2. We start by introducing the notions of simple and non-simple graphs, graph homomorphisms. We define the dictionary structure that can be used to assign attributes to the nodes and edges of different graphs. These mathematical objects are used as the building blocks for the KR system of our interest. We then introduce the basic notions from category theory necessary to define the constructions that serve us as tools for knowledge update and curation. Among such notions are those of pullback and pushout, well-known in the community of graph rewriting and beyond. We also present the slighly less-known notions of pullback complements and image factorizations, predominantly used in the context of the SqPO rewriting. Additionally, this appendix assembles a set of useful lemmas collected from different literature and used across this thesis.

A.1 Graphs

Definition A.1.1. A *simple graph* G is defined by a tuple (V, E), where V is a set of vertices, $E \subseteq V \times V$ is a set of edges.

Definition A.1.2. A *non-simple graph* G is defined by a tuple (V, E, s, t), where V is a set of vertices, E is a set of edges, $s : E \to V$ and $t : E \to V$ are two functions assigning to each edge its source and target node respectively.

Remark A.1.3. In the definition of simple graphs, E being a set implies that at most one edge is allowed from the same source vertex to the same target vertex. We can draw an analogy with the definition of simple graphs for non-simple graphs, and say that edges of non-simple graphs form a *multiset* of pairs of vertices (which precisely implies that more than one edge is allowed from the same source to the same target vertex).

For the sake of conciseness, given a graph G, we will denote the set of its vertices (or nodes) as V_G and the set of its edges as E_G, without specifically saying that G is defined by a tuple (V_G, E_G). Moreover, for an edge $e \in E_G$, we will often denote with $source(e)$ and $target(e)$ the source and the target nodes of e respectively. For a simple graph $G = (V_G, E_G)$ and an edge $e = (u, v) \in E_G$, $source(e) = u$ and $target(e) = v$, while for a non-simple graph $H = (V_H, E_H, s_H, t_H)$ and an edge $e \in E_H$, $source(e) = s_H(e)$ and $target(e) = t_H(e)$.

Definition A.1.4. A *path* from a node s to a node t, denoted $path(s,t)$, is defined by a sequence of edges $(e_1, e_2, \ldots, e_n)$ in G such that $source(e_1) = s$, $target(e_n) = t$ and $target(e_i) = source(e_{i+1})$ for all $1 \leq i \leq n-1$.

Definition A.1.5. A graph $G = (V, E)$ is *directed acyclic* (or is a DAG), if for every node $v \in V$ there does not exists a non-empty $path(v, v)$, called a cycle.

Definition A.1.6. A *homomorphism of simple graphs* $G = (V^G, E^G)$ and $H = (V^H, E^H)$ is defined by a mapping $h : V^G \to V^H$ such that edges are preserved, i.e. if $(u, v) \in E^G$, $(h(u), h(v)) \in E^H$.

Definition A.1.7. A *homomorphism of non-simple graphs* $G = (V^G, E^G, s^G, t^G)$ and $H = (V^H, E^H, s^H, t^H)$ is defined by two mappings $h_v : V^G \to V^H$ and $h_e : E^G \to E^H$ such that $s(h_e(e)) = h_v(s(e))$ and $t(h_e(e)) = h_v(t(e))$ for all $e \in E$.

In our KR system we would like to equip graph nodes and edges with attributes that can be used to express, for example, states, properties, qualities of entities and relations represented with nodes and edges. Thus, attributes of a graph element are given by a *dictionary*.

Definition A.1.8. A *dictionary* is a function $d : V \to K$ that maps a finite set of values to a finite set of keys, V and K here are the objects of the category $\mathbf{Sets}_{fin}$

Definition A.1.9. A dictionary $d_1 : V_1 \to K_1$ is a *subdictionary* of $d_2 : V_2 \to K_2$ ($d_1 \leq d_2$), if the following square commutes:

$$\begin{array}{ccc} V_1 & \xrightarrow{d_1} & K_1 \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle g} \\ V_2 & \xrightarrow{d_2} & K_2 \end{array}$$

where arrows f and g are injective maps. Intuitively, these injective maps can be seen as set inclusions $V_1 \subseteq V_2$ and $K_1 \subseteq K_2$ up to renaming (of keys and values). This defines the *dictionary inclusion* relation $\leq$.

We also define the operations of dictionary union and difference that are useful when performing transformations of objects equipped with attributes.

Definition A.1.10. The *union* $d_1 \cup d_2$ of two dictionaries $d_1 : V_1 \to K_1$ and $d_2 : V_2 \to K_2$ is a dictionary $d : V_1 \cup V_2 \to K_1 \cup K_2$ such that for all $v_1 \in V_1$ $d(v_1) = d_1(v_1)$ and for all $v_2 \in V_2$ $d(v_2) = d_2(v_2)$.

Definition A.1.11. The *difference* $d_1 \setminus d_2$ of two dictionaries $d_1 : V_1 \to K_1$ and $d_2 : V_2 \to K_2$ is a dictionary $d : V_1 \setminus V_2 \to K_1$ such that $d(v) = d_1(v)$ for all $v \in V_1$ such that $v \notin V_2$.

We define *graphs with attributes* adding to the respective definitions two sets A^V and A^E of vertex and edge attributes respectively and two functions $f : V \to A^V$ and $g : E \to A^E$ mapping an attribute dictionary for every vertex and edge. A homomorphism between two graphs with attributes is then required to satisfy the following property: for every graph element (a vertex or an edge) the attribute dictionary of the source graph element is a *subdictionary* of the target graph element it maps to. We formalize it for simple and non-simple graphs in the following definitions.

Definition A.1.12. A *simple graph with attributes* G is defined by a tuple (V, E, A^V, A^E, f, g), where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, A^V and A^E are sets of dictionaries, a function $f : V \to A^V$ assigns a dictionary from A^V to every vertex and $g : E \to A^E$ assigns a dictionary from A^E to every edge of the graph.

Definition A.1.13. A *homomorphism* between simple graphs with attributes $G = (V_G, E_G, A^V_G, A^E_G, f_G, g_G)$ and $H = (V_H, E_H, A^V_H, A^E_H, f_H, g_H)$ is defined by a mapping $h : V_G \to V_H$ such that

- if $(u, v) \in E_G$, $(h(u), h(v)) \in E_H$ (edges are preserved),
- $f_G(u) \leq f_H(h(u))$ for all $u \in V_G$ (node attributes are preserved),
- $g_G(u, v) \leq g_H(h(u), h(v))$ for all $(u, v) \in E_G$ (edge attributes are preserved).

Definition A.1.14. A *non-simple graph with attributes* G is defined by a tuple $(V, E, s, t, A^V, A^E, f, g)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, $s : E \to V$ and $t : E \to V$ are two functions mapping edges to their source and target nodes respectively, A^V and A^E are sets of dictionaries, a function $f : V \to A^V$ assigns a dictionary from A^V to every vertex and $g : E \to A^E$ assigns a dictionary from A^E to every edge of the graph.

Definition A.1.15. A *homomorphism* between non-simple graphs with attributes $G = (V_G, E_G, s_G, t_G, A^V_G, A^E_G, f_G, g_G)$ and $H = (V_H, E_H, s_H, t_H, A^V_H, A^E_H, f_H, g_H)$ is defined by two mappings $h_v : V_G \to V_H$ and $h_e : E_G \to E_H$ such that

- $s(h_e(e)) = h_v(s(e))$ and $t(h_e(e)) = h_v(t(e))$ for all $e \in E$,
- $f_G(u) \leq f_H(h(u))$ for all $u \in V_G$,
- $g_G(e) \leq g_H(h_e(e))$ for all $e \in E$.

A.2 Categories

The following definition of what is a category can be found in [5]:

Definition A.2.1. A *category* is given by:

- objects: $A, B, C, D, \ldots$
- arrows : $f, g, h, \ldots$
- for each arrow f, two objects $dom(f)$ and $codom(f)$ are given, they are called the *domain* and the *codomain* of f respectively; if $dom(f) = A$ and $codom(f) = B$, we write $f : A \to B$;
- given a pair of arrows $f : A \to B$ and $g : B \to C$ (with $codom(f) = dom(g)$), there is an arrow $g \circ f : A \to C$ called the *composite* of f and g;
- for every object A, an arrow $1_A : A \to A$ is given, it is called the *identity arrow* of A.

such that the following properties hold:

- *Associativity:* $h \circ (g \circ f) = (h \circ g) \circ f$ for all $f : A \to B, g : B \to C$ and $h : C \to D$.
- *Unit:* $f \circ 1_A = f = 1_B \circ f$ for all $f : A \to B$.

Classical and well-known examples of categories include:

- the category of sets and functions **Sets**,
- the category of finite sets and set inclusions $\mathbf{Sets_{fin}}$,
- the category of posets and monotone functions **Pos**,
- the category of topological spaces and continuous maps **Top**.

Categories of our interest include

- the category of simple graphs and their homomorphisms **SimpGrph**,
- the category of non-simple graphs and their homomorphisms **Grph**,
- the category of dictionaries and dictionary inclusions **Dict**,
- the category of simple graphs with attributes and their homomorphisms $\mathbf{SimpGrph}_{attrs}$,
- the category of non-simple graphs with attributes and their homomorphisms $\mathbf{Grph}_{attrs}$.

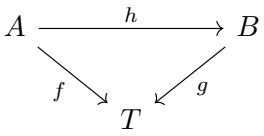
Definition A.2.2. An arrow $f : B \to C$ is a *monomorphism* (or a *mono*) if for any object A and any arrow $g_1, g_2 : A \to B$, $f \circ g_1 = f \circ g_2$ implies that $g_1 = g_2$. We will often denote monos as $f : B \rightarrowtail C$.

Definition A.2.3. An arrow $f : A \to B$ is an *epimorphism* (or an *epi*) if for any object C and any arrow $g_1, g_2 : B \to C$, $g_1 \circ f = g_2 \circ f$ implies that $g_1 = g_2$. We will often denote epis as $f : B \twoheadrightarrow C$.

Definition A.2.4. An arrow $f : A \to B$ is an *isomorphism* (or an *iso*) if it is invertible, i.e. there exists an *inverse* $f^- : B \to A$ such that $f \circ f^- = Id_B$ and $f^- \circ f = Id_A$.

Definition A.2.5. An *initial object* of a category **C** is an object I such that, for every other object X, there exists a unique arrow $f : I \to X$.

Definition A.2.6. The *slice category* $\mathbf{C}/T$ of a category **C** over an object T from **C** is the category whose objects are all arrows f from **C** with $codom(f) = T$, and whose arrows h from an object $f : A \to T$ to an object $g : B \to T$ are given by arrows $h : A \to B$ from **C** that make the following diagram commute.

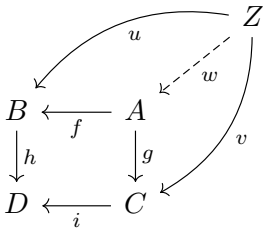


(A.1)

Example A.2.1. *For a fixed simple or non-simple graph T the slice category* $\mathbf{SimpGrph}/T$ *or* $\mathbf{Grph}/T$ *respectively defines the category of graphs* typed *by T.*

A.3 Pullbacks

Definition A.3.1. The *pullback* of a co-span $B \xrightarrow{h} D \xleftarrow{i} C$ is given by a span $B \xleftarrow{f} A \xrightarrow{g} C$ for which $h \circ f = i \circ g$ and which has the following universal property: for all $B \xleftarrow{u} Z \xrightarrow{v} C$ such that $h \circ u = i \circ v$ there exists a unique $w : Z \to A$ such that $u = f \circ w$ and $v = g \circ w$.



(A.2)

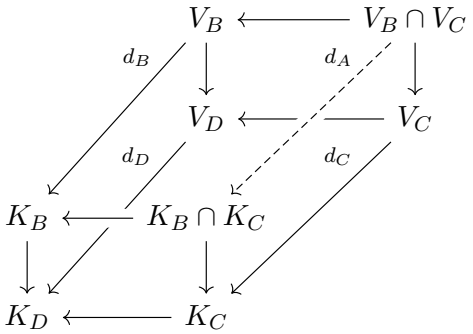
Example A.3.1. *In the category* $\mathbf{Set}_{fin}$ *of finite sets and set inclusions, given three sets* B*,* C *and* D *such that* $B \subseteq D$ *and* $C \subseteq D$*, the pullback is given by the set* $A := B \cap C$*.*

Example A.3.2. *In the category* **Set** *of sets and functions, given three sets* B*,* C *and* D *and two functions* $f : B \to D$ *and* $g : C \to D$*, the pullback is given by the set* A *defined as*

$$A := \{(b, c) \in B \times C \mid f(b) = g(c)\}$$

and two homomorphisms $g' : A \to B$ *and* $f' : A \to C$ *defined as the projections of* $B \times C$ *into* B *and* C*, i.e. for all* $(b, c) \in B \times C$ $g'(b, c) = b$ *and* $f'(b, c) = c$*.*

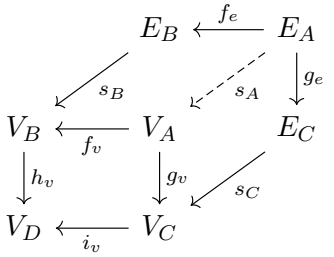
Example A.3.3. *In the category* **Dict** *of dictionaries and dictionary inclusions, given three dictionaries* $d_B : V_B \to K_B$*,* $d_C : V_C \to K_C$ *and* $d_D : V_D \to K_D$*, the pullback is defined by a dictionary* $d_A : V_B \cap V_C \to K_B \cap K_C$*, where* $\forall v \in V_B \cap V_C$ $d_A(v) := d_B(v)$ *(or equivalently* $d_A(v) := d_C(v)$*). It is easy to verify that such a* d_A *exists and is unique by the universal property of* $K_B \cap K_C$ *as the pullback in the category* **Set** *illustrated in the following commutative diagram:*



(A.3)

Example A.3.4. *For the categories* **SimpGrph**, **Grph**, *as well as* $\mathbf{SimpGrph}_{attrs}$ *and* $\mathbf{Grph}_{attrs}$*, the pullback can be constructed separately using pullbacks in* **Set** *for nodes and edges, and in* **Dict** *for attributes. The source and target functions in the definitions of objects from* **Grph***, as well as the attribute-assigning ones from* $\mathbf{SimpGrph}_{attrs}$ *and* $\mathbf{Grph}_{attrs}$*, are uniquely determined by the universal properties of pullbacks constructed in* **Set***. For instance, source functions can be obtained in the following way. Let* $B = (V_B, E_B, s_B, t_B)$*,* $C = (V_C, E_C, s_C, t_C)$ *and* $D = (V_D, E_D, s_D, t_D)$ *be three graphs and* $h : B \to D$ *and* $i : C \to D$ *be two homomorphisms defined by the pairs* $(h_v : V_B \to V_D, h_e : E_B \to E_D)$ *and* $(i_v : V_C \to V_D, i_e : E_C \to E_D)$*. Having*

constructed V_A, E_A, $f_v : V_A \to V_B$, $f_e : E_A \to E_B$, $g_v : V_A \to V_C$ *and* $g_e : E_A \to E_C$ *using pullback constructions in* **Set**, *we can obtain the unique homomorphism* $s_A : E_A \to V_A$ *applying the universal property of the pullback that constructs* V_A *as in the following diagram.*



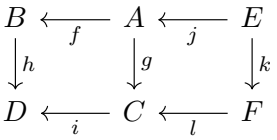
(A.4)

The homomorphism $t_A : E_A \to V_A$ *can be found in the analogous way.*

In the categories of simple graphs, to construct the pullback object we need to handle a subtlety which arises due to the fact that graph homomorphims defined by maps of nodes, while maps of edges are given implicitly and are more similar to set inclusions in $\mathbf{Set}_{fin}$. *Therefore, we cannot construct our set* E_A *in the same way as in Example A.3.2. Having constructed the set of nodes* V_A *of the pullback object* A *and the homomorphisms* $g' : V_A \to V_B$ *and* $f' : V_A \to V_C$, *the set of edges is defined as follows:*

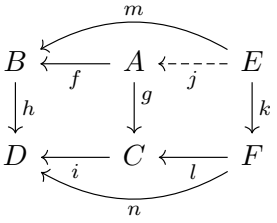
$$E_A := \{(u, v) \in V_A \times V_A \mid (g'(u), g'(v)) \in E_B \text{ and } (f'(u), f'(v)) \in E_C\}.$$

Lemma A.3.2. (*Pasting lemma for pullbacks*) In the diagram below, let $B \leftarrow f - A - g \rightarrow C$ be the pullback from h and i. $B \leftarrow f \circ j - E - k \rightarrow F$ is the pullback from h and $i \circ l$ if and only if $A \leftarrow j - E - k \rightarrow F$ is the pullback from g and l. In other words, if the left square in the diagram is a pullbacks, then the outer square is also one, if and only if the right square is also a pullback.



(A.5)

Corollary A.3.3. Let $n = i \circ l$, $B \leftarrow f - A - g \rightarrow C$ be the pullback from h and i and $B \leftarrow m - E - k \rightarrow F$)— the pullback from h and n, then there exists a unique $j : E \to A$ such that $m = f \circ j$ and $g \circ j = l \circ k$, and, moreover, the square formed by $g \circ j$ and $l \circ k$, (right inner square in the following diagram) is a pullback.



(A.6)

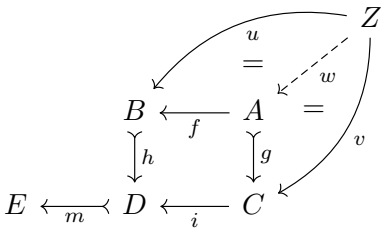
Lemma A.3.4. (*Pullbacks reflect monos*) In the diagram below, let $B \leftarrow f - A - g \rightarrow C$ be the pullback from h and i. If h is a monomorphism, g is also a monomorphism.

$$\begin{array}{ccc} B & \xleftarrow[f]{} & A \\ \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle g} \\ D & \xleftarrow[i]{} & C \end{array} \tag{A.7}$$

Lemma A.3.5. (*Pullback post-composed with a mono*) In the diagram below, let $B \leftarrow f - A - g \rightarrow C$ be the pullback from h and i and $m : D \rightarrowtail E$ be a mono, then $B \leftarrow f - A - g \rightarrow C$ is also the pullback from $m \circ h$ and $m \circ i$.

$$\begin{array}{ccccc} & & B & \xleftarrow[f]{} & A \\ & & \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle g} \\ E & \xleftarrow[m]{} & D & \xleftarrow[i]{} & C \end{array} \tag{A.8}$$

Proof. To prove this lemma we need to demonstrate two things: first, that the square formed by $m \circ h \circ f$ and $m \circ i \circ g$ commutes, which is immediate; second, that this square has the universal property of a pullback. Let $B \leftarrow u - Z - v \rightarrow C$ be a span such that $m \circ h \circ u = m \circ i \circ v$ (as in the diagram below). The homomorphism m being a mono implies that $h \circ u = i \circ v$, which allows us to use the universal property of the pullback square from the diagram and show that there exists a unique arrow $w : Z \rightarrow A$ such that all the triangles in the following diagram commute. This concludes our proof.



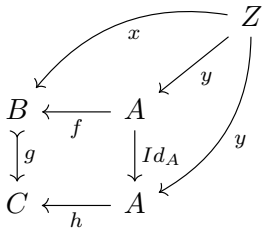
(A.9)

□

Lemma A.3.6. (*Pullback triangle from a mono*) The square in the following diagram is a pullback, if $g \circ f = h$ and g is a mono.

$$\begin{array}{ccc} B & \xleftarrow[f]{} & A \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle Id_A} \\ C & \xleftarrow[h]{} & A \end{array} \tag{A.10}$$

Proof. Because $g \circ f = h$, for any object Z and two homomorphisms $x : Z \to B$ and $y : Z \to A$ such that $g \circ x = h \circ y$, $g \circ x = g \circ f \circ y$, and g being a mono implies that $x = f \circ y$. Trivially y is the unique homomorphism for which $y = Id_A \circ y$.

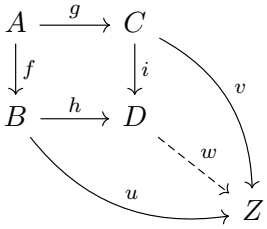


(A.11)

□

A.4 Pushouts

Definition A.4.1. The *pushout* of a span $B \leftarrow f - A - g \rightarrow C$ is given by a co-span $B - h \rightarrow D \leftarrow i - C$ for which $h \circ f = i \circ g$ and which has the following universal property: for all $B - u \rightarrow Z \leftarrow v - C$ such that $u \circ f = v \circ g$ there exists a unique $w : D \to Z$ such that $u = w \circ h$ and $v = w \circ i$.



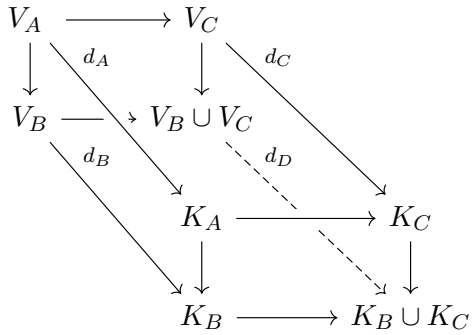
(A.12)

Example A.4.1. *In the category* $\mathbf{Set}_{fin}$ *of finite sets and set inclusions, given three sets A, B and C such that* $A \subseteq B$ *and* $A \subseteq C$*, the pushout is given by the set* $D := B \cup C$*.*

Example A.4.2. *In the category* **Set** *of sets and functions, given three sets A, B and C and two functions* $f : A \to B$ *and* $g : A \to C$*, the pushout is given by the set* $D := (B \uplus C)_{\approx}$*, i.e. the quotient set of the disjoint union of B and C, modulo the equivalence relation* $\approx$*, where* $\approx := \{(b, c) \in B \times C \mid \exists a \in A : f(a) = b \text{ and } g(a) = c\}$*. The two homomorphisms* $g' : B \to D$ *and* $f' : C \to D$ *are then defined in such a way that they map elements from B and C to the corresponding equivalence classes [21].*

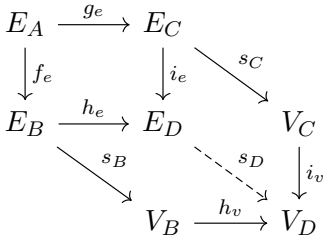
Example A.4.3. *In the category* **Dict** *of dictionaries and dictionary inclusions, given three dictionaries* $d_A : V_A \to K_A$*,* $d_B : V_B \to K_B$ *and* $d_C : V_C \to K_C$*, the pushout is defined by a dictionary* $d_D : V_B \cup V_C \to K_B \cup K_C$*, where* $\forall v \in V_B \; d_D(v) := d_B(v)$ *and* $\forall v \in V_C \; d_D(v) := d_C(v)$*. It is easy to verify that such a* d_D *exists and is unique by the universal property of* $V_B \cup V_C$ *as the pullback in the category* **Set** *illustrated in the following commutative*

diagram:



(A.13)

Example A.4.4. *Similarly to pullbacks, for the categories* **SimpGrph**, **Grph**, $\mathbf{SimpGrph}_{attrs}$ *and* $\mathbf{Grph}_{attrs}$, *a pushout can be constructed separately using pushouts in* **Set** *for nodes and edges, and in* **Dict** *for attributes. The source and target functions in the definitions of objects from* **Grph**, *as well as the attribute-assigning ones from* $\mathbf{SimpGrph}_{attrs}$ *and* $\mathbf{Grph}_{attrs}$, *are uniquely determined by the universal properties of pushouts constructed in* **Set**. *For instance, source functions can be obtained in the following way. Let* $A = (V_A, E_A, s_A, t_A)$, $B = (V_B, E_B, s_B, t_B)$ *and* $C = (V_C, E_C, s_C, t_C)$ *be three graphs and* $f : A \to B$ *and* $g : A \to C$ *be two homomorphisms defined by the pairs* $(f_v : V_A \to V_B, f_e : E_A \to E_B)$ *and* $(g_v : V_A \to V_C, g_e : E_A \to E_C)$. *Having constructed* V_D, E_D, $h_v : V_B \to V_D$, $h_e : E_B \to E_D$, $i_e : V_C \to V_D$ *and* $i_e : E_C \to E_D$ *using pushout constructions in* **Set**, *we can obtain the unique homomorphism* $s_D : E_D \to V_D$ *applying the universal property of the pushout that constructs* E_D *as in the following diagram.*



(A.14)

The homomorphism $t_D : E_D \to V_D$ *can be found in the analogous way.*

As in the case of pullbacks, a subtlety arises due to the fact that graph homomorphims in **SimpGrph** *are defined by maps of nodes, while maps of edges are given implicitly. We cannot define the equivalence relation* $\approx$ *in the same way as in Example A.4.2. Having constructed the set of nodes* V_D *of the pushout object* D *and the homomorphisms* $g' : V_B \to V_D$ *and* $f' : V_C \to V_D$, *the set of edges is defined as follows:*

$$E_D := \{(u, v) \in V_D \times V_D \mid (\exists (u', v') \in E_B : g'(u') = u \text{ and } g'(v') = v) \text{ or } (\exists (u', v') \in E_C : f'(u') = u \text{ and } f'(v') = v)\}.$$

Lemma A.4.2. (*Pasting lemma for pushouts*) In the diagram below, let $B \xrightarrow{h} D \xleftarrow{i} C$ be the pushout from f and g, then $B \xrightarrow{k \circ h} F \xleftarrow{l} E$ is the pushout from f and $j \circ g$, if and only if $D \xrightarrow{k} F \xleftarrow{l} E$ is the pushout from i and j. In other words, if the left square in the following diagram is the pushout, the outer square is also one if and only if the right square is the pushout.

$$\begin{array}{ccccc} A & \xrightarrow{g} & C & \xrightarrow{j} & E \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle l} \\ B & \xrightarrow{h} & D & \xrightarrow{k} & F \end{array} \tag{A.15}$$

Corollary A.4.3. In the diagram below, let $m = j \circ g$, $B \xrightarrow{h} D \xleftarrow{i} C$ be the pushout from f and g and $B \xrightarrow{n} F \xleftarrow{l} E$—the pushout from f and m, then there exists a unique $k : D \to F$ such that $n = k \circ i$ and $k \circ h = l \circ j$, and, moreover, the square formed by $k \circ h$ and $l \circ j$, (right inner square in the following diagram) is a pushout.

$$\begin{array}{ccccc} & & \overset{m}{\frown} & & \\ A & \xrightarrow{g} & C & \xrightarrow{j} & E \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle l} \\ B & \xrightarrow{h} & D & \overset{k}{\dashrightarrow} & F \\ & & \underset{n}{\smile} & & \end{array} \tag{A.16}$$

Lemma A.4.4. (*Pushouts reflect epis*) Let $B \xrightarrow{h} D \xleftarrow{i} C$ be the pushout from f and g. If f is a epimorphism, i is also an epimorphism.

$$\begin{array}{ccc} A & \xrightarrow{g} & C \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle i} \\ B & \xrightarrow{h} & D \end{array} \tag{A.17}$$

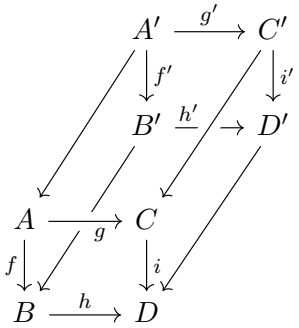
Lemma A.4.5. (*Pushout pre-composed with an epi*) In the following diagram, let $B \xrightarrow{h} D \xleftarrow{i} C$ be the pushout from f and g and $e : E \twoheadrightarrow A$ be an epi, then $B \xrightarrow{h} D \xleftarrow{i} C$ is also the pushout from $f \circ e$ and $g \circ e$.

$$\begin{array}{ccccc} E & \xrightarrow{e} & A & \xrightarrow{g} & C \\ & & \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle i} \\ & & B & \xrightarrow{h} & D \end{array} \tag{A.18}$$

Proof. The proof is similar to the one for Lemma A.3.5 using the definition of epi and the universal property of pushouts. □

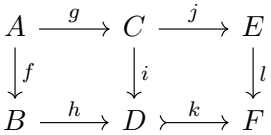
Definition A.4.6. (*Stable pushouts*) Let $B \xrightarrow{h} D \xleftarrow{i} C$ be the pushout from f and g, this pushout is *stable* if for every commutative cube as in the diagram below, the back face is a

pushout, if the left, top, right and bottom faces are pullbacks.



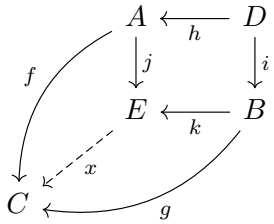
(A.19)

Lemma A.4.7. (*Pushout splitting*) Let $B \leftarrow f - A - g \rightarrow C$ be the pullback from h and i and $D \leftarrow i - C - j \rightarrow E$ be the pullback from k and l (i.e. the two inner squares in the following diagram are pullbacks), $B - k \circ h \rightarrow F \leftarrow l - E$ be the pushout from f and $j \circ g$ that is stable (i.e. the outer square is a pushout) and the homomorphism k be a mono, then $B - h \rightarrow D \leftarrow i - C$ is also the pushout from f and g and $D - k \rightarrow F \leftarrow l - E$ is the pushout from i and j (i.e. the two inner squares are also pushouts, see Lemma 4, [29]).



(A.20)

Definition A.4.8 (*Pushout factorization of a pullback*)**.** Let $f : A \rightarrow C$ and $g : B \rightarrow C$ be two arrows, $A \leftarrow h - D - i \rightarrow B$ be the pullback from f and g and $A - j \rightarrow E \leftarrow k - B$ be the pushout from h and i. The unique arrow $x : E \rightarrow C$ that renders the following diagram commutative obtained by the universal property of the pushout is called *pushout factorization* of the pullback of f and g.

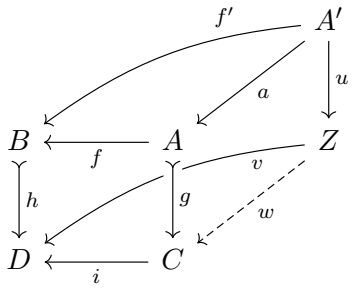


(A.21)

A.5 Pullback complements

In this appendix we define the notion of a final pullback complement and some of its useful properties. The interested reader can find more details in [65].

Definition A.5.1. The *final pullback complement* of a pair of composable arrows $A \xrightarrow{f} B \xrightarrow{h} D$ is another pair of composable arrows $A \xrightarrow{g} C \xrightarrow{i} D$ for which $h \circ f = i \circ g$, forming a square that is a pullback, and which has the following universal property: for all $B \xleftarrow{f'} A' \xrightarrow{u} Z \xrightarrow{v} D$, for which $h \circ f' = v \circ u$ and the formed square is a pullback, and for every arrow $a : A' \to A$ such that $f' = f \circ a$ there exists a unique $w : Z \to C$ such that $g \circ a = w \circ u$ and $v = i \circ w$.



(A.22)

In the categories of our interest the final pullback complement always exists if h is a monomorphism, then by the fact that pullbacks preserve monos g is always a mono.

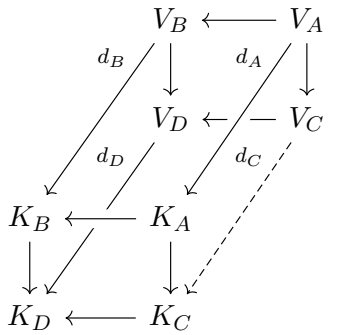
Example A.5.1. *In the category* $\mathbf{Set}_{fin}$ *of finite sets and set inclusions, given three sets A, B and D such that* $A \subseteq B \subseteq D$*, the final pullback complement is given by the set* $C := D \setminus (B \setminus A)$*.*

Example A.5.2. *In the category* **Set** *of sets and functions, given three sets A, B and D and two functions* $f : A \to B$ *and* $g : B \rightarrowtail D$*, the final pullback complement is given by the set C defined as*

$$C := A \cup \{d \in D \mid \nexists b \in B : g(b) = d\}.$$

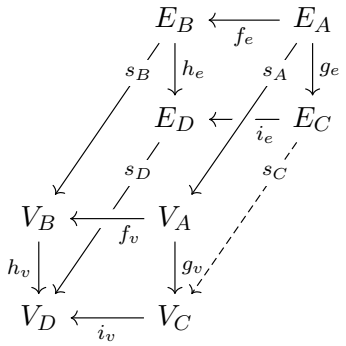
The homomorphism $g' : A \to C$ *is induced by the identity morphism* Id_A*, and* $f' : C \to D$ *is defined as* $f'(a) = g \circ f(a)$ *for all* $a \in A$ *and* $f'(d) = d$ *for all* $d \in D$ *such that* $\nexists b \in B : g(b) = d$*.*

Example A.5.3. *In the category* **Dict** *of dictionaries and dictionary inclusions, given three dictionaries* $d_A : V_A \to K_A$*,* $d_B : V_B \to K_B$ *and* $d_D : V_D \to K_D$*, the final pullback complement is defined by a dictionary* $d_C : V_C \to K_C$*, where* $V_C := V_D \setminus (V_B \setminus V_A)$*,* $K_C := K_D \setminus (K_B \setminus K_A)$*,* $\forall v \in V_A \; d_C(v) := d_A(v)$ *and* $\forall v \in V_D \setminus V_B \; d_C(v) := d_D(v)$*. It is easy to verify that such a* d_C *exists and is unique by the universal property of* $K_D \setminus (K_B \setminus K_A)$ *as the final pullback complement in the category* **Set** *illustrated in the following commutative diagram:*



(A.23)

Example A.5.4. *For the categories* **SimpGrph**, **Grph**, *as well as* $\mathbf{SimpGrph}_{attrs}$ *and* $\mathbf{Grph}_{attrs}$, *the final pullback complement can be constructed separately using final pullbacks complements in* **Set** *for nodes and edges, and in* **Dict** *for attributes. The source and target functions in the definitions of objects from* **Grph**, *as well as the attribute assigning ones from* $\mathbf{SimpGrph}_{attrs}$ *and* $\mathbf{Grph}_{attrs}$, *are uniquely determined by the universal properties of constructed final pullback complements. For instance, source functions can be obtained in the following way. Let* $A = (V_A, E_A, s_A, t_A)$, $B = (V_B, E_B, s_B, t_B)$ *and* $D = (V_D, E_D, s_D, t_D)$ *be three graphs and* $f : A \to B$ *and* $h : B \to C$ *be two homomorphisms defined by the pairs* $(f_v : V_A \to V_B, f_e : E_A \to E_B)$ *and* $(h_v : V_B \to V_C, h_e : E_B \to E_C)$. *Having constructed* V_C, E_C, $g_v : V_A \to V_C$, $g_e : E_A \to E_C$, $i_v : V_C \to V_D$ *and* $i_e : E_C \to E_D$ *using final pullback complement constructions in* **Set**, *we can obtain the unique homomorphism* $s_C : E_C \to V_C$ *applying the universal property of the final pullback complement that constructs* V_C *as in the following diagram.*



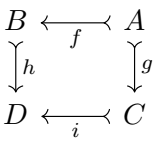
(A.24)

The homomorphism $t_C : E_C \to V_C$ *can be found in the analogous way.*

Similarly to pullbacks and pushouts, we need to additionally define the set of edges of the final pullback complement object C *in* **SimpGrph**. *Having constructed the set of nodes* V_C *and the homomorphisms* $g' : V_A \to V_C$ *and* $f' : V_C \to V_D$, *the set of edges is defined as follows:*

$$E_C := E_A \cup \{(u, v) \in V_C \times V_C \mid ((f'(u), f'(v)) \in E_D \text{ and } (f'(u) \notin img(g) \text{ or } f'(v) \notin img(g))) \text{ or } (g^{-1}(f'(u)), g^{-1}(f'(v))) \notin E_B\}.$$

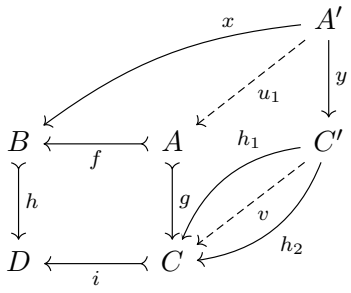
Lemma A.5.2. (*Pullback complement preserve monos*) Let $A \rightarrowtail_g C \xrightarrow{i} D$ be the final pullback complement of f and h (as in the diagram below), and f be monomorphisms, then i is also a monomorphism.



(A.25)

Proof. To prove this lemma we would like to show that for any object C' and two homomorphisms $h_1 : C' \to C$ and $h_2 : C' \to C$ such that $i \circ h_1 = i \circ h_2$, it holds that $h_1 = h_2$.

First, construct the pullback $B \leftarrow x - A' - y \rightarrow C'$ from h and $i \circ h_1$. By the universal property of the pullback (corresponding to the inner square in the diagram below) there exists a unique $u_1 : A' \to A$ such that $f \circ u_1 = x$ and $g \circ u_1 = y \circ h_1$. Now, we can use the universal property of the final pullback complement that constructed C, i.e. from the fact that the outer square is a pullback, and that $f \circ u_i = x$ follows that there exists a unique homomorphism $v : C' \to C$ such that $g \circ u_1 = v \circ y$ and $i \circ v = i \circ h_1$.



(A.26)

Clearly h_1 satisfies both equations, which implies that $v = h_1$. Meanwhile our second homomorphism h_2 satisfies only (b).

Now, let us construct the pullback $B \leftarrow x' - A'' - y' \rightarrow C'$ from h and $i \circ h_2$. By its universal property there exists a unique $u_2 : A'' \to A$ such that $f \circ u_2 = x$ and $g \circ u_2 = y' \circ h_2$. Since $i \circ h_2 = i \circ h_1$ we have that $x = x'$, and therefore $f \circ u_2 = f \circ u_1$. Recall that f is a monomorphism, which implies that $u_1 = u_2$, and makes h_2 also satisfy (a). Therefore $h_2 = v = h_1$, which completes our proof.

□

Lemma A.5.3 (*Horizontal pasting lemma for final pullback complements*)**.** In the following diagram, let $A \rightarrowtail g \rightarrow C - i \rightarrow D$ be the final pullback complement of f and h, then $E \rightarrowtail k \rightarrow F - i \circ l \rightarrow D$ is the final pullback complement of $f \circ j$ and h, if and only if $E \rightarrowtail k \rightarrow F - l \rightarrow C$ is the final pullback complement of j and g.

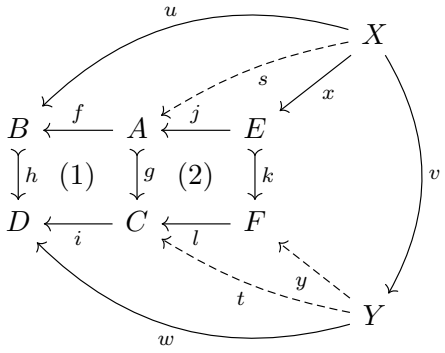
$$\begin{array}{ccccc} B & \xleftarrow{f} & A & \xleftarrow{j} & E \\ \downarrow{\scriptstyle h} & (1) & \downarrow{\scriptstyle g} & (2) & \downarrow{\scriptstyle k} \\ D & \xleftarrow[i]{} & C & \xleftarrow[l]{} & F \end{array} \tag{A.27}$$

Proof. ($\implies$) By the pasting lemma for pullbacks the outer square in Diagram A.27 is a pullback (the square obtained from the composition of the squares denoted with (1) and (2)). Now we need to prove that this outer square has the universal property of a final pullback complement, i.e. that for any X, Y, $u : X \to B$, $v : X \to Y$, $w : Y \to D$ such that the outer square in Diagram A.28 is the pullback and the homomorphism $x : X \to E$ such that $u = f \circ j \circ x$, there exists a unique homomorphism $y : Y \to F$ such that $y \circ v = k \circ x$ and $i \circ l \circ y = w$.

First of all, by the universal property of the pullback given by the square (1) and the fact that $u = f \circ j \circ x$ and $h \circ f = i \circ g$, there exists a unique homomorphism $s : X \to A$ such that $u = f \circ s$ and $g \circ j \circ x = g \circ s$ (as in the diagram below). This also implies that $s = j \circ x$ (as g is a

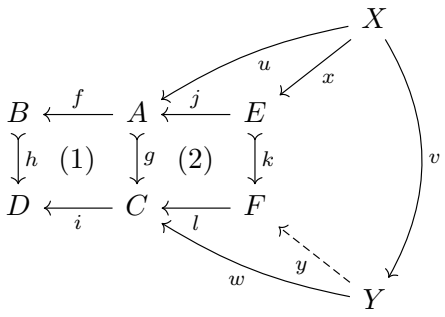
mono). We can also use the universal property of the final pullback complement $A \rightarrowtail C \to D$ and show that there exists a unique homomorphism $t : Y \to C$ such that $g \circ s = t \circ v$ and $w = i \circ t$. Now we need to show that the square formed by $g \circ s$ and $t \circ v$ is a pullback, which follows immediately from the corollary A.3.3 and the fact that $w = i \circ t$.

Finally, by the the universal property of the final pullback complement that gives $E \rightarrowtail F \to C$ there exists a unique homomorphism $y : Y \to F$ such that $k \circ x = y \circ v$ and $t = l \circ y$. The uniqueness of y implies that there cannot exist another homomorphism $y \neq y'$ such that $i \circ l \circ y' = w$, which concludes this part of the proof.



(A.28)

($\Longleftarrow$) To prove the converse, knowing that the outer square in Diagram A.27, the composition of (1) and (2), is the final pullback complement, we need to show that (2) has the universal property of a final pullback complement, i.e. that for any X, Y, $u : X \to A$, $v : X \to Y$, $w : Y \to C$ such that the square formed by $g \circ u$ and $w \circ v$ is the pullback and the homomorphism $x : X \to E$ such that $u = j \circ x$, there exists a unique homomorphism $y : Y \to F$ such that $y \circ v = k \circ x$ and $l \circ y = w$.



(A.29)

The square (1) and the one formed by $g \circ u$ and $w \circ v$ are pullbacks, therefore pasting lemma for pullbacks tells us that the composition of these square is a pullback. This allows us to apply the universal property of the final pullback complement $E \rightarrowtail F \to D$ and show that there exists a unique homomorphism $y : Y \to F$ such that $k \circ x = y \circ v$ and $i \circ w = i \circ l \circ y$. To finalize the prove we notice that the uniqueness of y implies that, moreover, y is the unique homomorphism for which $w = l \circ y$.

□

Lemma A.5.4 (*Vertical pasting lemma for final pullback complements*)**.** Let $A \rightarrowtail_g \to C \to_i D$ be the final pullback complement of f and h, then:

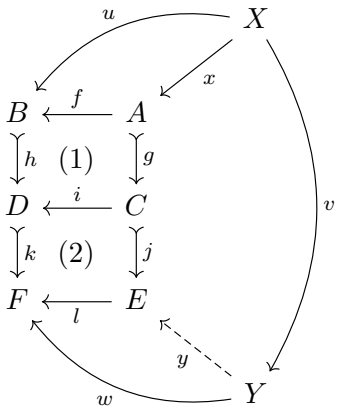
(a) if $C \rightarrowtail_j \to E \to_l F$ is the final pullback complement of i and k, then $A \rightarrowtail_{j \circ g} \to E \to_l F$ is also the final pullback complement of f and $k \circ h$;

(b) if $A \rightarrowtail_{j \circ g} \to E \to_l F$ is the final pullback complement of f and $k \circ h$ and the square denoted with (2) in Diagram A.30 is a pullback, then $C \rightarrowtail_j \to E \to_l F$ is the final pullback complement of i and k.

$$\begin{array}{ccc} B & \xleftarrow{f} & A \\ \downarrow{\scriptstyle h} & (1) & \downarrow{\scriptstyle g} \\ D & \xleftarrow{i} & C \\ \downarrow{\scriptstyle k} & (2) & \downarrow{\scriptstyle j} \\ F & \xleftarrow[l]{} & E \end{array} \tag{A.30}$$

Proof. (a) As in the previous lemma we know that the outer square in diagram A.30 is the pullback. Now, as before, we need to prove the universal property, i.e. that for any X, Y, $u : X \to A$, $v : X \to Y$, $w : Y \to E$ such that the outer square in diagram A.31 is the pullback and the homomorphism $x : X \to C$ such that $u = f \circ x$, there exists a unique homomorphism $y : Y \to F$ such that $y \circ v = j \circ g \circ x$ and $w = l \circ y$.

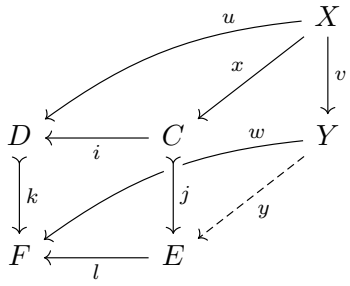
This proof is more straightforward than its horizontal counterpart. Performing a simple diagram chase we can verify that all the conditions to directly apply the universal property of the final pullback complement $C \rightarrowtail_j \to E \to_l F$ are satisfied and we obtained the desired unique arrow y.



(A.31)

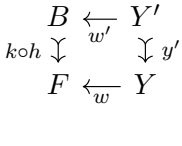
(b) Now, knowing that the square (2) in A.30 is a pullback, we need to prove that $C \rightarrowtail_j \to E \to_l F$ is the final pullback complement of i and k. We need to show that the square (2) has the universal property of the final pullback complement as in the diagram below, i.e. for any $D \leftarrow_u - X \to_u Y \to_w F$, for which $k \circ u = w \circ v$ and the formed square is a pullback, and for

any arrow $x : X \to C$ such that $u = i \circ x$ there exists a unique $y : Y \to E$ such that $w = l \circ y$ and $y \circ v = j \circ x$.

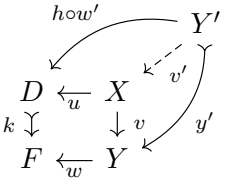


(A.32)

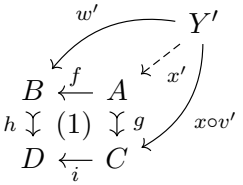
First of all, we can find the pullback $B \leftarrow w' - Y' - y' \rightarrow Y$ from $k \circ h$ and w (Diagram A.33). We can now apply the universal property of the pullback $D \leftarrow u - X - v \rightarrow Y$ and show that there exists a unique $v' : Y' \to X$ such that $h \circ w' = u \circ v'$ and $y' = v \circ v'$ as in Diagram A.34. At this step, we again apply the universal property of the pullback square (1) (Diagram A.35) and obtain a unique arrow $x' : Y' \to A$ such that $w' = f \circ x'$ and $x \circ v' = g \circ x'$.



(A.33)

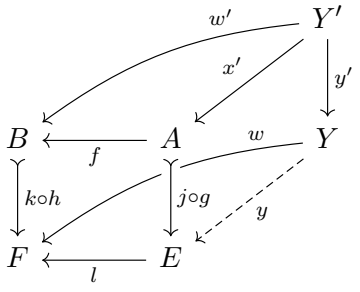


(A.34)

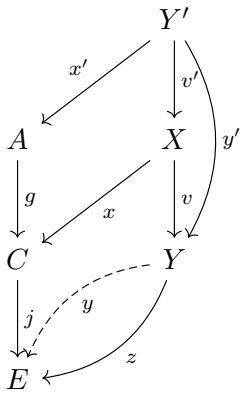


(A.35)

The square in A.33 is a pullback by construction and $w' = f \circ x'$, therefore we can use the universal property of the final pullback complement given by the composition of squares (1) and (2) as in Diagram A.36. It states that there exists a unique $y : Y \to E$ such that $w = l \circ y$ and $j \circ g \circ x' = y \circ y'$. For any homomorphism z such that $z \circ v = j \circ x$ by a chase of Diagram A.37 it would also hold $z \circ v \circ v' = j \circ x \circ v' = j \circ g \circ x' = z \circ y'$, which implies that $z = y$ by the uniqueness of y, which concludes our proof.



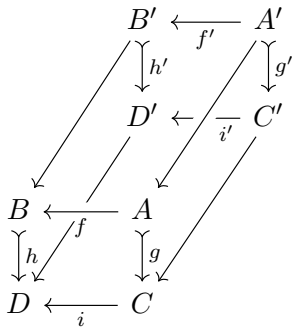
(A.36)



(A.37)

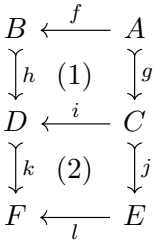
□

Lemma A.5.5 (*Stability of final pullback complement*)**.** If all the faces of a commutative cube as in Diagram A.38 are pullbacks and the front face is a final pullback complement ($A \rightarrowtail g \rightarrow C \rightarrow iD$ is the final pullback complement to f and h), then the back face is also a final pullback complement (see Lemma 1, [29]).



(A.38)

Lemma A.5.6. If $D \leftarrow i - C \rightarrowtail j \rightarrow E$ is the pullback from k and l, $A \rightarrowtail j \circ g \rightarrow E - l \rightarrow F$ is the final pullback complement of f and $k \circ h$ and the arrow k is a mono, then $A \rightarrowtail g \rightarrow C - i \rightarrow D$ is the final pullback complement of f and h and $C \rightarrowtail j \rightarrow E - l \rightarrow F$ is the final pullback complement of i and k.



(A.39)

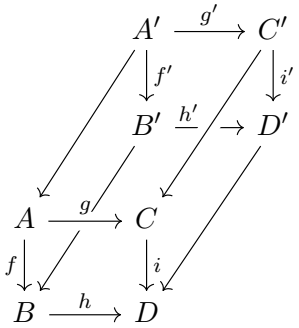
Proof. See the proof of Lemma 38 in [65]. □

A.6 Adhesive categories

In this section we define the notion of adhesive categories. These are the categories where pushouts along monomorphisms 'behave well'. Formalized in [61], adhesivity is characteristic of many categories of interest to us, such as **SimpGrph**, **Grph**, as well as $\mathbf{SimpGrph}_{attrs}$ and $\mathbf{Grph}_{attrs}$. We first define the notion of a van Kampen square, which defines the notion of a 'well-behaved' pushout.

Definition A.6.1 (*van Kampen square*)**.** A *van Kampen* square (VK-square) is a pushout such that given a commutative cube as in Diagram A.40, where this pushout forms the front face and

the left and the top faces are pullbacks, the right and bottom faces of the cube are pullbacks if and only if the back face is a pushout.

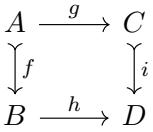


(A.40)

Definition A.6.2 (*Adhesive category [61]*)**.** A category **C** is adhesive if:

- **C** has all pushouts along monos;
- **C** has all pullbacks;
- pushouts along monos are VK-squares.

Lemma A.6.3 (*Pushouts preserve monos*)**.** In the diagram below, let $B \rightarrow^{h} D \leftarrow^{[-i]} C$ be the pushout from f and g. If f is a monomorphism, so is i.



(A.41)

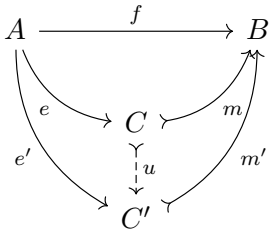
Lemma A.6.4. In adhesive categories pushouts along monomorphisms are also pullbacks.

A.7 Factorizations

Definition A.7.1. Given a homomorphism $f : A \to B$ in a category **C** the *image factorization* of f is given by the object C and the monomorphism $m : C \to B$ satisfying the following universal property:

(a) there exists a homomorphism $e : A \to C$ such that $f = m \circ e$;

(b) for any object C' with a homomorphism $e' : A \to C'$ and a monomorphism $m' : C' \to B$ such that $f = m' \circ e'$, there exists a unique homomorphism $u : C \to C'$ such that $e' = u \circ e$ and $u \circ m' = m$.

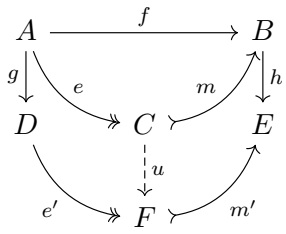


(A.42)

Remark A.7.2. The homomorphism v is a monomorphism.

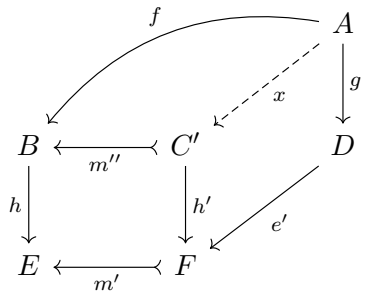
Proposition A.7.3. If **C** has all equalizers then the homomorphism e in the factorization $f = m \circ e$ is an epimorphism.

Lemma A.7.4. Given a commutative square as the square formed by $h \circ f$ and $m' \circ e' \circ g$ in the diagram below with m' being a mono, and the image factorization of f given by $A \twoheadrightarrow C \rightarrowtail B$, there exists a unique homomorphism $u : C \to F$ such that $u \circ e = e' \circ g$ and $m' \circ u = h \circ m$.

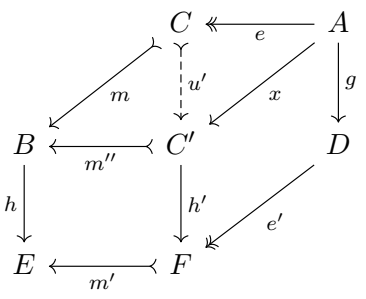


(A.43)

Proof. We first proceed by constructing the pullback $B \leftarrow m'' \prec C' \xrightarrow{h'} F$ from h and m' as in Diagram A.44. We can show that by the universal property of this pullback there exists a unique arrow $x : A \to C'$ that renders our diagram commutative. We also note that in the categories of our interest pullbacks preserve monos, which means that the constructed $m'' : C' \to B$ is a mono. This allows us to use the universal property of the image factorization C as in Diagram A.45 and show that there exists a unique $u' : C \to C'$ that renders this diagram commutative. We can construct a homomorphism $u : C \to F$ as a composition $h' \circ u'$ and by a diagram chase we can show that $u \circ e = e' \circ g$ and $m' \circ u = h \circ m$. Now, we still need to prove that u is a unique homomorphism that satisfies these properties.



(A.44)



(A.45)

First, suppose that there exists $\bar{u} \neq u$ such that $\bar{u} \circ e = e' \circ g$. By the statement of our lemma $h \circ f = m' \circ e' \circ g$, therefore $h \circ f = m' \circ e' \circ g = m' \circ \bar{u} \circ e = m' \circ u \circ e$, and m' being a mono implies that $u = \bar{u}$, which leads us to a contradiction. Now, let us suppose that there exists $\bar{u} \neq u$ such that $m' \circ \bar{u} = h \circ m$. Again, by the statement of our lemma $h \circ f = m' \circ e' \circ g$ and $f = m \circ e$. Therefore, $m' \circ e' \circ g = h \circ m \circ e = m' \circ \bar{u} \circ e = m' \circ u \circ e$, and e being an epi implies that $u = \bar{u}$, which again leads us to a contradiction. This concludes our proof and shows the uniqueness of such u.

□

Example A.7.1. *In the category* $\mathbf{Set}_{fin}$ *of finite sets and set inclusions, given two sets* A *and* B *such that* $A \subseteq B$*, the image factorization is simply given by the set* $C := A$*.*

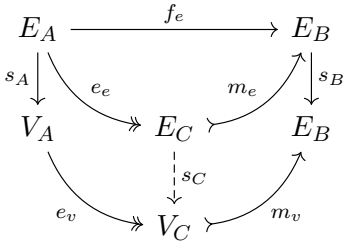
Example A.7.2. *In the category* **Set** *of sets and functions, given two sets* A *and* B *and a function* $f : A \to B$*, the image factorization is given by the usual epi-mono factorization, i.e. the set* C *defined as*

$$C := \{b \in B \mid \exists a \in A : f(a) = b\}.$$

For all $a \in A$*, the homomorphism* $e : A \to C$ *is defined as* $e(a) = f(a)$ *and for all* $b \in B$ *such that* $b \in C$*,* $m(b) = b$*.*

Example A.7.3. *In the category* **Dict** *of dictionaries and dictionary inclusions, given two dictionaries* $d_A : V_A \to K_A$ *and* $d_B : V_B \to K_B$ *such that* $d_A \subseteq d_B$*, the image factorization is given by the dictionary* $d_C : V_C \to K_C$*, where* $V_C := V_A$*,* $K_C := K_A$ *and* $d_C := d_A$*.*

Example A.7.4. *For the categories* **SimpGrph**, **Grph**, *as well as* $\mathbf{SimpGrph}_{attrs}$ *and* $\mathbf{Grph}_{attrs}$*, the image factorization can be constructed separately using image factorizations in* **Set** *for nodes and edges, and in* **Dict** *for attributes. In the case of* **Grph** *and* $\mathbf{Grph}_{attrs}$ *we can reconstruct the source and target functions in the following way. Let* $A = (V_A, E_A, s_A, t_A)$ *and* $B = (V_B, E_B, s_B, s_B)$ *be two graphs and* $f : A \to B$ *be a homomorphism given by a pair* $f_v : V_A \to V_B$ *and* $f_e : E_A \to E_B$*. Using image factorization in* **Set** *we construct* V_C*,* E_C *and homomorphisms* $e_v : V_A \to V_C$*,* $e_e : E_A \to E_C$*,* $m_v : V_C \rightarrowtail V_B$ *and* $m_e : E_C \rightarrowtail E_B$*. It is not hard to verify that the outer square in the following diagram commutes, therefore, we can use Lemma A.7.4 to show that there exists a unique homomorphism* $s_C : E_C \to V_C$ *that renders the diagram commutative. The same argument can be applied to show the existence and the uniqueness of* $t_C : E_C \to V_C$*. Thus, we can construct the image factorization object* $C = (V_C, E_C, s_C, t_C)$*.*



(A.46)

Appendix B

Proofs

Proof of Claim 2.1.3. To prove this claim we need to recall the definition of pushouts in the category of simple graphs from Example A.4.4 and final pullback complements from A.5.4. Let us first construct the object $G^- = (V_{G^-}, E_{G^-})$ corresponding to the pullback complement $L^- \rightarrowtail m^- \rightarrow G^- - g^- \rightarrow G$ to r^- and m. Its sets of nodes and edges are defined as follows:

$$V_{G^-} := V_{L^-} \uplus \{v \in V_G \mid \nexists w \in V_L : m(w) = v\}$$

$$\begin{aligned} E_{G^-} := E_{L^-} \uplus \{(u,v) \in V_{G^-} \times V_{G^-} \mid \\ ((g^-(u), g^-(v)) \in E_G \text{ and } (g^-(u) \notin img(m) \text{ or } g^-(v) \notin img(m))) \\ \text{or } (m^{-1}(g^-(u)), m^{-1}(g^-(v))) \notin E_L\} \end{aligned}$$

The sets of nodes and edges of the object $\bar{L}^-$ given by the final pullback complement $L^- \rightarrowtail l^- \rightarrow \bar{L}^- - \bar{r}^- \rightarrow \bar{L}$ of r^- and l are defined as follows:

$$V_{\bar{L}^-} := V_{L^-} \uplus \{v \in V_{\bar{L}} \mid \nexists w \in V_L : l(w) = v\} = V_{L^-} \uplus V^+.$$

$$\begin{aligned} E_{\bar{L}^-} := E_{L^-} \uplus \{(u,v) \in V_{\bar{L}^-} \times V_{\bar{L}^-} \mid \\ r^-(u) \notin img(l) \text{ or } r^-(v) \notin img(l) \\ \text{or } (l^{-1}(\bar{r}^-(u)), l^{-1}(\bar{r}^-(v))) \notin E_L\}. \end{aligned}$$

Now, we would like to construct the object $\bar{G}$ given by the pushout from $barr^-$ and $\bar{m}^-$. The set of its nodes $V_{\bar{G}}$ is defined as $(V_{\bar{L}} \uplus V_{G^-})_{\approx} = (V_L \uplus V^+ \uplus V_{G^-})_{\approx}$, where

$$\begin{aligned} \approx &= \{(u,v) \in (V_L \uplus V^+) \times V_{G^-} \mid \exists w \in V_{\bar{L}^-} : \bar{r}^-(w) = u \text{ and } \bar{m}^-(w) = v\} \\ &= \{(u,v) \in (V_L \uplus V^+) \times V_{G^-} \mid \exists w \in V_{L^-} : r^-(w) = u \text{ and } m^-(w) = v\} \\ &\quad \cup \{(u,v) \in (V_L \uplus V^+) \times V_{G^-} \mid \exists w \in V^+ : w = u \text{ and } w = v\} \\ &= \{(u,v) \in (V_L \uplus V^+) \times V_{G^-} \mid \exists w \in V_{L^-} : r^-(w) = u \text{ and } m^-(w) = v\} \\ &\quad \cup \{(u,v) \in V^+ \times V_{G^-} \mid \exists w \in V^+ : w = u \text{ and } w = v\}. \end{aligned}$$

The first term of the union in the definition of $\approx$ adds pairs of cloned nodes to the equivalence relation and the second term identifies the same nodes from the set of nodes added to the refined

rule V^+ and V_{G^-}. It is not hard to verify that the set $(V_L \uplus V^+ \uplus V_{G^-})_{\approx}$ is isomorphic to the set of nodes V_G. The set of edges of the pushout object $V_{\bar{G}}$ is defined as follows:

$$\begin{aligned}
E_{\bar{G}} :&= \{(u,v) \in V_G \times V_G \mid \\
&\quad (\exists (u',v') \in E_L \uplus E^+ : \bar{m}(u') = u \text{ and } \bar{m}(v') = v) \\
&\quad\quad \text{or } (\exists (u',v') \in E_{G^-} : g^-(u') = u \text{ and } g^-(v') = v)\} \\
&= \{(u,v) \in V_G \times V_G \mid \exists (u',v') \in E_L : m(u') = u \text{ and } m(v') = v\} \\
&\quad \cup \{(u,v) \in V_G \times V_G \mid \exists (u',v') \in E_+ : \bar{m}(u') = u \text{ and } \bar{m}(v') = v\} \\
&\quad \cup \{(u,v) \in V_G \times V_G \mid \exists (u',v') \in E_{G^-} : g^-(u') = u \text{ and } g^-(v') = v\}
\end{aligned}$$

In the definition of $E_{\bar{G}}$, the first term of the union is isomorphic to the set of edges from G matched by the original left hand-side of the rule. The second term is isomorphic to the set of edges incident to the removed nodes and added to the refined left-hand side. Finally, the third term is isomorphic to the set of edges that stayed preserved as the result of rewriting, i.e. these are edges that where not removed by the rule neither explicitly nor implicitly as a side-effect. Thus, the union of these three sets is isomorphic to the original set of edges of G. □

Proof of Claim 2.1.4. To prove this claim, as before, we will use the definition of pushouts and final pullback complements for simple graphs. The set of nodes of G^+ corresponding to the pushout from m and r^+ is defined as $V_{G^+} := (V_G \uplus V_{L^+})_{\approx}$, where

$$\approx := \{(u,v) \in V_G \times V_{L^+} \mid \exists w \in L : m(w) = u \text{ and } r^+(w) = v\}.$$

The set of its edges E_{G^+} is defined as follows:

$$\begin{aligned}
E_{G^+} := \{(u,v) \in V_{G^+} \times V_{G^+} \mid \\
(\exists (u',v') \in E_G : g^+(u') = u \text{ and } g^+(v') = v) \\
\text{or } (\exists (u',v') \in E_{L^+} : m^+(u') = u \text{ and } m^+(v') = v)\}.
\end{aligned}$$

The set of nodes of the pushout object $\bar{L}^+$ from l and r^+ (i.e. the right-hand side of the refined rule) is defined as $V_{\bar{L}^+} := (V_{\bar{L}} \uplus V_{L^+})_{\approx'}$, where

$$\approx' := \{(u,v) \in V_{\bar{L}} \times V_{L^+} \mid \exists w \in L : l(w) = u \text{ and } r^+(w) = v\};$$

We also observe that $V_{\bar{L}^+} \cong V_{L^+} \uplus V^+$. On the other hand, its edges are given by the following set:

$$\begin{aligned}
E_{\bar{L}^+} := \{(u,v) \in V_{\bar{L}^+} \times V_{\bar{L}^+} \mid \\
(\exists (u',v') \in E_{\bar{L}} : \bar{r}^+(u') = u \text{ and } \bar{r}^+(v') = v) \\
\text{or } (\exists (u',v') \in E_{L^+} : l^+(u') = u \text{ and } l^+(v') = v)\}.
\end{aligned}$$

Now, we would like to construct the object $\bar{G}$ corresponding to the final pullback complement $\bar{r}^+$ and $\bar{m}^+$. The set of its nodes $V_{\bar{G}}$ is defined as

$$\begin{aligned}
V_{\bar{G}} :&= V_{\bar{L}} \uplus \{v \in V_{G^+} \mid \nexists w \in V_{\bar{L}^+} : \bar{m}^+(w) = v\} \\
&= V_L \uplus V^+ \uplus \{v \in V_{G^+} \mid \nexists w \in V_{L^+} \uplus V^+ : \bar{m}^+(w) = v\}.
\end{aligned}$$

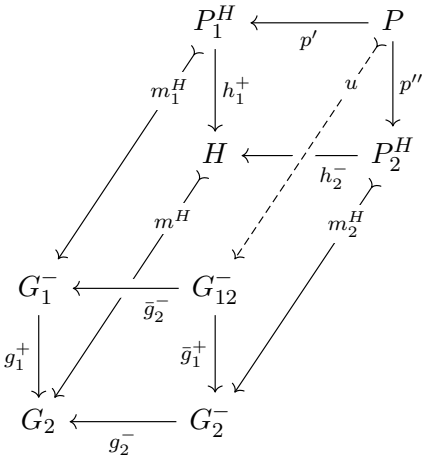
Thus, the set of nodes $V_{\bar{G}}$ is given by the union of three sets: nodes from G matched by the original left-hand side, nodes added to the refined left-hand side, and the rest of the nodes in G. Therefore the set $V_{\bar{G}}$ is isomorphic to the original set of nodes V_G.

The set of edges of the final pullback complement object $\bar{G}$ is defined as follows:

$$\begin{aligned} E_{\bar{G}} := E_{\bar{L}} \cup \{(u,v) \in V_{\bar{G}} \times V_{\bar{G}} \mid \\ ((g^+(u), g^+(v)) \in E_{G^+} \text{ and } (g^+(u) \notin img(\bar{m}^+) \text{ or } g^+(v) \notin img(\bar{m}^+))) \\ \text{or } ((\bar{m}^+)^{-1}(g^+(u)), (\bar{m}^+)^{-1}(g^+(v)))) \notin E_{\bar{L}^+}\}. \end{aligned}$$

The first term of the union is isomorphic to the set of edges from G matched by the refined left-hand side of the rule (i.e. edges matched by the original left-hand side and the edges added as the result of refinement). The second term represents precisely the edges from G that where not matched in $\bar{L}$ (i.e. either their source/target were not matched in $\bar{L}$ or their source *and* target were matched, but no edge between the corresponding nodes was present in $\bar{L}$). Thus, the union of these two sets is isomorphic to he original set of edges of G. □

Proof of Theorem 2.1.5. To prove our theorem let us first construct the pullback $G_1^- \leftarrow \bar{g}_2^- - G_{12}^- - \bar{g}_1^+ \rightarrow G_2^-$ from $G_1^- - g_1^+ \rightarrow G_2 \leftarrow g_2^- - G_2^-$ corresponding to the front face of the cube in Diagram B.1. We observe that the back face of the cube is a pullback by construction. Its bottom face is a pullback by the vertical pasting lemma for pullback complements. Therefore, by the pasting lemma for pullbacks, the composition of the back and bottom squares is also a pullback. The front face being a pullback and the left face a commutative square, allows us to apply Corollary A.3.3 and show that there exists a unique arrow $u : P \to G_{12}^-$ that makes the top and right faces of the cube commute. Moreover, by this corollary, the top face is also a pullback. As the arrow $m_1^H : P_1^H \rightarrowtail G_1^-$ is a mono, the latter observation implies that the arrow u is also a mono (by Lemma A.3.4).



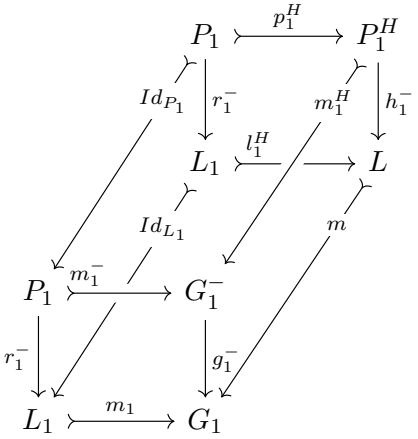
(B.1)

Let us consider Diagram B.2. To prove that applying the rule $L \leftarrow h_1^- \circ p' - P - h_2^+ \circ p'' \rightarrow R$ through the instance $m : L \rightarrowtail G_1$ gives us an object isomorphic to the original object G_3, we need to show that the squares in the following diagram have the following properties: (a) and (b) are final pullback complement squares (i.e. G_1^- is a final pullback complement of h_1^- and m, and G_{12}^- of p' and m_1^H), while (c) and (d) are pushouts. This will allow us to apply the respective pasting lemmas (Lemma A.5.3 for pullback complements (a) and (b) and Lemma A.4.2 for pushouts (c) and (d)).

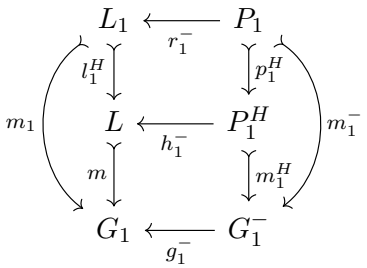
$$\begin{array}{ccccccccc} L & \xleftarrow{h_1^-} & P_1^H & \xleftarrow{p'} & P & \xrightarrow{p''} & P_2^H & \xrightarrow{h_2^+} & R \\ \downarrow{\scriptstyle m} & (a) & \downarrow{\scriptstyle m_1^H} & (b) & \downarrow{\scriptstyle u} & (c) & \downarrow{\scriptstyle m_2^H} & (d) & \downarrow{\scriptstyle m^+} \\ G_1 & \xleftarrow[g_1^-]{} & G_1^- & \xleftarrow[\bar{g}_2^+]{} & G_{12}^- & \xrightarrow[\bar{g}_1^-]{} & G_2^- & \xrightarrow[g_2^+]{} & G_3 \end{array} \tag{B.2}$$

It is easy to verify that the square (b) is a final pullback complement by the vertical pasting lemma (Lemma A.5.4), while the square (d) is a pushout by the pasting lemma for pushouts. Now, to show that the squares (a) and (c) have the desired properties, we need to study the cube in Diagram B.1 further.

To show that (a) is a final pullback complement we will use the adhesivity of our categories from the statement. Consider the cube in Diagram B.3: its front face is a pushout by the reversibility of the rule r_1 and its back face is a pushout by construction. Moreover, its left and top faces are pullbacks by Lemma A.3.6. We know that, by definition, in adhesive categories pushouts along monos are VK-squares. This implies that the right and bottom faces are pullbacks. Let us now consider Diagram B.4. Its outer square is a final pullback complement by construction and its bottom inner square is exactly the right face of the cube from Diagram B.3 and, therefore, is a pullback. Because m is a mono, we can use Lemma A.5.6 and show that both inner squares are final pullback complements. The bottom inner square from Diagram B.4 is exactly the square (a) from B.2.



(B.3)



(B.4)

This implies that the square composed of (a) and (b) is a final pullback complement square and, therefore G_{12}^- constructed as a pullback from g_1^+ and g_2^- is isomorphic to $G_1^{\ominus}$ from the statement of our lemma.

Finally, the proof that square (c) is a pushout consists of a couple of intermediate steps. First of all, we need to prove that the left face in Diagram B.1 is a pushout. Let us recall Diagram 2.18, its outer square is a pushout by construction and we have already shown that square (2) is a pullback. We also observe that arrow $m^H : H \rightarrowtail G_2$ in the diagram is a mono. Recall that, by assumption, we are working in an adhesive category. In such a category, by definition, pushouts are stable under pullbacks (see Definition A.4.6). This allows us to use Lemma A.4.7 and show that both inner squares in this diagram are pushouts. Therefore, the left face of the cube is a pushout square.

We have previously shown that the top, front, bottom and back faces of the cube in the diagram are pullbacks. By the stability of pushouts, because the left face is a pushout, the right face is also a pushout. This is precisely the last ingredient of our proof corresponding to square (c) in Diagram B.2, which concludes the proof of our theorem.

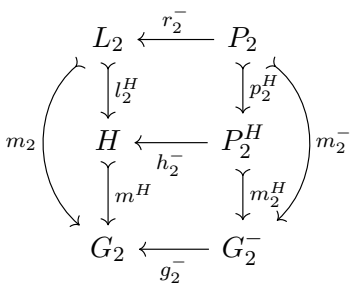
□

Proof of Lemma 2.1.6. To prove this lemma we will use Diagram B.2 from the proof of Theorem

2.1.5. Namely, we will show that the squares (a) and (b) are pushout squares and the squares (c) and (d) are final pullback complement squares (i.e. G_2^- is the final pullback complement of h_2^+ and m^+, and G_{12}^- of p'' and m_2^H).

To show that the square (a) is a pushout we can simply use the inverse of the pasting lemma for pushouts. To prove that the square (b) is a pushout, a couple of intermediate steps is further required.

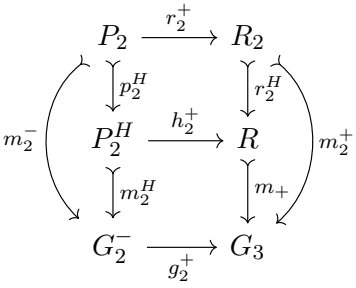
First of all, let us consider Diagram B.5. Its outer square is a final pullback complement by construction ($P_2 \rightarrowtail m_2^- \to G_2^- - g_2^- \to G_2$ is the final pullback complement of r_2^- and m_2). The upper inner square in this diagram is also a final pullback complement, i.e. $P_2 \rightarrowtail p_2^H \to P_2^H - h_2^- \to H$ is the final pullback complement of r_2^- and l_2^H. By the inverse of the vertical pasting lemma, the bottom inner square is also a final pullback complement, and therefore, a pullback square. Recall that the outer square in the diagram is also a pushout by the reversibility of the second rewriting rule (given by r_2^- and r_2^+). This allows us to apply Lemma A.4.7 and show that the bottom square is a pushout. Now, recall the commutative cube from Diagram B.1. Its left, back, right and front faces are pullbacks, and, as we have just shown, its bottom face is a pushout. By the stability of pushouts in the categories of our interest, it implies that the top face is also a pushout (this face corresponds exactly to the square (b) from Diagram B.2).



(B.5)

To show that the square (c) is a final pullback complement, we observe the following. By the vertical pasting lemma, the left face of the cube from Diagram B.1 is a final pullback complement. Moreover, in the proof of Theorem 2.1.5 it has been shown that all the faces of the cube are pullbacks. Therefore, by the stability of pullback complements (Lemma A.5.5), the right face, which is exactly the square (c), is also a final pullback complement.

Finally, to show that the square (d) is a final pullback complement square consider Diagram B.6. Its outer and inner top squares are pushouts by construction. By the pasting lemma for pushouts, its inner bottom square is also a pushout. By Lemma A.6.4, pushouts along monos in adhesive categories are also pullbacks, therefore, the inner bottom square is also a pullback. Because the outer square is a final pullback complement (by the reversibility of the second rewriting) and the arrow m^+ is a mono, the bottom square is also a final pullback complement. This square is exactly the square (d).



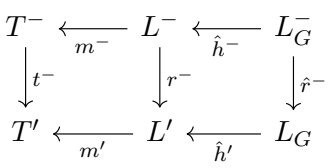
(B.6)

□

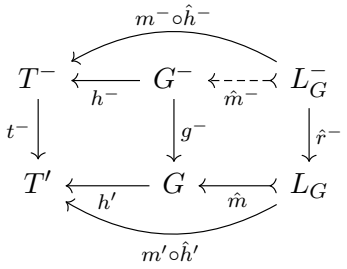
Proof of Theorem 2.2.12. First of all, observe that by the pasting lemma for pullbacks (Lemma A.3.2) $m^- \circ \hat{h}^-$ and $\hat{r}^-$ form the pullback from t^- and $m' \circ \hat{h}'$, i.e. because the two inner squares in Diagram B.7 are pullbacks, the outer square is also a pullback.

Now, observing Diagram B.8 below, we note that h^- and g^- form the pullback by construction. Knowing that the outer square is also a pullback allows us to apply Corollary A.3.3 and show that there exists a unique arrow $\hat{m}^- : L_G^- \to G^-$ that renders the diagram commutative. Moreover, by this corollary, $\hat{m}^-$ and $\hat{r}^-$ give the pullback from g^- and $\hat{m}$. Because pullbacks

preserve monos and $\hat{m}$ is one, $\hat{m}^-$ is a mono as well (by Lemma A.3.4).

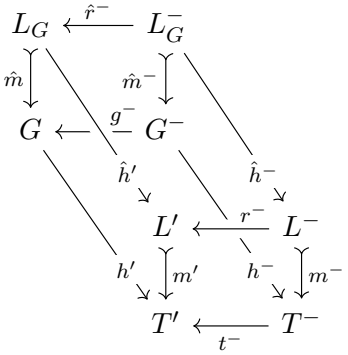


(B.7)

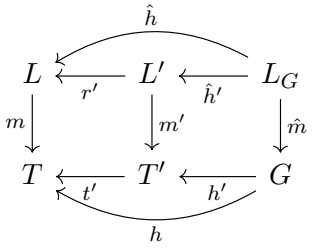


(B.8)

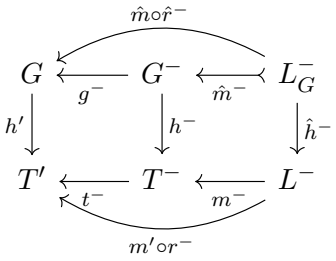
Observe the cube in Diagram B.9. Its front, top and bottom faces are pullbacks by construction. We have also previously shown that its back face is a pullback. Now, if we observe Diagrams B.10 and B.11, we note that their outer squares and left inner squares are pullbacks, by the pasting lemma for pullbacks, the right inner squares are pullbacks as well. These pullbacks form the left and the right faces of the cube in B.9. Moreover, the front face of the cube is a final pullback complement by construction. By stability of pullback complements (Lemma A.5.5), the back face is also a final pullback complement square, i.e. $\hat{m}^-$ and g^- form the final pullback complement of $\hat{r}^-$ and $\hat{m}$, which concludes our proof.



(B.9)



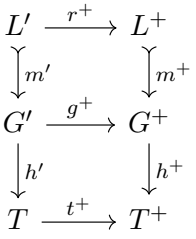
(B.10)



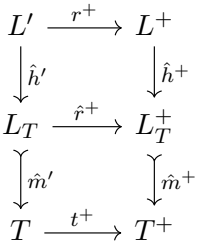
(B.11)

□

Proof of Theorem 2.2.18. The proof of this theorem consists in applying the pasting lemma for pushouts twice. First, we consider Diagram B.12, where two inner squares are pushouts by construction. This implies that the outer square (formed by the composition of the two inner squares) is also a pushout, i.e. t^+ and $\hat{m}^+ \circ \hat{h}^+$ is the pushout from $\hat{m}' \circ \hat{h}'$ and r^+. Now, observing Diagram B.13, we note that its outer square is precisely the previous outer square that we have proven to be a pushout and its upper inner square is a pushout by construction. By the pasting lemma, it implies that the bottom square is also a pushout, which concludes our proof.



(B.12)

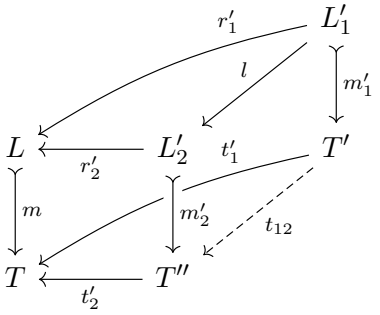


(B.13)

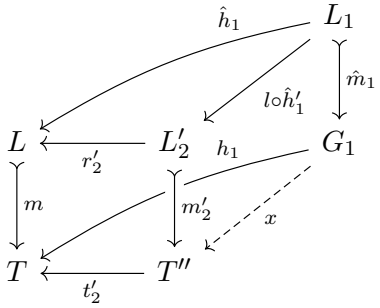
□

Proof of Theorem 2.2.21. Let T' and T'' be the result of the strict phase of rewriting for the factorizations corresponding to L'_1 and L'_2 respectively, constructed as the final pullback complements corresponding to the back and the front faces of Diagram B.14. By the statement of the theorem, $\hat{h}_1 = r'_2 \circ l$, which allows us to use the universal property of the final pullback complement and show that there exists a unique $h'_{12} : T' \to T''$ (that renders the diagram commutative).

By the universal property of the same final pullback complement square and the fact that $r'_1 = r'_2 \circ l$ and $\hat{h}_1 = r'_1 \circ \hat{h}'_1$ (by the statement), there exists a unique $x : G_1 \to T''$ such that (1) $t'_2 \circ x = h_1$ and (2) $x \circ \hat{m}_1 = m'_2 \circ l \circ \hat{h}'_1$, i.e. that renders Diagram B.15 commutative.

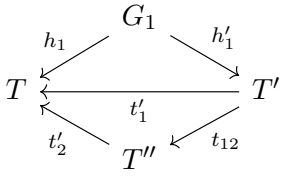


(B.14)

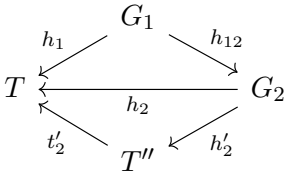


(B.15)

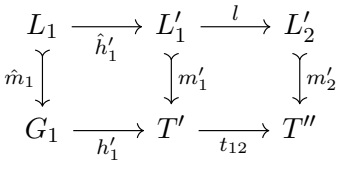
We would like show that the two composed homomorphisms $h'_2 \circ h_{12}$ and $t_{12} \circ h'_1$ satisfy (1) and (2), which will imply that $x = h'_2 \circ h_{12} = t_{12} \circ h'_1$. To do so, let us first observe the four commuting diagrams below. Using the diagram chase we can verify that $t'_2 \circ t_{12} \circ h'_1 = h_1$, $t'_2 \circ h'_2 \circ h_{12} = h_1$, $m'_2 \circ l \circ \hat{h}'_1 = t_{12} \circ h'_1 \circ \hat{m}'_1$ and $m'_2 \circ \hat{h}'_2 \circ \hat{h}_{12} = h'_2 \circ h_{12} \circ \hat{m}_1$, i.e. $t_{12} \circ h'_1$ and $h'_2 \circ h_{12}$ satisfy (1) and (2), which implies that $x = t_{12} \circ h'_1 = h'_2 \circ h_{12}$.



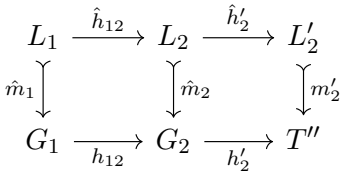
(B.16)



(B.17)

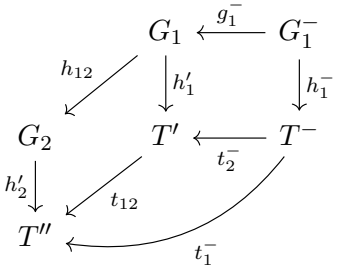


(B.18)

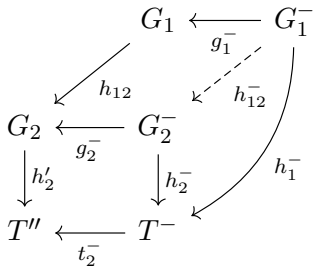


(B.19)

Consider Diagram B.20, all its inner squares commute, which implies that its outer square commutes. This allows us to apply the universal property of the pullback that constructed G_2^-, as in Diagram B.21, and show that there exists a unique arrow h_{12}^- that renders this diagram commutative. This concludes the proof and shows that the two propagations are composable.



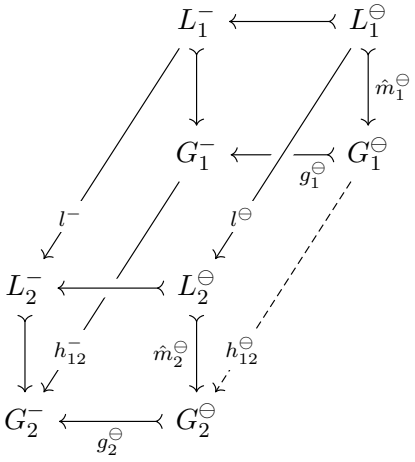
(B.20)



(B.21)

□

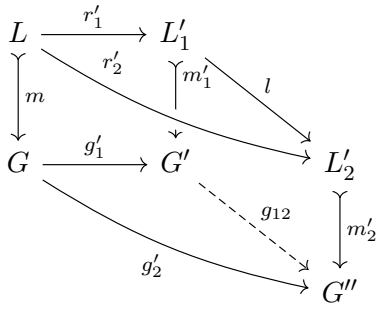
Proof of Theorem 2.2.22. Consider the cube in Diagram B.22. Its back and left faces are pullbacks, therefore, by the pasting lemma, their composition is also a pullback. Moreover, the top face is a commutative square by the statement of the theorem. This allows us to apply the universal property of the final pullback complement that gives $G_2^\ominus$ and show that there exists a unique $h_{12}^\ominus$ that renders the right and the bottom faces of the square commutative, i.e. $h_{12}^\ominus$ is such that $\hat{m}_2^\ominus \circ l^\ominus = h_{12}^\ominus \circ \hat{m}_1^\ominus$ and $g_2^\ominus \circ h_{12}^\ominus = h_{12}^- \circ g_1^\ominus$. We also know that $g_2^\ominus$ is a mono, it implies that for any homomorphism $z : G_1^\ominus \to G_2^\ominus$ such that $g_2^\ominus \circ z = h_{12}^- \circ g_1^\ominus = g_2^\ominus \circ h_{12}^\ominus$, it holds that $z = h_{12}^\ominus$. Therefore, $h_{12}^\ominus$ is the unique homomorphism that renders the bottom face of the cube commutative.



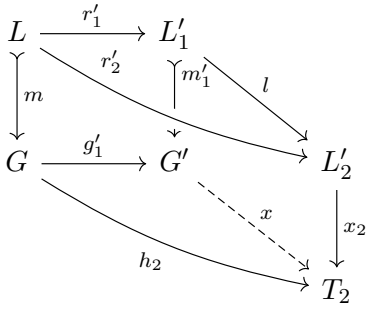
(B.22)

□

Proof of Theorem 2.2.25. Let G' and G'' be the result of the strict phase of rewriting given the factorizations L_1' and L_2' respectively, constructed as the pushouts corresponding to the back and the front phases of Diagram B.23. We know by the statement of the theorem that $l \circ r_1' = r_2'$. This allows us to state that the outer square in the diagram commutes. Therefore, we can apply the universal property of the pushout that constructed G' and show that there exists a unique homomorphism $g_{12} : G' \to G''$ that renders our diagram commutative. Similarly, by the statement, $h_2 \circ m = x_2 \circ r_2'$ and $r_2' = l \circ r_1'$, which implies that the outer square in Diagram B.24 commutes. This allows us to apply the same universal property of G' and show that there exists a unique homomorphism $x : G' \to T_2$ that renders the following diagram commutative, i.e. that satisfies (1) $h_2 = x \circ g_1'$ and (2) $x \circ m_1' = x_2 \circ l$.

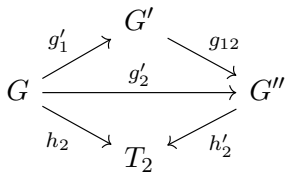


(B.23)

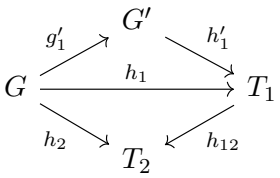


(B.24)

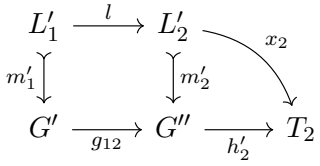
We would like to show that the two composed homomorphisms $h'_2 \circ g_{12}$ and $h_{12} \circ h'_1$ satisfy (1) and (2), which will imply that $x = h'_2 \circ g_{12} = h_{12} \circ h'_1$. First of all, observe that all four diagrams below commute. Using the diagram chase we can verify that $h'_2 \circ g_{12} \circ g'_1 = h_2$, $h_2 = h_{12} \circ h'_1 \circ g'_1$, $h'_2 \circ g_{12} \circ m'_1 = x_2 \circ l$ and $h_{12} \circ h'_1 \circ m'_1 = x_2 \circ l$, i.e. $h'_2 \circ g_{12}$ and $h_{12} \circ h'_1$ satisfy (1) and (2), which implies that $x = h'_2 \circ g_{12} = h_{12} \circ h'_1$.



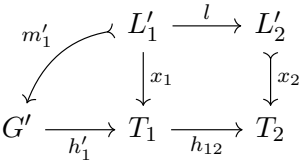
(B.25)



(B.26)

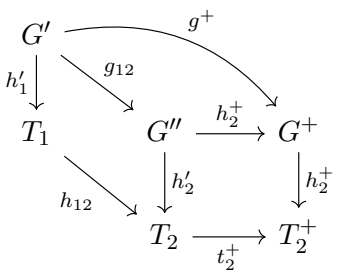


(B.27)

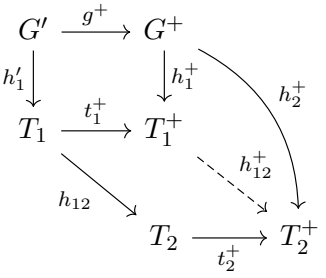


(B.28)

Consider Diagram B.29, all its inner squares commute, which implies that its outer square commutes. This allows us to apply the universal property of the pushout that constructed T_1^+, as in Diagram B.30, and show that there exists a unique arrow h_{12}^+ that renders this diagram commutative. This concludes our proof and shows that the two propagations are composable.



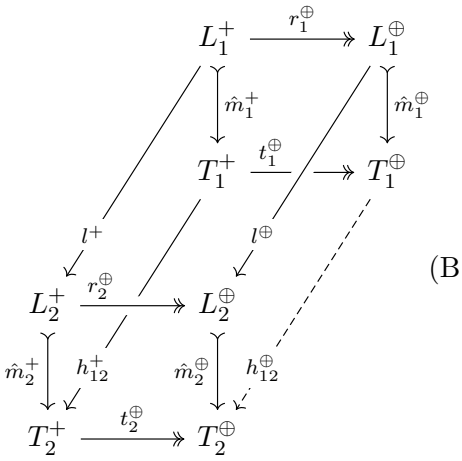
(B.29)



(B.30)

□

Proof of Theorem 2.2.26. Consider the cube in Diagram B.31. Its back, left and top faces are commutative squares, which allows us to apply the universal property of the pushout that gives $T_1^{\oplus}$ and show that there exists a unique $h_{12}^{\oplus} : T_1^{\oplus} \to T_2^{\oplus}$ that renders all the right and the bottom faces of the square commutative, i.e. $h^{\oplus}$ is such that $h_{12}^{\oplus} \circ t_1^{\oplus} = t_2^{\oplus} \circ h_{12}^{+}$ and $h_{12}^{\oplus} \circ \hat{m}_1^{\oplus} = \hat{m}_2^{\oplus} \circ l^{\oplus}$. We also know that $t_1^{\oplus}$ is an epi, it implies that for any homomorphism $z : T_1^{\oplus} \to T_2^{\oplus}$ such that $z \circ t_1^{\oplus} = t_2^{\oplus} \circ h_{12}^{+} = h_{12}^{\oplus} \circ t_1^{\oplus}$ it holds that $z = h_{12}^{\oplus}$. Therefore, $h_{12}^{\oplus}$ is the unique homomorphism that renders the bottom face of the cube commutative.



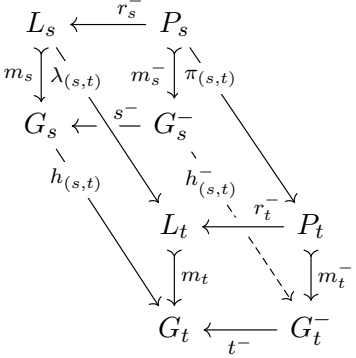
(B.31)

□

Proof of Proposition 2.2.33. Consider Diagram B.32 representing the result of the first phase of rewriting of two objects corresponding to the skeleton nodes $s, t \in V$ such that $(s, t) \in E$. We need to show that there exists a unique arrow $h_{(s,t)}^{-}$ rendering the cube in the diagram commutative.

Recall that, by construction of our rule hierarchy, if the object G_t was not affected by backwards propagation, the rule r_t^- is an identity rule. If r_t^- is an identity arrow, i.e. $r_t^- = Id_{L_t}$, then t^- is also an identity, i.e. $t^- = Id_{G_t}$, and, therefore, the object G_t stays unchanged as the result of rewriting. It implies that the desired homomorphism is precisely $h_{(s,t)} \circ s^-$.

On the other hand, if r_t^- is not an identity, i.e. we have performed a restrictive rewrite (for example, backward propagation) to G_t, then we have performed backward propagation to G_s. This implies that, by construction, the left face of the cube in Diagram B.32 is a pullback. Therefore, we can apply the universal property of the final pullback complement that constructed G_t^- and find the unique homomorphism $h_{(s,t)}^{-} : G_s^- \to G_t^-$ that renders our diagram commutative.

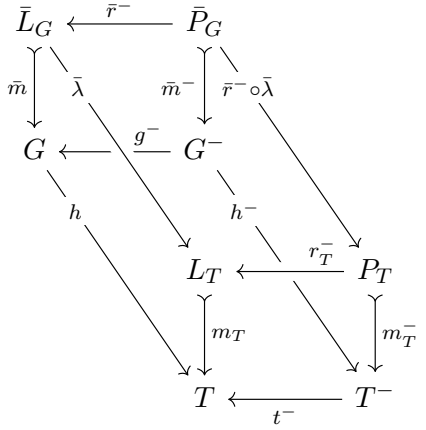


(B.32)

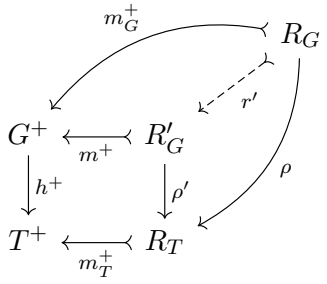
□

Proof of Proposition 2.2.36. Let us start our proof by showing that $G^- \leftarrow \bar{m}^- \prec \bar{P}_G - \bar{r} \circ \bar{\lambda} \rightarrow P_T$ is the pullback from h^- and m_T^-. Consider the cube in Diagram B.33. Its left, back and front faces are pullbacks by construction, while its top and bottom faces are commuting squares. By the inverse of the pasting lemma, its right face is also a pullback.

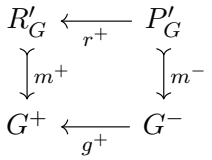
Now, let us construct the pullback $G^+ \leftarrow m^+ \prec R'_G - \rho' \rightarrow R_T$ from h^+ and m_T^+. By the universal property of this pullback, there exists a unique arrow $r' : R_G \to R'_G$ that makes Diagram B.34 commute. Moreover, it is not hard to verify that this arrow is a mono (as the top triangle is a pullback and pullbacks preserve monos). Let $R'_G \leftarrow r^+ - P'_G \succ m^- \rightarrow$ be a pullback from m^+ and g^+ as in Diagram B.35.



(B.33)



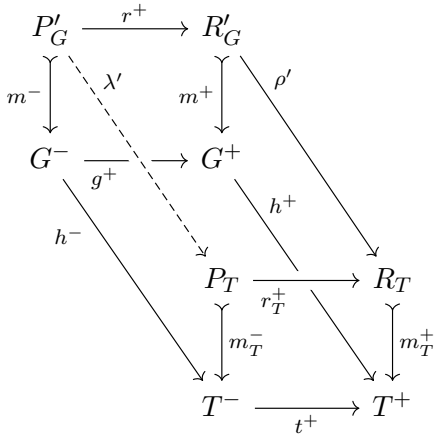
(B.34)



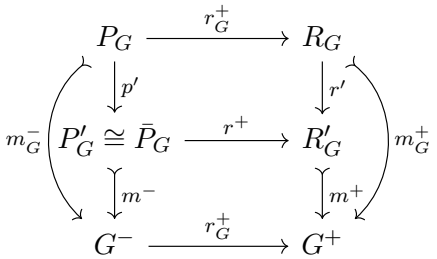
(B.35)

Observe the cube in Diagram B.36. Because, by assumption, our rules are reversible, its front face is a pullback, and there exists a unique arrow $\lambda' : P'_G \to P_T$ that makes our cube commute. By construction, the back face is also a pullback and previously we have shown that the right face is a pullback. Because all the faces of the cube commute, by the inverse of the pasting lemma, the left face is also a pullback. This implies that $P'_G \cong \bar{P}_G$.

Consider Diagram B.37, we know that its outer square is a pullback (because the rule is reversible), while the bottom square is a pullback by construction. We also know that the outer square is a pushout by construction and that m^+ is a mono. This allows us to apply Lemma A.4.7 and show that the two inner squares are also pushouts. Therefore, $R'_G \cong \bar{R}_G$, which concludes our proof.



(B.36)



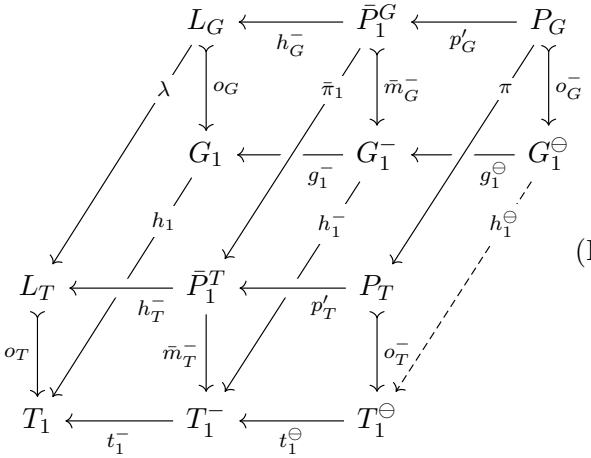
(B.37)

□

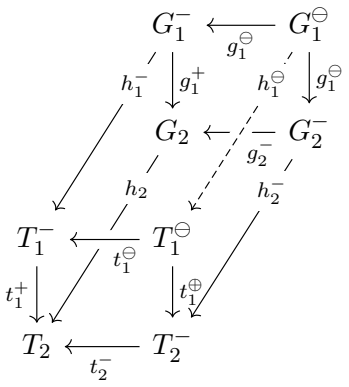
Proof of Theorem 2.2.37. Consider Diagram B.38, in the proof of Theorem 2.1.5 we have shown that G_1^- is also the final pullback complement of h_G^- and o_G. In a similar way, T_1^- can be obtained as the final pullback complement of h_T^- and o_T. Recall that, because the rule hierarchy $\mathcal{R}_1$ is applicable given m_G and m_T, there exists a unique homomorphism $h_1^- : G_1^- \to T_1^-$ that renders the diagram commutative. Let $G_1^{\ominus}$ and $T_1^{\ominus}$ in Diagram B.38 be constructed as the final pullback

complements of the pair p'_G and $\bar{m}^-_G$ and the pair p'_T and $\bar{m}^-_T$ respectively. By the horizontal pasting lemma $G^\ominus_1$ and $T^\ominus_1$ are also the final pullback complements of the pair $h^-_G \circ p'_G$ and o_G and the pair $h^-_T \circ p'_T$ and o_T (i.e. we can obtain $G^\ominus_1$ and $T^\ominus_1$ by applying the restrictive rules given by $h^-_G \circ p'_G$ and $h^-_T \circ p'_T$ through the respective instances o_G and o_T). We need to show that there exists a unique $h^\ominus_1 : G^\ominus_1 \to T^\ominus_1$ that renders this diagram commutative.

In the proof of Theorem 2.1.5 we have shown that the objects $G^\ominus_1$ and $T^\ominus_1$ correspond to the pullbacks from $G^-_1 - g^+_1 \to G_2 \leftarrow g^-_2 - G^-_2$ and $T^-_1 - t^+_1 \to T_2 \leftarrow t^-_2 - T^-_2$ respectively (see the front and back faces of Diagram B.39). This fact allows us to use the universal property of pullbacks and show that there exists a unique $h^\ominus_1$ that renders the cube in Diagram B.39 commutative.

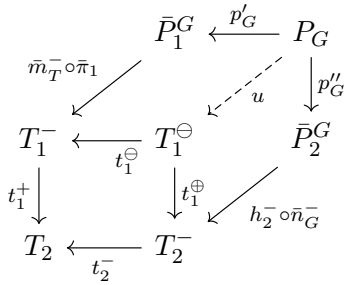


(B.38)



(B.39)

First of all, let us prove that such homomorphism $h^\ominus_1$ makes the right face of the cube from Diagram B.38 commute. To do so, we will use the universal property of pullbacks in the following way. Let $\bar{n}^-_G : \bar{P}^G_2 \rightarrowtail G^-_2$ and $\bar{n}^-_T : \bar{P}^T_2 \rightarrowtail T^-_2$ be two homomorphisms given by the universal property of the final pullback complements that constructed G^-_2 and T^-_2 (see the construction presented in Diagram 2.17). We will show that there exists a unique $u : P_G \to T^\ominus_1$ such that (a) $t^\ominus_1 \circ u = \bar{m}^-_T \circ \bar{\pi}_1 \circ p'_G$ and (b) $t^\oplus_1 \circ u = h^-_2 \circ \bar{n}^-_G \circ p''_G$, i.e. that renders Diagram B.40 commutative. It is easy to verify that both $o^-_T \circ \pi$ and $h^\ominus_1 \circ o^-_G$ satisfy (a) and (b). This implies that $o^-_T \circ \pi = h^\ominus_1 \circ o^-_G$.



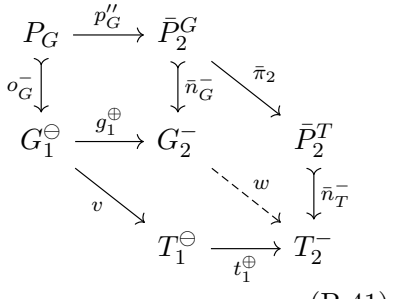
(B.40)

Finally, to prove that $h^\ominus_1$ is indeed the unique homomorphism that renders the right face of the cube from Diagram B.38 commutative, we assume that there exists another such homomorphism $v : G^\ominus_1 \to T^\ominus_1$ (and $v \neq h^\ominus_1$). Then, we apply the universal property of pushouts and show that given such v, there exists a unique $w : G^-_2 \to T^-_2$ that makes Diagram B.41 commute. Clearly, $w = h^-_2$ and therefore $t^\ominus_1 \circ v = h^-_2 \circ g^\oplus_1$, which implies that $v = h^\ominus_1$ (by the universal property in Diagram B.39).

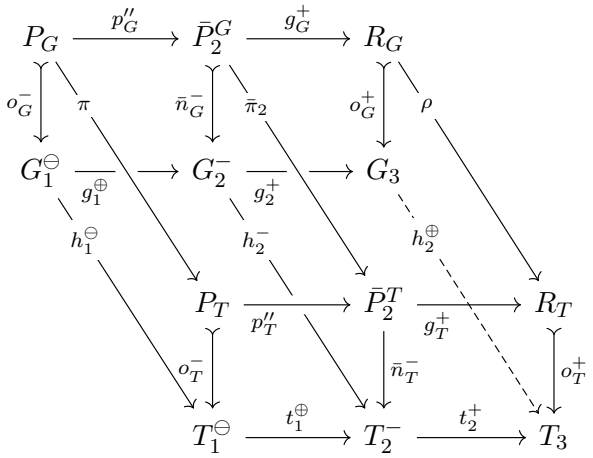
Consider Diagram B.42, in the proof of Theorem 2.1.5 we have shown that G^-_2 is also the pushout from p''_G and o^-_G. Similarly, T^-_1 can be obtained as the pushout from p''_T and o^-_T. Recall that, because the homomorphic pair of rules q_G and q_T is applicable, there exists a unique homomorphism $h^-_2 : G^-_2 \to T^-_2$ that renders the diagram commutative. In the proof of Theorem 2.1.5 we have also seen that G_3 and T_3 be constructed as the pushouts from the pair

g_G^+ and $\bar{n}_G^-$ and the pair g_T^+ and $\bar{n}_T^-$ respectively. By the pasting lemma for pushouts G_3 and T_3 are also the pushouts from the pair $g_G^+ \circ p''_G$ and o_G^- and the pair $g_T^+ \circ p''_T$ and o_T^- (i.e. they can be obtained by applying the expansive rules given by $g_G^+ \circ p''_G$ and $g_T^+ \circ p''_T$ through the instances o_G^- and o_T^-). We need to show that there exists a unique $h_2^\oplus : G_3 \to T_3$ that renders this diagram commutative and that $h_2^\oplus = h_3$ from Diagram 2.119.

First of all, by the universal property of pushouts, there exists a unique arrow $h_2^\oplus$ that renders the right-most cube in Diagram B.42 commutative. It can be easily verified that $h_2^\oplus$ satisfies (a) $g_2^+ \circ h_2^\oplus = t_2^+ \circ h_2^-$ and (b) $n_G^+ \circ h_2^\oplus = n_T^+ \circ \rho_2$. This implies that $h_2^\oplus = h_3$ by the universal property of pushouts that guarantees the existence and the uniqueness of h_3.



(B.41)



(B.42)

□

Appendix C

Algorithms

C.1 Propagation algorithms

The following pseudocode illustrates the algorithm of rewriting and propagation in hierarchies. Note that it assumes that the input rule factorizations and clean-up arrows satisfy consistency conditions given in Theorem 2.2.21 and 2.2.25. The algorithm uses subroutines `propagate_backward`, `propagate_forward` and `restore_edges` given in Algorithm 2, 3 and 4 respectively.

Algorithm 1: Hierarchy rewriting algorithm

Data: $H = (V, E, \mathcal{O}, \mathcal{H}, \alpha, \beta)$, $v \in V$, a rule $r = L \overset{r^-}{\leftarrow} P \overset{r^+}{\rightarrow} R$, a match $m : L \rightarrowtail \alpha(v)$, sets $\mathcal{B}$ and $\mathcal{F}$ defining composable rule factorizations and clean-up arrows for ancestors and descendants of v respectively, a function f associating nodes of the hierarchy with elements from $\mathcal{B} \cup \mathcal{F}$.

Result: $H' = (V, E, \mathcal{O}', \mathcal{H}', \alpha', \beta')$, $m^+ : R \rightarrowtail \alpha'(v)$.

$\mathcal{O}' := \emptyset$; $\mathcal{H}' := \emptyset$;
`// set the origin of rewriting`
$G := \alpha(v)$;

`// rewrite` G `and obtain` $G \overset{g^-}{\leftarrow} G^- \overset{g^+}{\rightarrow} G^+$ `and` $m^+ : R \rightarrow G^+$
$G^-, G^+, g^-, g^+, m^+ :=$ rewrite $\alpha(v)$ with r through the match m;
`// backward propagation to all the ancestors of` v
$\mathcal{O}_{anc}, \mathcal{H}^A, \mathcal{H}^O_{anc}, a, o_{anc} :=$ `propagate_backward(`$H, v, G^-, g^-, \mathcal{B}, \alpha'$`)`;
`// forward propagation to all the descendants of` v
$\mathcal{O}_{desc}, \mathcal{H}^D, \mathcal{H}^O_{desc}, d, o_{desc} :=$ `propagate_forward(`
$H, v, G^-, G^+, g^-, g^+, \mathcal{F}, \alpha'$`)`;
$\mathcal{O}' := \{G^+\} \cup \mathcal{O}_{anc} \cup \mathcal{O}_{desc}$; $\alpha'(v) = G^+$;
`// restore edges preserving the consistency condition`
$\mathcal{H}', \beta' :=$ `restore_edges(`$H, v, g^+, a, o_{anc}, d, o_{desc}$`)`;
$H' = (V, E, \mathcal{O}', \mathcal{H}', \alpha', \beta')$

Algorithm 2: `propagate_backward`

Data: $H, v, T^-, t^-, \mathcal{B}, f, \alpha'$

Result: a set $\mathcal{O}_{anc}$ of the updated ancestors, a set $\mathcal{H}^A$ of homomorphisms from the updated ancestors to the original ancestors, a set $\mathcal{H}^O$ of homomorphisms from the updated ancestors to the updated origin, respective functions a and o that map nodes of the hierarchy to the homomorphisms from $\mathcal{H}^A$ and $\mathcal{H}^O$

```
// set T to be the origin of rewriting
```
$T := \alpha(v)$;
$\mathcal{H}^A := \emptyset$;
$\mathcal{H}^O = \emptyset$;
$\mathcal{O}_{anc} := \emptyset$;
```
// breadth-first traversal with backward propagation
```
$visited := \emptyset$;
$current\_predecessors :=$ predecessors of v in H;
while $current\_predecessors \neq \emptyset$ **do**
  $next\_predecessors := \emptyset$;
  for $p \in current\_predecessors$ **do**
    if $p \notin visited$ **then**
      $visited := visited \cup \{p\}$;
      $G := \alpha(p)$;
      $h :=$ composition of homomorphisms along $path(p, v)$; `// find` G^-`,` $g^- : G^- \to G$ `and` $h^- : G^- \to T^-$
      $G^-, g^-, h^- :=$ propagate using T, T^-, t^- and the data from $f(p)$;
      $\mathcal{H}^A := \mathcal{H}^A \cup \{g^-\}$;
      $a(p) := g^-$;
      $\mathcal{H}^O := \mathcal{H}^O \cup \{h^-\}$;
      $o(p) := h^-$;
      $\mathcal{O}_{anc} := \mathcal{O}_{anc} \cup \{G^-\}$;
      $\alpha'(p) := G^-$;
      $next\_predecessors := next\_predecessors \cup \{p' \in$ predecessors of p in $H \mid p' \notin visited\}$;
    end
  end
  $current\_predecessors := next\_predecessors$;
end

Algorithm 3: `propagate_forward`

Data: $H, v, G^-, G^+, g^-, g^+, \mathcal{F}, f, \alpha'$

Result: a set $\mathcal{O}_{desc}$ of the updated descendants, a set $\mathcal{H}^D$ of homomorphisms from the original descendants to the updated descendants, a set $\mathcal{H}^O$ of homomorphisms from the updated origin to the updated descendants, respective functions d and o that map nodes of the hierarchy to the homomorphisms from and $\mathcal{H}^D$ and $\mathcal{H}^O$

```
// set G to be the origin of rewriting
G := α(v);
O_desc := ∅;
H^D := ∅;
H^O = ∅;
// breadth-first traversal with forward propagation
visited := ∅;
current_successors := successors of v in H;
while current_successors ≠ ∅ do
    next_successors := ∅;
    for s ∈ current_successors do
        if s ∉ visited then
            visited := visited ∪ {s};
            T := α(s);
            h := composition of homomorphisms along path(v, s); // find T⁺,
                 t⁺ : T → T⁺ and h⁺ : G⁺ → T⁺
            T⁺, t⁺, h⁺ := propagate using G, G⁻, G⁺, g⁻, g⁺ and
                                the data from f(s);
            H^D := H^D ∪ {t⁺};
            d(s) := t⁺;
            H^O := H^O ∪ {h⁺};
            o(s) := h⁺;
            O_desc := O_desc ∪ {T⁺};
            α'(s) := T⁺;
            next_successors := next_successors ∪ {s' ∈
                 successors of s in H | s' ∉ visited};
        end
    end
    current_successors := next_successors;
end
```

Algorithm 4: `restore_edges`

Data: $H, v, g^+, a, o_{anc}, d, o_{desc}$
Result: a set of homomorphisms $\mathcal{H}'$ and a function β' that associates homomorphisms from $\mathcal{H}'$ with the edges of the hierarchy

for $(s,t) \in$ *the set of edges of* H **do**
  if $s \in$ *ancestors of* v **then**
    if $t \in$ *ancestors of* v **then**
      $h_{s,t}$:= find the homomorphism from the backward composability theorems (2.2.21 and 2.2.22), uniquely determined by $a(s), a(t), \beta(s,t), o_{anc}(s)$ and $o_{anc}(t)$;
    else if $t = v$ **then**
      $h_{s,t} := g^+ \circ o_{anc}(s)$;
    else if $t \in$ *descendants of* v **then**
      $h_{s,t} := o_{desc}(t) \circ g^+ \circ o_{anc}(s)$;
    else
      $h_{s,t} := \beta(s,t) \circ a(s)$;
    end
  else if $s = v$ **then**
    $h_{s,t} := o_{desc}(t)$;
  else if $s \in$ *descendants of* v **then**
    if $t \in$ *descendants of* v **then**
      $h_{s,t}$:= find the homomorphism from the forward composability theorems (2.2.25 and 2.2.26), uniquely determined by $d(s), d(t), \beta(s,t), o_{desc}(s)$ and $o_{desc}(t)$;
  else
    if $t \in$ *descendants of* v **then**
      $h_{s,t} := d(t) \circ \beta(s,t)$;
    else
      $h_{s,t} := \beta(s,t)$;
    end
  end
  $\mathcal{H}' := \mathcal{H}' \cup \{h_{s,t}\}$;
  $\beta'(s,t) := h_{s,t}$;
end

Appendix D

Cypher queries

D.1 Example clone query

```
// Query performing clone of a node
MATCH (a { id : 'a' })
// create a node corresponding to the clone
CREATE (a1)
WITH a, a1
SET a1 = a
WITH a, a1

// match successors and out-edges
OPTIONAL MATCH (a)-[out_edge:edge]->(suc)
WITH a, a1, filter(
  el IN collect(
    {neighbor: suc, edge: out_edge})
  WHERE NOT el.neighbor IS NULL) as suc_maps
// match predecessors and in-edges
OPTIONAL MATCH (pred)-[in_edge:edge]->(a)
WITH a, a1, suc_maps, filter(
  el IN collect(
    {neighbor: pred, edge: in_edge})
  WHERE NOT el.neighbor IS NULL) as pred_maps

// copy all incident edges of the original node
FOREACH (suc_map IN suc_maps |
  FOREACH(suc IN [suc_map.neighbor] |
    CREATE (a1)-[new_edge:edge]->(suc)
    SET new_edge = suc_map.edge))
FOREACH (pred_map IN pred_maps |
  FOREACH(pred in [pred_map.neighbor] |
    CREATE (pred)-[new_edge:edge]->(a1)
    SET new_edge = pred_map.edge))

// copy self loop
FOREACH (suc_map IN suc_maps |
  FOREACH (self_loop IN
```

```
    CASE WHEN suc_map.neighbor=a
    THEN [suc_map.edge] ELSE [] END |
      CREATE (a1)-[new_edge:edge]->(a1)
      SET new_edge = self_loop))
WITH a, a1
RETURN a1
```

D.2 Example merge query

```
// As the following is not allowed by Cypher
// `SET a[key] = b[key]`, so we use APOC instead:
// `SET a = apoc.map.setKey(a, key, b[key])`
MATCH (a { id : 'a'}), (b { id : 'b'})

// Add properties of 'b' to 'a'
FOREACH(key in keys(b) |
  FOREACH(dummy IN
    CASE WHEN key IN keys(a)
    THEN [] ELSE [NULL] END |
      // SET a[key] = b[key]
      SET a = apoc.map.setKey(
        a, key, b[key]))
  FOREACH(dummy IN
    CASE WHEN key IN keys(a)
    THEN [NULL] ELSE [] END |
      SET a = apoc.map.setKey(
        a, key, a[key] + filter(
          el IN b[key] WHERE NOT el in a[key]))))
// list with ids of merged nodes to track self loops
WITH a as merged_node, b,
  [id(a), id(b)] as merged_nodes

// match successors of 'b'
OPTIONAL MATCH (b)-[out_edge:edge]->(suc)
WITH merged_node, b, merged_nodes, filter(
  el IN collect({neighbor: suc, edge: out_edge})
  WHERE NOT el.neighbor IS NULL) AS all_suc_maps
WITH merged_node, b, merged_nodes,
  filter(
    el in all_suc_maps
    WHERE NOT id(el.neighbor) IN merged_nodes)
      AS new_suc_maps,
  filter(
    el in all_suc_maps
    WHERE id(el.neighbor) IN merged_nodes)
      AS loop_suc_maps

//  match predecessors of 'b'
OPTIONAL MATCH (pred)-[in_edge:edge]->(b)
WITH merged_node, b, merged_nodes, new_suc_maps,
  loop_suc_maps, filter(
```

```
  el IN collect({neighbor: pred, edge: in_edge})
  WHERE NOT el.neighbor IS NULL) AS all_pred_maps
WITH merged_node, b, merged_nodes, new_suc_maps,
  loop_suc_maps, filter(
    el IN all_pred_maps
    WHERE NOT id(el.neighbor) IN merged_nodes)
      AS new_pred_maps,
  filter(
    el IN all_pred_maps
    WHERE id(el.neighbor) IN merged_nodes)
      AS loop_pred_maps

// create edges for sucs/preds that
// didn't exist before and/or merge
// their attributes into existing edges
FOREACH (suc_map IN new_suc_maps |
  FOREACH(suc IN [suc_map.neighbor] |
    MERGE (merged_node)-[edge:edge]->(suc)
    // Merge dicts
    FOREACH(key in keys(suc_map.edge) |
      FOREACH(dummy IN
        CASE WHEN key IN keys(edge)
        THEN [] ELSE [NULL] END |
        // SET edge[key] = suc_map.edge[key]
        SET edge = apoc.map.setKey(
          edge, key, suc_map.edge[key])
      )
      FOREACH(dummy IN
        CASE WHEN key IN keys(edge)
        THEN [NULL] ELSE [] END |
          SET edge = apoc.map.setKey(
            edge, key, edge[key] + filter(
              el IN suc_map.edge[key]
              WHERE NOT el in edge[key])))))

FOREACH (pred_map IN new_pred_maps |
  FOREACH(pred in [pred_map.neighbor] |
    MERGE (pred)-[edge:edge]->(merged_node)
    FOREACH(key in keys(pred_map.edge) |
      FOREACH(dummy IN
        CASE WHEN key IN keys(edge)
        THEN []
        ELSE [NULL] END |
          SET edge = apoc.map.setKey(
            edge, key, pred_map.edge[key])
      )
      FOREACH(dummy IN
        CASE WHEN key IN keys(edge)
        THEN [NULL] ELSE [] END |
        SET edge = apoc.map.setKey(
          edge, key, edge[key] + filter(
            el IN pred_map.edge[key]
            WHERE NOT el in edge[key])))))
```

```
// handle self loops
WITH merged_node, b, loop_suc_maps, loop_pred_maps
OPTIONAL MATCH (merged_node)-[old_loop:edge]->(merged_node)
FOREACH(dummy IN
  CASE WHEN NOT old_loop IS NULL OR
    length(loop_suc_maps) > 0 OR
    length(loop_pred_maps) > 0
  THEN [NULL] ELSE [] END |
  MERGE (merged_node)-[
      old_loop:edge]->(merged_node)
  FOREACH (map IN loop_suc_maps + loop_pred_maps |
    FOREACH(key in keys(map.edge) |
      FOREACH(dummy IN
        CASE WHEN key IN keys(old_loop)
        THEN [] ELSE [NULL] END |
        SET old_loop = apoc.map.setKey(
          old_loop, key, map.edge[key])
      )
      FOREACH(dummy IN
        CASE WHEN key IN keys(old_loop)
        THEN [NULL] ELSE [] END |
        SET old_loop = apoc.map.setKey(
          old_loop, key, old_loop[key] + filter(
            el IN map.edge[key]
            WHERE NOT el in old_loop[key])))))
DETACH DELETE b
RETURN merged_node
```

Index

Bibliography

[1] R. Aebersold and M. Mann. Mass-spectrometric exploration of proteome structure and function. *Nature*, 537(7620):347, 2016.

[2] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. Molecular biology of the cell 4th edn (new york: Garland science). *Ann Bot*, 91:401, 2002.

[3] J. Allen, W. de Beaumont, L. Galescu, and C. M. Teng. Complex event extraction using drum. Technical report, Florida Institute for Human and Machine Cognition Pensacola United States, 2015.

[4] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):68, 2017.

[5] S. Awodey. *Category theory*. Oxford University Press, 2010.

[6] F. Baader, D. Calvanese, D. McGuinness, P. Patel-Schneider, and D. Nardi. *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[7] N. Behr. Sesqui-pushout rewriting: Concurrency, associativity and rule algebra framework. *arXiv preprint arXiv:1904.08357*, 2019.

[8] P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM, 2007.

[9] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. Querying graphs. *Synthesis Lectures on Data Management*, 10(3):1–184, 2018.

[10] A. Bonifati, P. Furniss, A. Green, R. Harmer, E. Oshurko, and H. Voigt. Schema validation and evolution for graph databases. *arXiv preprint arXiv:1902.06427*, 2019.

[11] A. Bonifati, P. Furniss, A. Green, R. Harmer, E. Oshurko, and H. Voigt. Schema validation and evolution for graph databases. In *International Conference on Conceptual Modeling*, pages 448–456. Springer, 2019.

[12] P. Boutillier, F. Camporesi, J. Coquet, J. Feret, K. Q. Lỳ, N. Theret, and P. Vignet. Kasa: a static analyzer for kappa. In *International Conference on Computational Methods in Systems Biology*, pages 285–291. Springer, 2018.

[13] P. Boutillier, T. Ehrhard, and J. Krivine. Incremental update for graph rewriting. In *European Symposium on Programming*, pages 201–228. Springer, 2017.

[14] P. Boutillier, M. Maasha, X. Li, H. F. Medina-Abarca, J. Krivine, J. Feret, I. Cristescu, A. G. Forbes, and W. Fontana. The kappa platform for rule-based modeling. *Bioinformatics*, 34(13):i583–i592, 2018.

[15] M. H. Burstein. R3e: Reading, reasoning and reporting explanations. Technical report, Smart Information Flow Technologies (SIFT) Minneapolis United States, 2018.

[16] P. P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[17] P. R. Cohen. Darpa's big mechanism program. *Physical biology*, 12(4):045008, 2015.

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

[19] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The category of typed graph grammars and its adjunctions with categories of derivations. In *International Workshop on Graph Grammars and Their Application to Computer Science*, pages 56–74. Springer, 1994.

[20] A. Corradini, T. Heindel, F. Hermann, and B. König. Sesqui-pushout rewriting. In *International Conference on Graph Transformation*, pages 30–45. Springer, 2006.

[21] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. 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.

[22] I. Cristescu, W. Fontana, and J. Krivine. Interactions between causal structures in graph rewriting systems. *arXiv preprint arXiv:1901.00592*, 2019.

[23] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the prism workbench. *Proceedings of the VLDB Endowment*, 1(1):761–772, 2008.

[24] T. Czauderna, C. Klukas, and F. Schreiber. Editing, validating and translating of sbgn maps. *Bioinformatics*, 26(18):2340–2341, 2010.

[25] V. Danos, J. Feret, W. Fontana, R. Harmer, J. Hayman, J. Krivine, C. Thompson-Walsh, and G. Winskel. Graphs, rewriting and pathway reconstruction for rule-based models. In *FSTTCS 2012-IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 18, pages 276–288, 2012.

[26] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling of cellular signalling. In *International conference on concurrency theory*, pages 17–41. Springer, 2007.

[27] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling and model perturbation. In *Transactions on Computational Systems Biology XI*, pages 116–137. Springer, 2009.

[28] V. Danos, J. Feret, W. Fontana, and J. Krivine. Scalable simulation of cellular signaling networks. In *Asian Symposium on Programming Languages and Systems*, pages 139–157. Springer, 2007.

[29] V. Danos, T. Heindel, R. Honorato-Zimmer, and S. Stucki. Reversible sesqui-pushout rewriting. In *International Conference on Graph Transformation*, pages 161–176. Springer, 2014.

[30] R. Davis, H. Shrobe, and P. Szolovits. What is a knowledge representation? *AI magazine*, 14(1):17–17, 1993.

[31] S. Decker, P. Mitra, and S. Melnik. Framework for the semantic web: an rdf tutorial. *IEEE Internet Computing*, 4(6):68–73, 2000.

[32] E. Demir, M. P. Cary, S. Paley, K. Fukuda, C. Lemer, I. Vastrik, G. Wu, P. D'eustachio, C. Schaefer, J. Luciano, et al. The biopax community standard for pathway data sharing. *Nature biotechnology*, 28(9):935, 2010.

[33] R. Dyckhoff and W. Tholen. Exponentiable morphisms, partial products and pullback complements. *Journal of Pure and Applied Algebra*, 49(1-2):103–116, 1987.

[34] H. Ehrig, C. Ermel, and G. Taentzer. A formal resolution strategy for operation-based conflicts in model versioning using graph modifications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 202–216. Springer, 2011.

[35] H. Ehrig, U. Golas, A. Habel, L. Lambers, and F. Orejas. M-adhesive transformation systems with nested application conditions. part 1: parallelism, concurrency and amalgamation. *Mathematical Structures in Computer Science*, 24(4), 2014.

[36] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Mathematical Structures in Computer Science*, 1(3):361–404, 1991.

[37] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 167–180. IEEE, 1973.

[38] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In *International conference on graph transformation*, pages 161–177. Springer, 2004.

[39] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630. ACM, 2015.

[40] J. R. Faeder, M. L. Blinov, and W. S. Hlavacek. Rule-based modeling of biochemical systems with bionetgen. In *Systems biology*, pages 113–167. Springer, 2009.

[41] J. Feret and K. Q. Lỳ. Reachability analysis via orthogonal sets of patterns. *Electronic Notes in Theoretical Computer Science*, 335:27–48, 2018.

[42] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Commun. ACM*, 28(9):904–920, 1985.

[43] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018.

[44] D. T. Gillespie. Stochastic simulation of chemical kinetics. *Annu. Rev. Phys. Chem.*, 58:35–55, 2007.

[45] B. M. Gyori, J. A. Bachman, K. Subramanian, J. L. Muhlich, L. Galescu, and P. K. Sorger. From word models to executable models of signaling networks using automated assembly. *Molecular systems biology*, 13(11):954, 2017.

[46] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3, 4):287–313, 1996.

[47] R. Harmer. Rule-based modelling and tunable resolution. *arXiv preprint arXiv:0911.2508*, 2009.

[48] R. Harmer. *Rule-based meta-modelling for bio-curation*. PhD thesis, 2017.

[49] R. Harmer, Y.-S. Le Cornec, S. Légaré, and E. Oshurko. Bio-curation for cellular signalling: the kami project. *IEEE/ACM transactions on computational biology and bioinformatics*, 2019.

[50] R. Harmer, Y.-S. Le Cornec, S. Légaré, and I. Oshurko. Bio-curation for cellular signalling: The kami project. In *Computational Methods in Systems Biology: 15th International Conference, CMSB 2017, Darmstadt, Germany, September 27–29, 2017, Proceedings*, volume 10545, page 3. Springer, 2017.

[51] R. Harmer and E. Oshurko. Kamistudio: An environment for biocuration of cellular signalling knowledge. In *International Conference on Computational Methods in Systems Biology*, pages 322–328. Springer, 2019.

[52] R. Harmer and E. Oshurko. Knowledge representation and update in hierarchies of graphs. In E. Guerra and F. Orejas, editors, *Graph Transformation*, pages 141–158, Cham, 2019. Springer International Publishing.

[53] R. Harmer and E. Oshurko. Knowledge representation and update in hierarchies of graphs. *arXiv preprint arXiv:2002.01766*, 2020.

[54] M. Hartung, J. Terwilliger, and E. Rahm. Recent advances in schema and ontology evolution. In *Schema matching and mapping*, pages 149–190. Springer, 2011.

[55] H. Hermjakob, L. Montecchi-Palazzi, C. Lewington, S. Mudali, S. Kerrien, S. Orchard, M. Vingron, B. Roechert, P. Roepstorff, A. Valencia, et al. Intact: an open source molecular interaction database. *Nucleic acids research*, 32(suppl_1):D452–D455, 2004.

[56] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, and W. Lehner. Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1101–1116. ACM, 2017.

[57] J. X. Huang, G. Lee, K. E. Cavanaugh, J. W. Chang, M. L. Gardel, and R. E. Moellering. High throughput discovery of functional protein modifications by hotspot thermal profiling. *Nature Methods*, 2019.

[58] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, et al. The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[59] J. Huelsken and J. Behrens. The wnt signalling pathway. *Journal of cell science*, 115(21):3977–3978, 2002.

[60] R. Isserlin, R. A. El-Badrawi, and G. D. Bader. The biomolecular interaction network database in psi-mi 2.5. *Database*, 2011, 2011.

[61] S. Lack and P. Sobocinski. Adhesive categories. In *International Conference on Foundations of Software Science and Computation Structures*, pages 273–288. Springer, 2004.

[62] S. Lack and P. Sobociński. Adhesive and quasiadhesive categories. *RAIRO-Theoretical Informatics and Applications*, 39(3):511–545, 2005.

[63] C. F. Lopez, J. L. Muhlich, J. A. Bachman, and P. K. Sorger. Programming biological models in python using pysb. *Molecular systems biology*, 9(1):646, 2013.

[64] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1-2):181–224, 1993.

[65] M. Löwe. Polymorphic sesqui-pushout graph rewriting. In *International Conference on Graph Transformation*, pages 3–18. Springer, 2015.

[66] F. Mantz, G. Taentzer, Y. Lamo, and U. Wolter. Co-evolving meta-models and their instance models: A formal approach based on graph transformation. *Science of Computer Programming*, 104:2–43, 2015.

[67] D. L. McGuinness, F. Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[68] M. Mesiti, R. Celle, M. A. Sorrenti, and G. Guerrini. X-evolution: A system for xml schema evolution and document adaptation. In *International Conference on Extending Database Technology*, pages 1143–1146. Springer, 2006.

[69] M. Minsky. A framework for representing knowledge. 1974.

[70] S. Orchard, H. Hermjakob, and R. Apweiler. The proteomics standards initiative. *PROTEOMICS: International Edition*, 3(7):1374–1376, 2003.

[71] L. O'Hara, A. Livigni, T. Theo, B. Boyer, T. Angus, D. Wright, S.-H. Chen, S. Raza, M. W. Barnett, P. Digard, et al. Modelling the structure and dynamics of biological pathways. *PLoS biology*, 14(8):e1002530, 2016.

[72] M. Pedersen, A. Phillips, and G. D. Plotkin. A high-level language for rule-based modelling. *PloS one*, 10(6):e0114296, 2015.

[73] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion: Next Generation Open Source Version Control*. " O'Reilly Media, Inc.", 2008.

[74] M. R. Quillian. Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral science*, 12(5):410–430, 1967.

[75] E. Rahm and P. Bernstein. An online bibliography on schema evolution. 2006.

[76] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[77] J. M. Song, A. Menon, D. C. Mitchell, O. T. Johnson, and A. L. Garner. High-throughput chemical probing of full-length protein–protein interactions. *ACS combinatorial science*, 19(12):763–769, 2017.

[78] L. Strömbäck and P. Lambrix. Representations of molecular pathways: an evaluation of sbml, psi mi and biopax. *Bioinformatics*, 21(24):4401–4407, 2005.

[79] Y. Sun, W.-Z. Liu, T. Liu, X. Feng, N. Yang, and H.-F. Zhou. Signaling pathway of mapk/erk in cell proliferation, differentiation, migration, senescence and apoptosis. *Journal of Receptors and Signal Transduction*, 35(6):600–604, 2015.

[80] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. Conflict detection for model versioning based on graph modifications. In *International Conference on Graph Transformation*, pages 171–186. Springer, 2010.

[81] J. F. Terwilliger, P. A. Bernstein, and A. Unnithan. Worry-free database upgrades: automated model-driven evolution of schemas and complex mappings. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1191–1194. ACM, 2010.

[82] M. Uschold and M. Gruninger. Ontologies: Principles, methods and applications. *The knowledge engineering review*, 11(2):93–136, 1996.

[83] M. A. Valenzuela-Escárcega, O. Babur, G. Hahn-Powell, D. Bell, T. Hicks, E. Noriega-Atala, X. Wang, M. Surdeanu, E. Demir, and C. T. Morrison. Large-scale automated reading with reach discovers new cancer driving mechanisms. In *Proceedings of the Sixth BioCreative Challenge Evaluation Workshop. Bethesda*, pages 201–3, 2017.

[84] Z.-H. You, Y.-K. Lei, J. Gui, D.-S. Huang, and X. Zhou. Using manifold embedding for assessing and predicting protein interactions from high-throughput experimental data. *Bioinformatics*, 26(21):2744–2751, 2010.