ÉCOLE NORMALE SUPÉRIEURE DE LYON

Graph-based hierarchical knowledge representation

an application to rule-based modelling and bio-curation

by

Eugenia Oshurko

Report of M2 Internship

in the
PLUME team (CNRS, ENS Lyon, UCBL, LIP)
under supervision of Russ Harmer
Computer Science Department

June 2017

Abstract

The aim of this project was to develop a framework, both conceptual and software, which would allow coherent aggregation of fragmentary biological facts about individual protein-protein interactions. This framework at the same time tries to solve a bio-curation problem and enables the accommodated knowledge to be directly ‘executed’ with tools for rule-based modelling and simulations.

Moreover, our goal was to build an expressive and mathematically robust general-purpose knowledge representation system that can be used for building graph-based models in any domain. As the result of our efforts, the Python library *ReGraph* was developed and was adopted as the main tool for building a knowledge representation system for the specific use case of modelling in cellular signalling.

The second part of the project was dedicated to the development of a software platform *KAMI: Knowledge Aggregator and Model Instantiator* for semi-automatic aggregation and annotation of knowledge about individual protein-protein interactions from multiple sources. This platform also provides means for automatic instantiation of knowledge into concrete systems’ models and their export to a rule-based modelling language *Kappa*.

Contents

1 Introduction and background

Knowledge about the behaviour of systems in sciences such as physics and chemistry traditionally relies on sound models built on fundamental laws. In addition, it suffices for these models to be a 'good' approximation of real phenomena. For biology, on the other hand, approaches inherited from these sciences become infeasible due to the complexity of the systems under study. Biological systems comprise an enormous numbers of hierarchically structured agents interacting in a non-linear fashion, which invalidates any crude approximations in the modelling of these systems [13].

Our main interest is focused on studies of cellular signalling mechanisms which underlie essential activities of a living cell. Thousands of highly complex molecules, proteins, take part in signalling through various protein-protein interactions (PPIs). In addition, a single protein can have multiple post-translational modifications of amino acids in various locations. All combinations of these modifications induce a large number of physical states of a protein, which alter its behaviour in PPIs [2]. Classical techniques for modelling the evolution of chemically reacting molecules (like modelling with ODE's, stochastic chemical kinetics) require an explicit listing of molecular species in the system. In the context of cellular signalling a protein in every physical state is regarded as a distinct molecular species [11], therefore these techniques fail as a number of species grows rapidly.

Rule-based modelling is an alternative approach to modelling PPIs which overcomes the difficulties associated with the combinatorial explosion in the number of protein agents [5, 8, 9]. It represents a system as a large graph structured in a particular way and models the system's evolution by application of a collection of graph rewriting rules which represent individual PPIs. A rule representing a PPI states only the necessary conditions for the interaction to appear. A language for rule-based modelling of protein interactions *Kappa* and a stochastic simulator *KaSim* have been developed in recent years [1].

Another difficulty which arises in the studies of biological mechanisms such as cellular signalling is also rooted in the complexity of the systems. Due to the way biological knowledge has been traditionally obtained and accumulated, it tends to be fragmentary and dispersed over multiple sources containing only partial mechanistic details of some phenomena. Knowledge of different provenance can also be incoherent or be valid only in some particular context. In addition, different biological facts can have different epistemic status: they can represent experimental results, hypotheses, or be inferred from other facts. Depending on its status every fact should be treated by a modeller in a respective way.

Therefore, there is a need of a knowledge representation system that would be able to gradually aggregate knowledge in a consistent way, and at the same time would allow maintaining variant models of the system obtained from the different versions of a single or a collection of PPIs. The aggregation of partial knowledge should be based on the understanding of the mechanisms underlying the interactions and on the refinement of phenomenological observations with detailed mechanistic descriptions of the interactions.

We would also like to build this infrastructure in a way that would enable us to directly instantiate models in *Kappa* and run simulations with *KaSim*. This way our knowledge representation system would not only accommodate data but also would be directly 'executable'.

To summarize, our goal is to build a platform that implements this knowledge representation system. The main use case lies in the systems biology field, and in particular, we are interested in building such an infrastructure for accommodating knowledge about signalling pathways in the cell.

2 Knowledge representation system

We have built an expressive and flexible mathematical model for knowledge representation (KR) based on category theory, that provides robust mechanisms for incremental aggregation of partial knowledge and for its transfer to the various representations.

The main modelling units in our KR system are graphs whose nodes correspond to arbitrary entities of the system, and edges represent relations between them (for example, hierarchical organisation of entities and so on), both equipped with dictionary attributes. Such graphs define a modeller's world-view on a system on some abstraction level. We keep our framework as generic as possible and provide a sophisticated mechanism through which 'syntax' and 'semantics' of the models can be expressed. Graphical representation of the models provides us a mechanism for expressing changes in the system, as well as changes in 'syntax' or 'semantics' with rules of graph rewriting.

Here we give the basic formalism used in the system together with the particular specification that we adopt for the solution of curation problem for cellular signalling we stated above as the main use-case of such KR system.

2.1 Basic formalism

First, we introduce a *dictionary* structure, which will be used to represent attributes assigned to different objects in the KR system. Attributes can be used to express properties of the entities in the system as well as to assign annotations to the accommodated facts (i.e. meta-data).

Definition 2.1. 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 2.2. 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:

$$\begin{tikzcd}
V_1 \arrow[r, "d_1"] \arrow[d, tail, "f"] & K_1 \arrow[d, tail, "g"] \\
V_2 \arrow[r, "d_2"] & K_2
\end{tikzcd}$$

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 $\leq$.

In the category of dictionaries **Dict** (it is also the subcategory of monos of the *arrow category* $\mathbf{Sets}^{\rightarrow}_{fin}$) the objects are dictionaries, the homomorphisms are dictionary inclusions $\leq$.

We would like to represent models in our KR graphically, therefore the main structure in the KR system is a graph equipped with the dictionary attributes.

Definition 2.3. 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 2.4. A *homomorphism* on graphs with attributes $G = (V^G, E^G, A^G_V, A^G_E, f^G, g^G)$ and $H = (V^H, E^H, A^H_V, A^H_E, f^H, g^H)$ is defined by a mapping $h : V^G \to V^H$ such that:

1. If $(u, v) \in E^G$, $(h(u), h(v)) \in E^H$ (edges are preserved).
2. For all $u \in V^G$ $f^G(u) \leq f^H(h(u))$.
3. For all $(u, v) \in E^G$, $g^G(u, v) \leq g^H(h(u), h(v))$.

The latter two definitions give the objects and the arrows of the category of graphs with attributes $\mathbf{Graphs}_{attrs}$.

We introduce a notion of *typing* for graphs with attributes via graph homomorphisms. Note that for a fixed graph T the category of graphs typed by T is a slice category $\mathbf{Graphs}_{attrs}/T$ for which the following diagram commutes:

$$\begin{tikzcd}
G \arrow[rr, "h"] \arrow[dr, "f"] & & G' \arrow[dl, "f'"] \\
& T &
\end{tikzcd}$$

i. e. $f = f' \circ h$

Consider the following example of knowledge representation with two graphs G and T and a typing of G by T via homomorphism f (represented with dashed arrows) describing simple facts about people living and working in some cities on the figure 1.

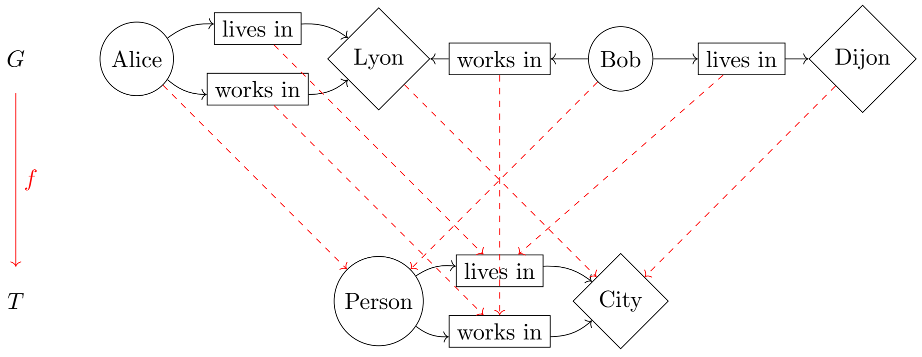


Figure 1: Example of KR system

Graph typing $G \to T$ sends nodes of the graph G to their types represented by the nodes of the graph T. Talking about graph typing by their homomorphisms, we often refer to G as a *model* and to T as a *meta-model* in this context. Meta-model in some sense defines a 'syntax' for all the graphical models it types by defining a set of allowed types (possibly with their attributes) and a set of allowed edges between different types of nodes.

2.2 Type respecting graph rewriting

Graph rewriting is a natural way to express the evolution of the system modeled by a graph with attributes. The category $\mathbf{Graphs}_{attrs}$ has all pushouts and pullback complements over injective morphisms, therefore we have a well-defined procedure of sesqui-pushout rewriting in this category [7] (more details can be found in the appendix A). Rewriting procedure applies a *rule* of the form $L \leftarrow P \rightarrow R$ to an instance (defined by a matching morphism $m_L : L \rightarrowtail G$) in a given graph G. We denote rewriting of G that produces G' as $G \rightsquigarrow G'$.

A single rule can have multiple instances (matching morphisms $m_L : L \rightarrowtail G$) in a given graph G. If G is typed by some graph T, it can be useful, however, to define a typing of L, namely a homomorphism $f_L : L \rightarrow T$. This typing restricts a set of possible matchings to the ones that commute with the typing f_L, i.e. $f \circ m_L = f_L$ as on the diagram:

$$\begin{tikzcd}
L \arrow[d, tail, "m_L"] \arrow[rdd, bend left, "f_L"] & P \arrow[l, "h_1"'] \arrow[r, "h_2"] & R \\
G \arrow[rd, "f"] & & \\
& T &
\end{tikzcd}$$

Moreover, having a typing $f : G \rightarrow T$ we are often interested in rewriting $G \rightsquigarrow G'$ such that the resulting graph G' remains typed by T, i.e. there exists a unique $f' : G' \rightarrow T$ (an object of the slice category $\mathbf{Graphs}_{attrs}/T$). A rule of such type-preserving rewriting is given by a span $L \overset{h_1}{\leftarrow} P \overset{h_2}{\rightarrow} R$ and a homomorphism $f_R : R \rightarrow T$ that defines a typing of the right-hand side of the rule. Then the typing $G' \rightarrow T$ is given by the universal property of the pushout from $G^- \leftarrowtail P \rightarrow R$ (see more details in the appendix A).

Recall the example from the figure 1 which illustrated a simple model describing facts about people living and working in some cities. Now consider the situation when Alice is being reallocated by her employer to another city, for example to Paris. The respective change in the system can be done using the graph rewriting rule depicted on the figure 2.

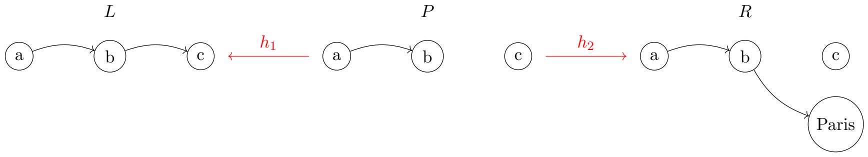


Figure 2: *Example of a graph rewriting rule.* This rule deletes the edge 'b'→ 'c', adds a new node 'Paris' and the edge 'b'→ 'Paris'.

Four different matchings $L \rightarrowtail G$ of this rule can be found in our system represented with the graph G from the example 1: (1) $m_L(a) = Alice$, $m_L(b) = lives\ in$, $m_L(c) = Lyon$; (2) $m_L(a) = Alice$, $m_L(b) = works\ in$, $m_L(c) = Lyon$; (3) $m_L(a) = Bob$, $m_L(b) = lives\ in$, $m_L(c) = Dijon$; (4) $m_L(a) = Bob$, $m_L(b) = works\ in$, $m_L(c) = Lyon$.

However, we are interested in the ones where the node ‘b’ is being matched to a node of the type ‘works in’, therefore we define the following typing f_L of L by the graph T: $f_L(a) = Person$, $f_L(b) = works\ in$, $f_L(c) = City$. Now there exist only two possible matchings of the rule that respect f_L, namely the instances (2) and (4). Finally, to model the previously described reallocation of Alice to Paris we rewrite instance (2). However, if we want the resulting graph to remain typed by T, we need to explicitly provide a typing homomorphism f_R of the right-hand side of the rule R where newly added node ‘Paris’ is mapped to the node ‘City’ from the meta-model T. The figure 3 illustrates the result of such rewriting.

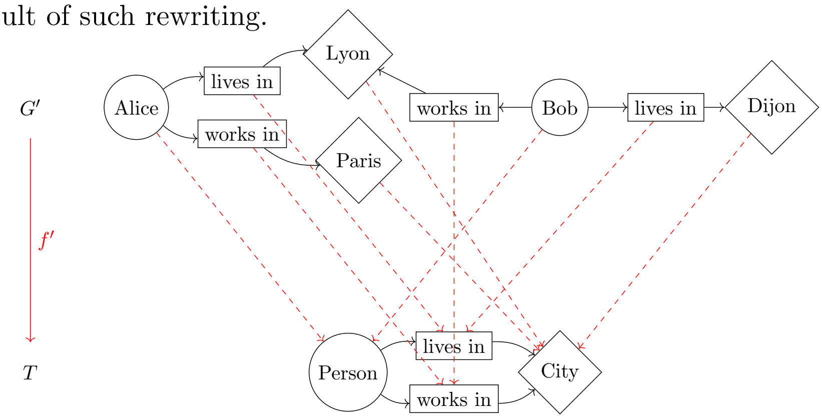


Figure 3: *KR system after rewriting.* The result of rewriting $G \rightsquigarrow G'$ obtained with an application of the rule from the figure 2 with the typing f_R of the right hand-side which maps the node ‘Paris’ to the node ‘City’ in T.

2.3 Graph hierarchy

The higher level construction which enables us to represent complex knowledge systems is a *graph hierarchy*. A graph hierarchy is a DAG, where nodes are graphs with attributes and edges are (partial) homomorphisms representing graph typing in the system. This construction provides means for mathematically robust procedures of propagation of changes (expressed through graph rewriting rules) on any level of the hierarchy, up to all the graphs which are transitively typed by the graph subject to rewriting.

Definition 2.5. A *graph hierarchy* is a DAG, where nodes are objects of the category $\mathbf{Graphs}_{attrs}$, edges represent typing homomorphisms between graphs, and for any two paths $h_1, h_2, \ldots, h_n$ and $i_1, i_2, \ldots, i_m$ from G_1 to G_2 (i.e. $dom(h_1) = dom(i_1) = G_1$ and $cod(h_n) = cod(i_m) = G_2$), the homomorphisms $f = h_n \circ h_{n-1} \circ \ldots \circ h_2 \circ h_1$ and $g = i_m \circ i_{m-1} \circ \ldots \circ i_2 \circ i_1$ (constructed with the composition of the homomorphisms along the paths) are equal.

In practice, we also allow to explicitly accommodate in the hierarchy undirected edges representing symmetric relations between graphs, which are defined by sets of pairs of nodes. A relation between two graphs G_1 and G_2 also corresponds to the span $G_1 \leftarrow G_{12} \rightarrow G_2$, therefore can be treated as such in the procedures of rewriting and propagation that will be defined in the next section. Introducing relations to a graph hierarchy would be useful further for definition of domain-specific semantic properties, as will be discussed in the section 3.

2.4 Rewriting in a hierarchy

In general, after rewriting $G \rightsquigarrow G'$ in a hierarchy, the following steps should be performed: (1) an update of the typings by all immediate successors of G; (2) a propagation of changes from G to all the graphs in the hierarchy which are transitively typed by it; (3) an update of all the edges affected by the rewriting. The following points give a detailed description of the respective steps.

1. Consider some graph G in a graph hierarchy and a set of its immediate successors (graphs typing G) as in the figure below:

$$\begin{tikzcd}
 & & G \arrow[ddll] \arrow[ddl] \arrow[ddr] \arrow[ddrr] & & \\
 & & \cdots & & \\
T_1 & T_2 & \cdots & T_{m-1} & T_m
\end{tikzcd}$$

   We want to perform a type-preserving rewriting $G \rightsquigarrow G'$ with a rule $r : L \leftarrow P \rightarrow R$ in this context, namely we want G' to be typed by $T_1, T_2, \ldots, T_n$ after rewriting, and for this purpose typing of R by every such successor should be provided as discussed in the previous section.

   Remark 2.6. In general, we allow hierarchies to have partial typing homomorphisms (more details can be found in the appendix B), in which case it is not required to type the right-hand side of the rule R. If the rewriting rule creates new nodes which are not typed by some T_k and if typing $G \rightarrow T_k$ was total, as the result of rewriting this typing becomes partial.

2. Consider a set of graphs typed by G (both immediate predecessors and graphs typed transitively via composition of homomorphisms along the path to G) $H_1, H_2, \ldots, H_n$ as on the figure below:

$$\begin{tikzcd}
H_1 \arrow[ddrr] & H_2 \arrow[ddr] & \cdots & H_{n-1} \arrow[ddl] & H_n \arrow[ddll] \\
 & & \cdots & & \\
 & & G & &
\end{tikzcd}$$

   After rewriting $G \rightsquigarrow G'$ we would like to *propagate* the changes in G to all the graphs $H_1, H_2, \ldots, H_n$ typed by it. Specifically, to preserve the coherence of the hierarchy we want to remove and clone all the nodes (and edges) that map to the nodes in G which are respectively removed or cloned. Adds and merges produced by the application of the rule do not affect the graphs typed by G in the hierarchy.

   For every graph H_k typed by G (immediately or transitively) and the span $G \leftarrow G^- \rightarrow G'$ produced as the result of sesqui-pushout rewriting procedure, propagation of deletions and clones from G to H_k can be performed with the pullback from $H_k \rightarrow G \leftarrow G^-$ as illustrated on the diagram below (here and further in this section we denote with green arrows the newly constructed homomorphisms to distinguish them from the homomorphisms initially given in the hierarchy and denoted with black arrows):

$$\begin{tikzcd}
H_k \arrow[d] & H_k^- \arrow[l] \arrow[d] & \\
G & G^- \arrow[l] \arrow[r] & G'
\end{tikzcd}$$

The graph H_k^- corresponds to the result of deletions and cloning of all the nodes in H_k which are typed by the nodes in G that were either deleted or cloned in the course of rewriting.

3. For all H_l that type H_k, which has been affected by the previously specified propagation, the typing is updated with the composition $(H_k \to H_l) \circ (H_k^- \to H_k)$ as shows the diagram below:

$$\begin{tikzcd}
 & H_k \arrow[dl] \arrow[d] & H_k^- \arrow[l] \arrow[d] \arrow[dll, bend right] & \\
H_l & G & G^- \arrow[l] \arrow[r] & G'
\end{tikzcd}$$

Consider a graph H_k typed by G via some homomorphism f_1 and a graph H_l typed by H_k via f_2 (illustrated on the diagram below). H_l is typed by G not directly, but transitively via the homomorphism constructed from the composition $f_1 \circ f_2$. In this case propagation of changes in G is performed similarly by finding the pullback from $H_l \to G \leftarrow G^-$. However, to preserve the initial graph structure of the hierarchy, a homomorphism $f_2' : H_l^- \to H_k^-$ should be found. Due to the fact that $f_1 \circ f_2 \circ h_2 = g \circ u$ and the universal property of the pullback $H_k \leftarrow H_k^- \to G^-$ from $H_k \to G \leftarrow G^-$, there exists a unique homomorphism f_2' as on the diagram below:

$$\begin{tikzcd}
H_l \arrow[d, "f_2"] \arrow[dd, bend right] & H_l^- \arrow[l, "h_2"'] \arrow[d, dashed, "f_2'"'] \arrow[dd, bend left, "u"] \\
H_k \arrow[d, "f_1"] & H_k^- \arrow[l, "h_1"'] \arrow[d, "f_1'"'] \\
G & G^- \arrow[l, "g"']
\end{tikzcd}$$

A special case arises when we propagate the changes to a graph whose predecessor has already been updated. For example, consider a hierarchy with the following structure: $H_k \to G$, $H_l \to G$ and $H_k \to H_l$ (denoted with black arrows on the diagram below). The changes to H_k have already been propagated (and therefore H_k^- has been constructed). Now we would like to propagate the changes to H_l, and to find an updated edge $H_k^- \to H_l^-$. H_l^- is constructed the same way as previously by finding the pullback $H_l \leftarrow H_l^- \to G^-$ from $H_l \to G \leftarrow G^-$. From the fact that $g \circ f_1' = f_2 \circ f_3 \circ h_1$ and the universal property of the pullback follows that there exists a unique homomorphism $f_3' : H_k^- \to H_l^-$ (denoted with red dashed arrow on the diagram).

$$\begin{tikzcd}
 & H_k^- \arrow[rr, dashed, "f_3'"] \arrow[dl, "h_1"'] \arrow[ddr, "f_1'"] & & H_l^- \arrow[dl, "h_2"'] \arrow[ddl, "f_2'"] \\
H_k \arrow[rr, "f_3"] \arrow[dr, "f_1"'] & & H_l \arrow[dl, "f_2"'] & \\
 & G & G^- \arrow[l, "g"] &
\end{tikzcd}$$

The high-level pseudocode of propagation procedure is presented in the appendix C. Starting from the initial rewritten graph G this procedure traverses a hierarchy in the breadth-first manner (in the reverse direction of the edges). Traversal propagates the changes to all the graphs transitively typed by G and updates all the edges affected by the propagation.

3 Knowledge representation for modeling of protein-protein interactions

We now present the specification of the knowledge representation system for rule-based modelling of protein-protein interactions. This KR system is the main component of the bio-curation platform *KAMI: Knowledge Aggregator and Model Instantiator* that will be presented section 5. Here we gradually present the graphs it encapsulates as well as the functions and the relations between these graphs, and their interpretation and usage in the rule-based modeling framework.

3.1 Meta-model

First we define a concrete graph M representing the *meta-model* for our protein-protein interaction KR. The meta-model defines a set of agents and actions in rule-based models, as well as a set of allowed edges between different types of nodes. The adopted meta-model (shown on the figure 4) represents a refined version of the meta-model previously presented in [5].

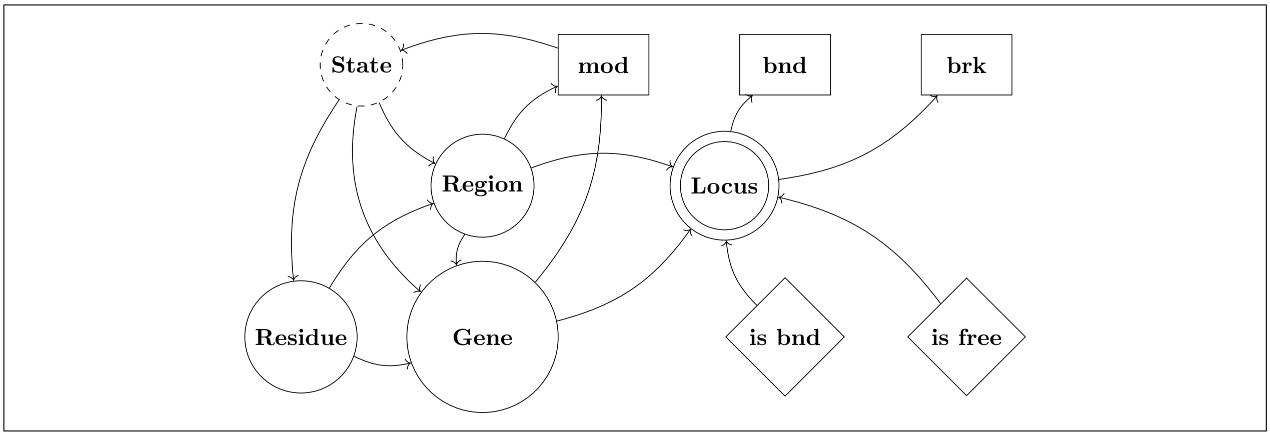


Figure 4: Graph M, the meta-model for protein-protein interaction knowledge representation

In figure 4 we schematically denote nodes corresponding to agents and their structural elements with circular nodes, and interactions with rectangular nodes. The node **State** denoted with dashed borders represents a special type of node with a unique attribute whose value can be modified during system evolution; using nodes of this type we will express modifications of different protein states such as phosphorylation, activity state, etc. The node **Locus** is another special node that, unlike other agent nodes in the meta-model (such as **Region** or **Residue**), does not represent a physical structural element of a protein, but serves a purely formal purpose in the meta-model, which will be explained later. The main protein-protein interactions of interest are modification of protein states, protein binding and breaking of protein bonds. In the meta-model these interactions are represented explicitly by the nodes **mod**, **bnd** and **brk** respectively. With diamonds we denote the nodes that we call 'test' nodes, and that are used to express the conditions of presence or absence of bindings.

By choosing this meta-model we build our KR system around a notion of *gene* as our 'reference entity'. From genes we can define their products (i.e. proteins) by providing specific regions, residues and states. Therefore, in our system an agent actually represents not a single protein, but a feasible *'neighbourhood in sequence space'* [5] of gene products. For example, we can specify an interaction, for which a particular gene product should have a specific mutation of some residue, by attaching a **Residue** node with a fixed value of amino acid corresponding to the mutation. More examples of the biological facts expressed with graphs typed by the meta-model M will follow.

3.2 Nuggets

Nugget is the main unit of information in our knowledge representation system. It describes a single biological fact, and formally it is a connected graph typed by the meta-model M. In addition, every nugget has the following properties: (1) it contains exactly one action node (a node of the type **mod**, **bnd** or **brk**); (2) any **bnd** and **brk** node has exactly two out-going edges; (4) for every agent node in a nugget there exists a path to a gene node. The first property forces our nuggets to contain knowledge about a single protein-protein interaction, property 2 guarantees that the expressed interactions are semantically correct (according to the domain-specific semantics of protein interactions), and the last one guarantees that the participating agents are grounded to their reference gene.

Consider the following example of a nugget:

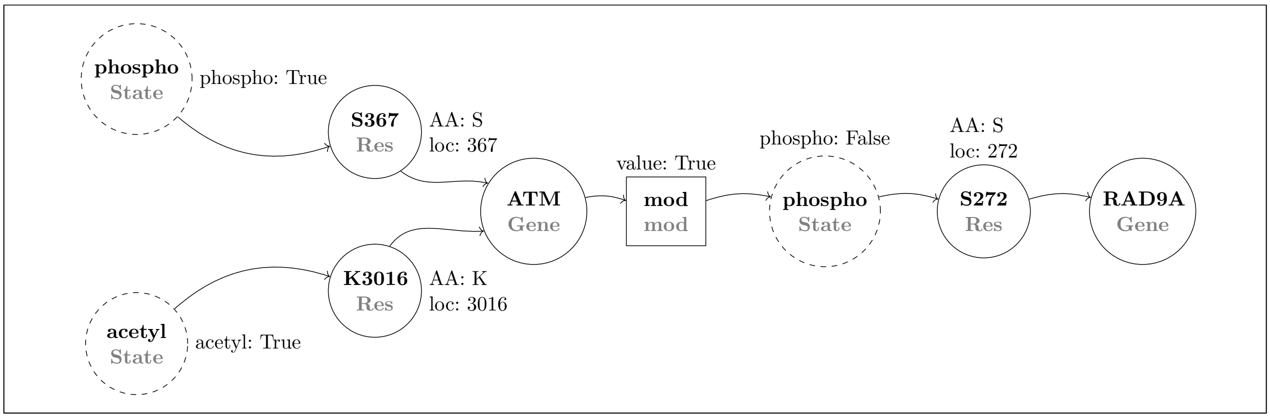


Figure 5: Example of a modification nugget. This nugget represents the following piece of knowledge: *"A protein ATM phosphorylates residue S272 of a protein RAD9A, when phosphorylated at the residue S367 and acetylated at the residue K3016"*. A type of a node in the meta-model M is given below the node's label (with the grey color), and attributes of a node are represented with a set of key-value pairs outside of the node.

In this typical modification nugget, the residues attached to the gene nodes express the *necessary* conditions for the modification to occur. For example, to take part in this interaction a molecule of the ATM protein should have key residues in the appropriate states (with the phosphorylated amino acid serine at the location 367, and the acetylated amino acid lysine at the location 3016). A name of amino acid in a residue node is encoded with the attribute 'AA' and a residue location with the attribute 'loc'. State nodes have exactly one attribute whose name corresponds to the name of the state and whose value is from the set $\{True,\ False\}$. Modification node **mod**, together with the state node **phospho** it points to, defines an interaction represented in a nugget, namely, it states that as the result of this interaction the phosphorylation state of the residue S272 of a molecule of $RAD9A$ will be set to $True$.

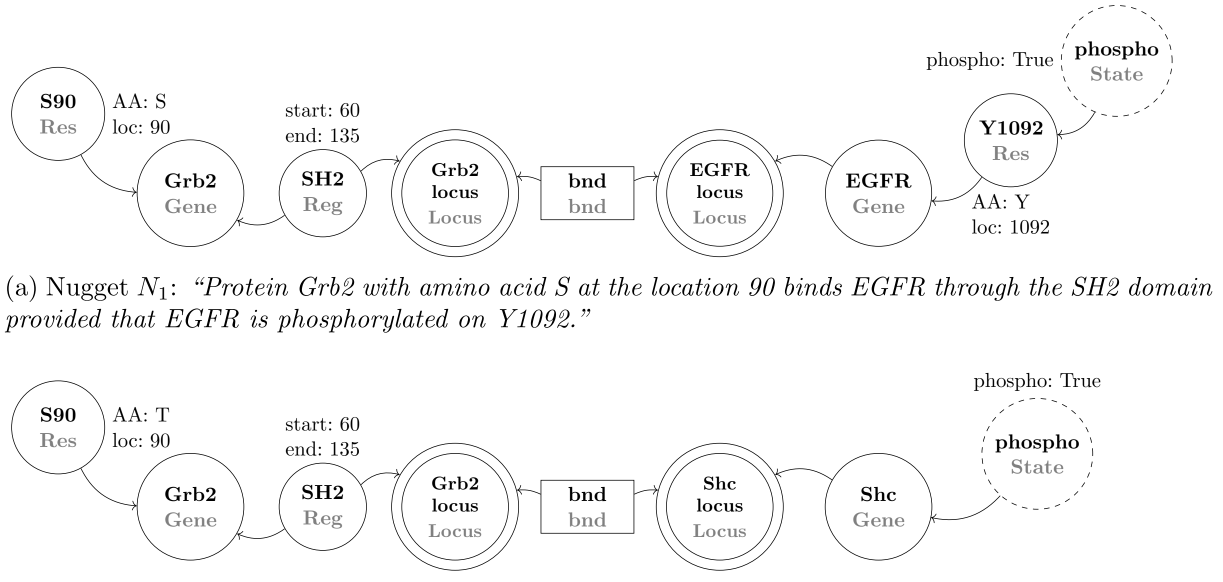
3.3 Action graph

An *action graph* $A_{\vec{N}}$ associated with a collection of nuggets $\vec{N}$ (also called a *pre-model* in [5]) is a graph typed by the meta-model M that types every nugget in this collection (via a respective collection of homomorphisms $\vec{N} \rightarrow A_{\vec{N}}$). An action graph assembles the knowledge represented in the nuggets from $\vec{N}$, and together with $\vec{N} \rightarrow A_{\vec{N}}$ it specifies the interpretation of nuggets.

Having a collection of nuggets $\vec{N} = \{N_1, N_2, \dots, N_m\}$, $A_{\vec{N}}$, and $\vec{N} \rightarrow A_{\vec{N}} = \{f_1 : N_1 \rightarrow A_{\vec{N}}, f_2 : N_2 \rightarrow A_{\vec{N}}, \dots, f_m : N_m \rightarrow A_{\vec{N}}\}$, for any two nuggets $N_k = (V_k, E_k)$ and $N_l = (V_l, E_l)$, we say that a node $u \in V_k$ corresponds to a node $v \in V_l$, if they are mapped to the same node in $A_{\vec{N}}$, i.e. if $f_k(u) = f_l(v)$. For example, if two gene nodes in different nuggets map to the same gene node in an action graph, these two nodes are interpreted as the same gene.

Note that for a given collection of nuggets we can construct multiple action graphs. Therefore, we say that a *model* of a system is given by a tuple $(\vec{N}, A_{\vec{N}}, \vec{N} \rightarrow A_{\vec{N}})$ [5], thus by a collection of facts and their interpretation defined in terms of correspondences between agents and interactions mentioned in these facts.

Consider two binding nuggets presented on the figure 6. These nuggets represent two rules of protein binding. In the first one a molecule of Grb2 binds to a molecule of EGFR through the SH2 domain region, and in the second one a molecule of the same protein Grb2 binds to a molecule of Shc through the SH2 domain.



(a) Nugget N_1: *"Protein Grb2 with amino acid S at the location 90 binds EGFR through the SH2 domain provided that EGFR is phosphorylated on Y1092."*

(b) Nugget N_2: *"Protein Grb2 with amino acid T at the location 90 binds Shc through the SH2 domain provided that Shc is phosphorylated."*

Figure 6: Examples of binding nuggets N_1 and N_2 (from [5])

Now consider the action graph A illustrated on the figure 7, and two functions $f_1 : N_1 \to A$ and $f_2 : N_2 \to A$ which map the nodes from the nuggets to the action graph nodes with the same name (labeled with bold font on the figures), except for the **bnd** and **Grb2 locus** nodes, namely, f_1(**Grb2 locus**) = **Grb2 locus 1**, f_2(**Grb2 locus**) = **Grb2 locus 2**, f_1(**bnd**) = **bnd 1**, f_2(**bnd**) = **bnd 2**.

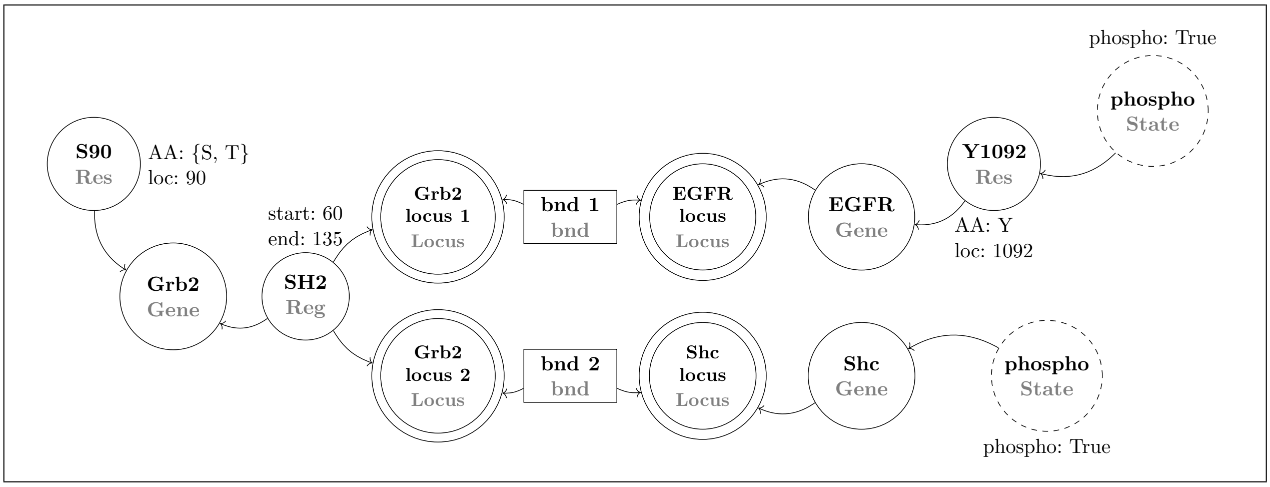


Figure 7: Action graph A

In the model $(\{N_1, N_2\}, A, \{f_1, f_2\})$ the interactions represented by the nuggets are interpreted in such a way that two bindings can happen at the same time, namely a molecule of Grb2 can bind both EGFR and Shc at the same time, creating the Grb2/EGFR/Shc complex. Now, consider another action graph A' depicted on the figure 8 and the functions $f'_1 : N_1 \to A'$ and $f'_2 : N_2 \to A'$ for which f'_1(**EGFR locus**) = **EGFR/Shc locus** and f'_2(**Shc locus**) = **EGFR/Shc locus**, and all the other nodes are mapped to the action graph nodes with the same name.

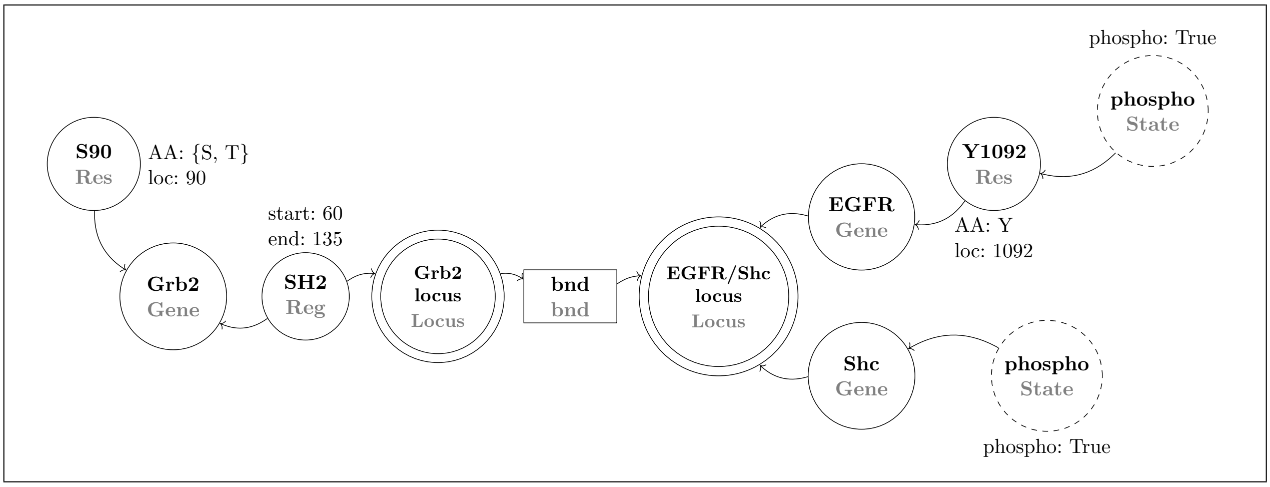


Figure 8: Action graph A'

In the model $(\{N_1, N_2\}, A', f'_1, f'_2)$ the biological facts represented by the nuggets are interpreted in a different way: a molecule of Grb2 can bind either EGFR or Shc, but not both at the same time, as the bindings use the same locus and so the two interactions are conflicting. For this set of nuggets another action graph A'' can be constructed (see figure 9) with similar functions f''_1 and f''_2 for which $f''_1(\mathbf{bnd}) = \mathbf{bnd\ 1}$, $f''_2(\mathbf{bnd}) = \mathbf{bnd\ 2}$, and the other nodes from the nuggets are sent to the action graph nodes with the same names as before.

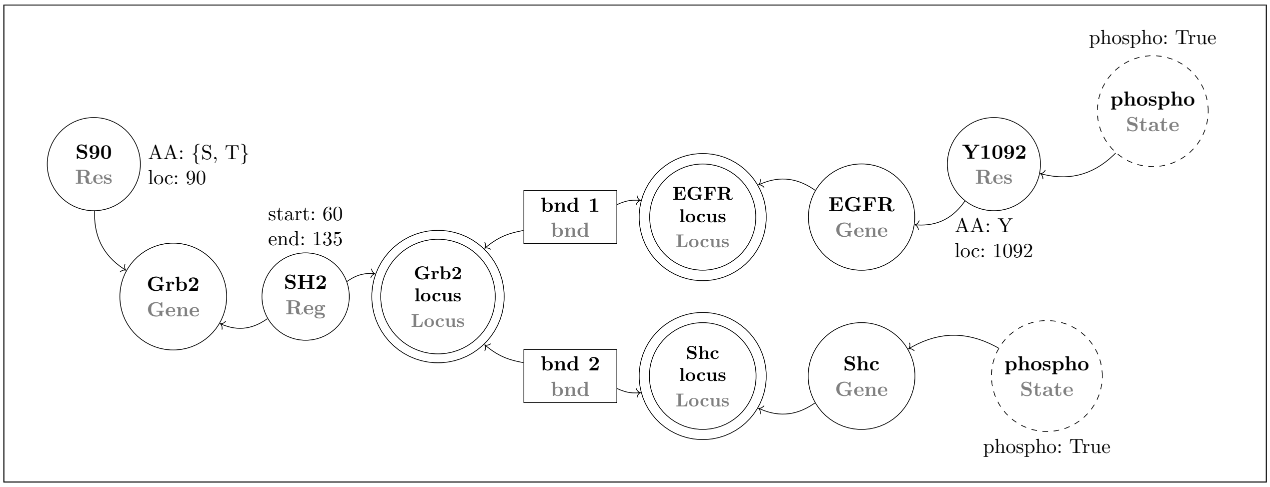


Figure 9: Action graph A''

In the model given by a tuple $(\{N_1, N_2\}, A'', \{f''_1, f''_2\})$, the interactions are conflicting as in the previous case, therefore Grb2 cannot bind EGFR and Shc at the same time, but in this case the binding mechanisms of Gbr2/EGFR and Gbr2/Shc bindings are different, and so represented in the model with two different binding nodes. This situation may appear, for example, when two bindings use different but overlapping sites, then both the interactions cannot happen at the same time.

We have presented here an example of three alternative models that can be build from one set of nuggets. We have also illustrated how locus nodes can be used to define different interpretations of binding interactions. The choice of a particular model, however, belongs to a modeller and often requires expert knowledge. Alternatively, this choice can be hypothesized and tested with simulation tools. However, there exist some possibilities to choose a model automatically using additional domain-specific knowledge provided to the KR system. In the following section we describe how we address the problems associated with the automatic model choice, whenever this choice can be made by the KR system.

3.4 Semantic nuggets and semantic action graph

We incorporate in our KR system a collection $\vec{SN}$ of graphs called *semantic nuggets* which represent 'templates' of biological facts containing knowledge about semantic properties of PPIs and their mechanisms, and which are typed by the meta-model. These properties play a crucial role in building models and understanding the mechanisms of interactions. A semantic nugget in our KR will often contain knowledge about specific protein domain, its functional properties. Fragments of concrete nuggets related (with the previously presented symmetric binary relations on graphs) to this semantic nugget would inherit the functional properties specified, and therefore the interaction mechanism. We also aggregate $\vec{SN}$ into a predefined *semantic action graph*, similarly to the way a collection of

$\vec{N}$ is aggregated in the action graph A in the previous section, and relate the action graph to the semantic action graph.

Semantic nuggets help us to (1) check that an input interaction is semantically correct; (2) autocomplete nuggets with more mechanistic details if required; (3) impose constraints on the action graph (which helps us to build and constrain the models).

For example, consider the semantic nugget presented on the figure 10:

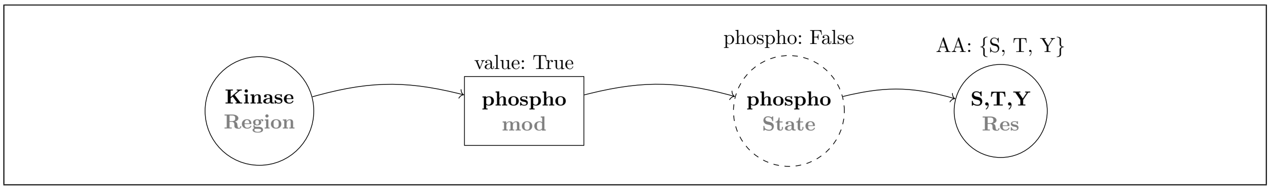


Figure 10: *"Protein kinase domain phosphorylates residues S, T or Y."*

This nugget represents the semantics of phosphorylation of protein kinase domain. We relate it to all modification nuggets corresponding to phosphorylation, which helps us to check their semantic consistency and to autocomplete them. For example:

- The nugget *"MEK1 phosphorylates STAT3"* would be autocompleted to *"MEK1 phosphorylates STAT3 though its protein kinase domain"*.
- The nugget *"MEK1 phosphorylates a residue with amino acid A of STAT3"* would be not correct semantically, as phosphorylation modification can happen only for key residues from the set $\{S, T, Y\}$.

Now consider some semantic action graph typing this phosphorylation nugget with a constraint on the number of outgoing edges from the node typing **Kinase** in the nugget. This constraint represents the fact that a kinase domain possesses a unique site for performing phosphorylation of residues. This constraint helps us to automatically map every modification action node attached to the kinase region of some protein to exactly one modification node in the action graph, which is related to the node typing **phospho** in the semantic action graph (the related clarifying example can be found in the appendix E).

These notions of semantic nuggets and semantic action graph serve as an example of how a 'semantics' of graphical models can be defined in a graph hierarchy.

3.5 Hierarchy

The shape of the hierarchy for our signalling KR system summarizing this section is presented in the diagram on the right. Here, red dashes between $\vec{N}$ and $\vec{SN}$, and between $\vec{A}$ and $\vec{SA}$ on the diagram represent symmetric graph relations. This hierarchy specification is used by our bio-curation platform KAMI that will be presented in section 5.

$$\begin{tikzcd}
\vec{N} \arrow[rr, dash, red] \arrow[d] & & \vec{SN} \arrow[d] \\
A \arrow[rr, dash, red] \arrow[dr] & & SA \arrow[dl] \\
& M &
\end{tikzcd}$$

4 Graph rewriting library ReGraph

The previous two sections presented the formalism for graph-based hierarchical KR and its specification for modelling individual PPIs. The following two sections present the generic software implementing the formalism and the platform for accommodating and managing the domain-specific graph hierarchy for rule-based modelling of PPIs.

In collaboration with other interns and post-docs involved in the project I started working on the Python library for graph rewriting `ReGraph` during my M1 internship. By the beginning of the M2 internship `ReGraph` contained data structures and functionality necessary for the basic rewriting and knowledge representation purposes.

`ReGraph` implements a general-purpose framework for modelling graph-based systems, and is the main tool used for building our domain-specific knowledge infrastructure. Most of the functionality of the library uses graph data structures from the `NetworkX` library for graph manipulation (hereafter we refer to this as `nx`), which is widely used in the network science community and contains a great number of implemented algorithms on graphs. `ReGraph` provides various utilities for graph rewriting which can be used for modelling the evolution of a system represented by a graph subject to rewriting. The rewriting functionality is based on the *sesqui-pushout rewriting procedure* [7]. In addition, the library enables a user to define a typing for models (graphs) that gives specifications for the structure of the models. This later functionality allows both to preserve the specified structure during rewriting and to propagate the changes to the specifications up to the models.

However, in order to make our knowledge infrastructure more powerful, a substantial refactoring of `ReGraph` was required. The main goal of this refactoring was to make the functionality more general, more flexible and, most importantly, more rigorous from the mathematical point of view.

The major novelty of the new version of `ReGraph` lies in replacing the rigid typing structure introduced by `Typed(Di)Graph` with a DAG-like structure implementing a typing hierarchy, where a directed edge is a typing homomorphisms between two graphs. The following section will detail the data structures and the functionality developed during this refactoring process. The source code of the library can be found by the following link `https://github.com/Kappa-Dev/ReGraph`. Also a short tutorial can be found by the link[1].

4.1 Library overview

`ReGraph` contains a collection of utilities for graph rewriting on `nx` graph objects, both undirected graphs (`nx.Graph`) and directed ones (`nx.DiGraph`). The library consists of the following main modules:

- `regraph.primitives` contains a set of functions for manipulating `nx` graphs (slightly extends the functionality provided by `nx.Graph` and `nx.DiGraph`). In addition, provides functions for finding subgraph matching in a graph (`regraph.primitives.find_matching`) and rewriting of a graph with a rule (`regraph.primitives.rewrite`). Subgraph matching procedure uses

[1] `https://github.com/Kappa-Dev/ReGraph/blob/master/examples/ReGraph_demo.ipynb`

nx.networkx.algorithms.isomorphism.(Di)GraphMatcher class which provides the method for subgraph isomorphism finding based on the VF2 algorithm [6]. The rewriting function implements sesqui-pushout rewiting.

- regraph.rules provides the data structure for representation of graph rewriting rules regraph.rules.Rule, which encapsulates the rule span $L \leftarrow P \rightarrow R$ used in sesqui-pushout rewriting. In addition, it provides an interface for gradual creation of rules from a pattern graph (left-hand side L) in an imperative way by either invoking primitive methods (methods corresponding to individual actions on the pattern like addition of nodes, cloning of nodes, etc.) or providing a respective list of commands (following a simple grammar for specifying graph transformations that can be parsed by regraph.parser).
- regraph.category contains a set of various category theory operations on graphs with attributes and their homomorphisms used in sesqui-pushout rewriting and propagation in a graph hierarchy, such as pullback, pushout, pullback complement, etc. It also provides functions for checking validity of graph homomorphisms and functions for finding unique maps by the universal property of pullbacks.
- regraph.hierarchy provides an implementation of the graph hierarchy data structure (more details follow in the section 4.2).

Some examples of ReGraph usage can be found in the appendix D.

4.2 Graph hierarchy data structure

Graph hierarchy, the specification of which was presented in the section 2.3, is implemented in regraph.hierarchy.Hierarchy (hereafter we refer to it as Hierarchy) data structure. Essentially, the data structure is a DAG. It inherits the nx.DiGraph class, therefore all the functionality for manipulating directed graphs can be applied to the hierarchy. Additionally, during the initialization and addition of edges, an instance of Hierarchy is checked to be acyclic and to respect the property of commuting paths from the same source to the same target.

Hierarchy supports two types of nodes: regraph.hierarchy.GraphNode and regraph.hierarchy.RuleNode, thus we slightly extend the structure from the section 2.3 by allowing to incorporate rewriting rules in a hierarchy as well. Rules are typed by edges which contain two typing homomorphisms corresponding to the left-hand side typing and the right-hand side typing. No nodes are allowed to be typed by a rule node, therefore during construction of the hierarchy and addition of edges Hierarchy forbids incoming edges for rule nodes. Incorporating rules into the hierarchy allows us to propagate changes to all the rules typed by a graph subject to rewriting, which is useful for rules that have to remain coherent with their meta-models during the modelling process.

Edges in a hierarchy are directed and represent typing homomorphisms. Two types of edges are allowed in the hierarchy: regraph.hierarchy.Typing (typing of a graph by another graph) and regraph.hierarchy.RuleTyping (typing of a rule by a graph, which includes left-hand side typing and right-hand side typing). We also introduce another type of edges between the nodes representing graph relations (`regraph.hierarchy.GraphRelation`). Relational edges are undirected, and are treated separately from the typing edges in the hierarchy.

5 KAMI: Knowledge Aggregator and Model Instantiator

KAMI: Knowledge Aggregator and Model Instantiator is a semi-automatic bio-curation platform which provides a user with tools for gradual aggregation of biological facts about individual protein-protein interactions of different provenance, their annotation, visualisation and further instantiation for building rule-based models of concrete systems and performing hypothesis testing.

The platform allows to input annotated biological facts through the graphical interface or programmatically through the API. The input facts can be automatically merged with an existing model in a context-dependent manner (the knowledge present in the model influences the way these facts are interpreted and, therefore, merged with the model). The model itself is a graph hierarchy (an instance of ReGraph's core data structure). Apart from the user input knowledge, the hierarchy contains expert knowledge used by KAMI for the interpretation of interactions, for example, definitions of protein families, protein variants definitions, the semantics of particular protein domains. This expert knowledge is obtained from both publicly available data (such as UniProt, Pfam, InterPro databases) and from collaboration with biologists. From the knowledge aggregated by a user in the hierarchy, KAMI allows to automatically instantiate models expressed with *Kappa* rules and perform simulations with *KaSim*. The source code of the library can be found by in the repository by the link[2].

5.1 Architecture

We have designed KAMI as a server-client application, where the server accommodates the hierarchy presented in the section 3.5, and contains a set of tools for accessing and manipulating the hierarchy, instantiating concrete models, and automatically resolving input knowledge for aggregation to the hierarchy. The client part of KAMI provides the graphical user interface and the application programming interface (API), as well as tools for the import of knowledge from various biological data formats. Figure 11 illustrates a schematic architecture of the main modules of KAMI and their relationships.

5.2 KAMI server

A web-server application for KAMI is being implemented by my collaborator Yves-Stan Le Cornec using Python micro-framework `Flask`. It follows a paradigm of thin-client and encapsulates the majority of functionality for manipulating the KAMI hierarchy. The server-side of KAMI incorporates several modules which are able to perform various tasks related to the automatic aggregation and instantiation of knowledge.

[2] `https://github.com/Kappa-Dev/KAMI-python`

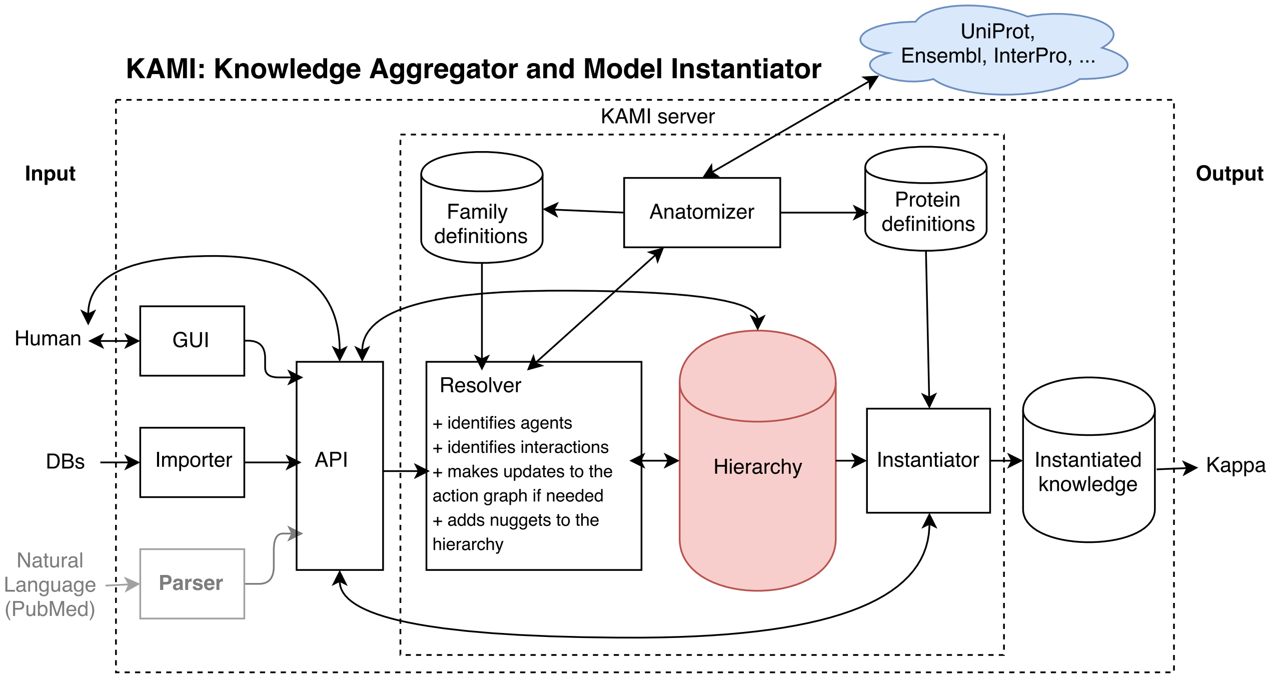

KAMI: Knowledge Aggregator and Model Instantiator
UniProt, Ensembl, InterPro, ...
KAMI server
Input
Output
Family definitions
Anatomizer
Protein definitions
Human
GUI
API
Resolver
+ identifies agents
+ identifies interactions
+ makes updates to the action graph if needed
+ adds nuggets to the hierarchy
Hierarchy
Instantiator
Instantiated knowledge
Kappa
DBs
Importer
Natural Language (PubMed)
Parser

Figure 11: High-level architecture of modules in KAMI

My main contribution to the KAMI project during the internship concerns the implementation of an *import* from different biological data formats (currently the INDRA [12] importer is implemented) and the development of a *resolver* of protein-protein interactions, which is able to automatically aggregate individual facts into the existing hierarchy.

Resolver

The main purpose of the resolver module is, provided an input protein-protein interaction, to be able to automatically:

1. create a nugget graph that respects the meta-model and corresponds to the most detailed mechanistic description possible of the interaction (including the interpretation of phenomenological interactions, auto-completion of nuggets);
2. identify genes, regions, residues and states mentioned in the interaction and map the corresponding nugget nodes to the ones in the existing action graph; alternatively, if the corresponding nodes do not exist in the action graph, the resolver is able to add new nodes to the action graph;
3. disambiguate and identify the interaction and map it into the corresponding node in the action graph. For binding interactions, in particular, the resolver should identify the interaction together with all the loci attached to it (as in the example from 3.3);
4. finally, add the nugget together with its typing by the action graph to the hierarchy.

For the purposes of input knowledge representation a set of intermediate data structures was implemented (in `kami.data_structures.entites` and `kami.data_structures.interactions` modules). These data structures represent biological entities of interest and their interactions. They serve both as the bridge between the input representation formats and our graphical representation, and as the unified interface for programmatic input of knowledge to KAMI. The idea behind these data structures is similar to the one of `Statements` in INDRA: Integrated Network and Dynamical Reasoning Assembler (a project tangent to KAMI) [12]. INDRA provides a set of tools for reading biological facts expressed with natural language and other formats like BioPAX [10] and representing these facts with a set of computable statements. As was mentioned earlier, we also implement an importer which is able to convert INDRA statements to the respective KAMI data structures (`kami.importers.indra_to_kami`).

The KAMI data structures for entities and interactions of interest differ from the INDRA statements in a couple of crucial points. The first important difference is that INDRA statements cannot represent the protein domains 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 related to the interaction, which is in some cases does not coincide with such level of INDRA knowledge representation. For example, using the INDRA statement of the type `indra.statements.Complex` we can represent a formation of the protein complex A/B/C (from the respective proteins A, B, C). On the other hand, to be able to input this fact to KAMI we need to provide a direct binary binding mechanisms involved in the formation of this complex (e.g. A binds B, and B binds C). Therefore, a direct import into KAMI data structures from the input sources would prevent us from losing the pertinent information about mechanistic details of interactions.

The sub-module `kami.resolvers.generators` contains a set of tools for generating nuggets from the KAMI interaction objects. A generator takes as an input a particular interaction object, and outputs a nugget together with its typing by the meta-model and the action graph. It does it in three steps:

- build a nugget graph and type it by the meta-model;
- use *anatomizer* (the `kami.anatomizer` module) to identify genes, regions, residues and states;
- use semantic nuggets $\vec{SN}$ from the hierarchy to identify the interactions, and autocomplete the nugget with more mechanistic details if necessary.

The `kami.anatomizer` module implemented by our biologist collaborator Sebastien Legare encapsulates a set of client utilities for collecting information of interest from the biological databases. This tool plays a crucial role for identification of standardized gene and protein domains references which carry functional annotations.

To disambiguate and identify an interaction we need to be able to understand the mechanism that underlies it. For this purpose KAMI tries to relate an input nugget with a semantic nugget from $\vec{SN}$ as it was presented in 3.4. Once the relation was established, the nugget is checked to be semantically correct (otherwise discarded), then autocompleted, and finally its action is mapped to an existing action node in the action graph related to the corresponding semantic action. A small example of programmatic input of interactions into KAMI model is presented in the appendix E.

We have tested the resolver on a real-world dataset (*PID: the pathway interaction database* [14] converted to the INDRA statements from its initial BioPAX format) containing 1724 phosphorylation modifications with no domains specified. As the result, 1724 nuggets were created, 1322 of which where successfully autocompleted with the identified protein kinase region and related to the phosphorylation semantic nugget. Aggregation of this knowledge resulted in the action graph with 2312 nodes (including all the regions found for every gene by the anatomizer), among which only 95 were modification nodes. It means that all 1322 phosphorylation modification nodes in the nuggets (with the unique protein kinase region found) were identified and classified into 95 distinct phosphorylation activities. Among the top three modification nodes with the highest number of outgoing edges in the action graph (which represents distinct genes that can be phosphorylated), the modifications performed by the genes belonging to the *Mitogen-activated protein kinase* superfamily of proteins were found (MAPK3 phosphorylates products of 57 distinct genes, MAPK14 – 56, and MAPK1 – 54).

6 Conclusions

In this report I have presented the results of my M2 internship in the ongoing project dedicated to the development of a graph-based hierarchical knowledge representation system for rule-based modeling of cellular signaling. We are developing both a formalism and software tools implementing it (the `ReGraph` library). Despite the fact that it is inspired and driven by the problem of bio-curation in cellular signalling, we tried to make it as generic as possible and potentially applicable for modelling of graph-based systems in any domain.

Another direction of our work is focused on the developments of a concrete specification of this KR system for purposes of modelling of protein-protein interactions and a tool which would be able to accommodate and manage it (the `KAMI` platform). In addition, this tool allows coherent aggregation of fragmentary biological facts about individual protein-protein interactions and encapsulates a machinery for specific to cellular signalling automated reasoning about interaction mechanisms based on expert knowledge. KAMI aims to facilitate the modelling processes for biologists. At the same time time tries to solve a bio-curation problem, serves as a modelling environment and enables the accommodated knowledge to be directly ‘executed’ and analyzed with tools for rule-based modeling and simulations.

7 Future work

In the short-term future, there are several things should be implemented for the KAMI platform. First, we would like to implement a set of importers which would support biological formats such as BioPAX, and existing tools for signalling-specific natural language processing, such as TRIPS

[3] or REACH [15]. Potentially, we would also like to implement our own tools for processing of biochemical literature, which would be able to extract relevant knowledge with the high accuracy and the required level of mechanistic details (but it goes beyond our short-term goals).

Another crucial part of KAMI that requires a substantial amount of further work is the resolver. We would like to provide a better support for aggregation and autocompletion of nuggets, and mechanistic interpretation of phenomenological knowledge. For this purpose, we need to acquire more semantic nuggets, and at the moment, it remains a bottleneck as the acquisition and the hard-wiring of semantic nuggets in our system requires deep expert knowledge. However, the number of semantic nuggets covering most of the common binding and modification mechanisms is rather small and remains feasible for our application.

In the long term future, we would like to make KAMI not only a tool for 'smart' knowledge aggregation but a full-blown modelling environment, which provides means for visualization, sophisticated annotation, hypothesis testing and assertions. In addition, KAMI should be able to accommodate model variants, support the simultaneous development of multiple models and their automated merge (similar to version control systems).

Our goal is to build a powerful modelling tool for the biological community, which by automating the curation process would leave biologists room for experimenting, hypothesizing, discovering new knowledge and understanding complex signalling mechanisms.

Acknowledgements

I would like to express my gratitude to my supervisor Russ Harmer for his motivation, support, trust and guidance throughout the internship. I also would like to thank my collaborators Yves-Stan Le Cornec and Sébastien Legare for their help, valuable advice and fruitful discussions.

This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army Research Office under grant numbers W911NF-14-1-0367 and W911NF-15-1-0544. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] KaSim Development Homepage. `http://dev.executableknowledge.org/`.

[2] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. Molecular biology of the cell. new york: Garland science; 2002. *Classic textbook now in its 5th Edition*, 2010.

[3] J. Allen, W. de Beaumont, L. Galescu, and C. M. Teng. Complex event extraction using drum. *ACL-IJCNLP 2015*, page 1, 2015.

[4] S. Awodey. *Category theory*. Oxford University Press, 2010.

[5] A. Basso-Blandin, W. Fontana, and R. Harmer. A knowledge representation meta-model for rule-based modelling of signalling networks. *arXiv preprint arXiv:1603.01488*, 2016.

[6] 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.

[7] A. Corradini, T. Heindel, F. Hermann, and B. König. Sesqui-pushout rewriting. In *International Conference on Graph Transformation*, pages 30–45. Springer, 2006.

[8] 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.

[9] 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.

[10] E. Demir, M. P. Cary, S. Paley, K. Fukuda, C. Lemer, I. Vastrik, G. Wu, P. D'Eustachio, C. Schaefer, J. Luciano, et al. Biopax–a community standard for pathway data sharing. *Nature biotechnology*, 28(9):935, 2010.

[11] D. T. Gillespie. Stochastic simulation of chemical kinetics. *Annu. Rev. Phys. Chem.*, 58:35–55, 2007.

[12] 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. *bioRxiv*, page 119834, 2017.

[13] A. Moreno, K. Ruiz-Mirazo, and X. Barandiaran. The impact of the paradigm of complexity on the foundational frameworks of biology and cognitive science. *Handbook of the philosophy of science*, pages 311–333, 2011.

[14] C. F. Schaefer, K. Anthony, S. Krupa, J. Buchoff, M. Day, T. Hannay, and K. H. Buetow. Pid: the pathway interaction database. *Nucleic acids research*, 37(suppl 1):D674–D679, 2009.

[15] M. A. Valenzuela-Escárcega, G. Hahn-Powell, and M. Surdeanu. Description of the odin event extraction framework and rule language. *arXiv preprint arXiv:1509.07513*, 2015.

Appendix A Sesqui-pushout rewriting

Sesqui-pushout (SqPO) rewriting is an approach for abstract deterministic rewriting in any category with pushouts and all pullback complements over monos [7]. We denote informally rewriting of a graph G as $G \rightsquigarrow G'$, where G' is a result of rewriting. In our context, SqPO allows to perform the following operations on graphs: addition/deletion of the nodes, addition/deletion of the edges, cloning and merging of the nodes.

Any sequence of these operations can be expressed with a rewriting rule which is defined by a span $L \leftarrow P \rightarrow R$, where L represents a pattern that is going to be matched in G. An instance of a rule is a matching morphism $L \rightarrowtail G$, SqPO procedure performs rewriting of the instance in two steps:

1. Perform all the deletions and cloning by constructing a graph G^- as a final pullback complement from $P \rightarrow L \rightarrowtail G$.
2. Build the graph G' as a pushout from the span $G^- \leftarrow P \rightarrow R$ containing all the nodes and edges added by the rule as well as merges of the nodes.

The diagram below illustrates the constructs used in the rewriting procedure from G to G':

$$\begin{tikzcd}
L \arrow[d, tail] & P \arrow[l, "h_1"] \arrow[r, "h_2"] \arrow[d, tail] & R \arrow[d, tail] \\
G \arrow[rr, squiggly, bend right] & G^- \arrow[l] \arrow[r] & G'
\end{tikzcd}$$

Informally application of a rule $L \leftarrow P \rightarrow R$ can be reformulated as follows (in this canonical order):

1. **Deletion of nodes:** if node n^L in L does not have a preimage in $h_1 : P \rightarrow L$, node in G mapped by n^L in $L \rightarrowtail G$ is going to be deleted by the rewriting (together with all its adjacent edges as a side effect).
2. **Deletion of edges:** for $h_1 : P \rightarrow L$, if $n_1^L = h_1(n_1^P)$, and $n_2^L = h_1(n_2^P)$, and there is an edge between n_1^L and n_2^L in L, and there is no edge between n_1^P and n_2^P in P, the edge in G between nodes mapped by n_1^L and n_2^L in $L \rightarrowtail G$ will be deleted.
3. **Cloning of nodes:** if two nodes n_1^P, n_2^P of P map to the same node n^L in L, node in G mapped by n^L in $L \rightarrowtail G$ is going to be cloned during graph rewriting.
4. **Merging of nodes:** if two nodes of n_1^P and n_2^P in P match to the same node n^R in R, corresponding nodes in G are merged.
5. **Addition of nodes:** if node n^R in R does not have a preimage in $h_2 : P \rightarrow R$, n^R is going to be added by the rewriting.
6. **Addition of edges:** for $h_2 : P \rightarrow R$, if $n_1^R = h_2(n_1^P)$, and $n_2^R = h_2(n_2^P)$, and there is an edge between n_1^R and n_2^R in R, and there is no edge between n_1^P and n_2^P in P, the edge between corresponding nodes in G will be added.

A rule $L \leftarrow P \rightarrow R$ is *typed* by T iff L, P and R are typed by T s.t.

$$\begin{tikzcd}
L \arrow[dr] \arrow[r, phantom, shift right=6ex, "="] & P \arrow[l] \arrow[r] \arrow[d] \arrow[r, phantom, shift right=6ex, "="] & R \arrow[dl] \\
& T &
\end{tikzcd}$$

The result of rewriting $G \rightsquigarrow G'$ with a rule $r : L \leftarrow P \rightarrow R$, where G and r are typed by T, is also typed by T. The following proposition illustrates this fact.

Proposition A.1. *There exists a unique typing $f' : G' \rightarrow T$, where G' is the result of $G \rightsquigarrow G'$ sesqui-pushout rewriting of a graph G with a rule $L \overset{h_1}{\leftarrow} P \overset{h_2}{\rightarrow} R$ with $f_R : R \rightarrow T$ (typing of R by T) on an instance $m_L : L \rightarrowtail G$.*

Proof. In the first step of rewriting graph G^- is obtained from the final pullback complement from $P \rightarrow L \rightarrowtail G$. G^- is typed by T by the homomorphism $f \circ g_1$ as illustrated on the diagram. In the second step of rewriting, G' is obtained with the pushout from the span $G^- \leftarrowtail P \rightarrow R$, and there exists a unique homomorphism $f' : G' \rightarrow T$, which follows from the universal property of the pushout and the fact that $f \circ g_1 \circ m_P = f_R \circ h_2$, as it is illustrated on the diagram. □

$$\begin{tikzcd}
L \arrow[d, tail, "m_L"] & P \arrow[l, "h_1"'] \arrow[r, "h_2"] \arrow[d, tail, "m_P"] & R \arrow[d, tail, "m_R"'] \arrow[ddl, bend left=60, "f_R"] \\
G \arrow[dr, "f"] & G^- \arrow[l, "g_1"] \arrow[r, "g_2"] & G' \arrow[dl, dashed, red, "f'"] \\
& T &
\end{tikzcd}$$

Appendix B Graph hierarchy and partial homomorphisms

Partial graph homomorphisms

To make our knowledge representation system more flexible we allow partial typing of graphs defined by a partial homomorphism $A \rightharpoonup B$ in a graph hierarchy. Intuitively, partial typing does not require for all nodes of A to be typed by nodes in B.

Definition B.1. Partial homomorphism $f : A \rightharpoonup B$ *approximates* $g : A \rightharpoonup B$ ($f \leq g$) iff for all $x \in A$, if $f(x)$ is defined and $f(x) = y$, then $g(x)$ is also defined and $g(x) = y$.

Definition B.2. Partial homomorphisms $f : A \rightharpoonup B$ and $g : A \rightharpoonup B$ are *"Kleene equal"* ($f \simeq g$) iff $f \leq g$ and $g \leq f$.

Our category of $\mathbf{Graphs}_{attrs}$ is a category with pullbacks, therefore a partial homomorphism $f : A \rightharpoonup B$ can be defined as the span $A \leftarrowtail A_B \to B$, where A_B is the domain of f and composition is defined by pullback. Note that a total homomorphism is a special case of a partial one, namely a homomorphism $f : A \to B$ can be expressed as the span $A \overset{id}{\leftarrowtail} A \to B$.

In terms of spans the fact that $f : A \rightharpoonup B$ approximates $g : A \rightharpoonup B$ can be reformulated as follows. $f \leq g$ iff there exists a monomorphism u such that $f_1 = g_1 \circ u$ and $f_2 = g_2 \circ u$, as shown on the diagram below:

$$\begin{tikzcd}
& A \arrow[dd, harpoon, bend right=80, red, "f"'] \arrow[dd, harpoon, bend left=80, red, "g"] & \\
A^f_B \arrow[ur, tail, "f_1"] \arrow[rr, tail, "u"] \arrow[dr, "f_2"'] & & A^g_B \arrow[ul, tail, "g_1"'] \arrow[dl, "g_2"] \\
& B &
\end{tikzcd}$$

In a hierarchy with partial typing we require that any two homomorphisms f and g constructed with the composition of the homomorphisms along the paths from A to B to be "Kleene equal":

$$\begin{tikzcd}
& A \arrow[dl, harpoon, bend right, "h_1"'] \arrow[dr, harpoon, bend left, "i_1"] \arrow[ddd, harpoon, bend right=30, red, "f"'] \arrow[ddd, harpoon, bend left=30, red, "g"] \arrow[ddd, phantom, "\simeq"] & \\
\cdot \arrow[d, harpoon, "h_2"'] & & \cdot \arrow[d, harpoon, "i_2"] \\
\cdots \arrow[dr, harpoon, bend right, "h_n"'] & & \cdots \arrow[dl, harpoon, bend left, "i_m"] \\
& B &
\end{tikzcd}$$

Propagation to partially typed graphs

Recall in a graph hierarchy we would like to propagate the result of rewriting $G \rightsquigarrow G'$ to all the graphs typed by G. Previously we have considered the procedure of propagation in case of total typing. If some graph H_k in a hierarchy is partially typed by G ($H_k \rightharpoonup G$) the propagation of changes from G to H_k is performed similarly with the pullback from $H_k \rightharpoonup G \leftarrow G^-$.

$$\begin{tikzcd}
H_k \arrow[d] & H_k^- \arrow[l] \arrow[d] \\
G & G^- \arrow[l]
\end{tikzcd}$$

Having two partial homomorphisms $f : A \rightharpoonup C$ and $g : B \rightharpoonup C$ the pullback is given by two morphisms $x : A \times^p_C B \rightharpoonup A$ and $y : A \times^p_C B \rightharpoonup B$ such that $f \circ x = g \circ y$ and for every Z and $z_1 : Z \rightharpoonup A$ and $z_2 : Z \rightharpoonup B$ with $f \circ z_1 = g \circ z_2$ there exists a unique partial morphism $u : Z \rightharpoonup A \times^p_C B$.

$$\begin{tikzcd}
 & Z \arrow[d, dashed, "u"] \arrow[ddl, bend right, "z_1"'] \arrow[ddr, bend left, "z_2"] & \\
 & A \times^p_C B \arrow[dl] \arrow[dr] & \\
A \arrow[dr, "f"] & & B \arrow[dl, "g"] \\
 & C &
\end{tikzcd}$$

Namely, for graphs with attributes and their partial homomorphisms the pullback graph $A \times^p_C B$ is given by the following set of nodes:

$$\begin{aligned} A \times^p_C B = & \{\ (a,b) \in A \times B \mid f(a) \text{ and } g(b) \text{ are defined and } f(a) = g(b)\ \} \cup \\ & \{\ a \in A \mid f(a) \text{ is not defined }\}\ \cup\ \{\ b \in B \mid g(b) \text{ is not defined }\} \cup \\ & \{\ (a,b) \mid f(a) \text{ and } g(b) \text{ are not defined }\} \end{aligned}$$

However, for the purposes of propagation we are interested in a special case when one homomorphism, say g, is total ($g : B \rightarrow C$). In this case a set of nodes in the $A \times^p_C B$ graph is given by:

$$\begin{aligned} A \times^p_C B = & \{\ (a,b) \in A \times B \mid f(a) \text{ and } g(b) \text{ are defined and } f(a) = g(b)\ \} \cup \\ & \{\ a \in A \mid f(a) \text{ is not defined }\} \end{aligned}$$

Remember that the partial homomorphism $f : A \rightharpoonup C$ can be represented as the span $A \leftarrowtail A_C \rightarrow C$. We can construct the pullback $A \overset{x}{\leftarrow} A \times^p_C B \overset{y}{\rightarrow} B$ in the following two-steps procedure (with the corresponding diagram below): (1) find the pullback from the cospan $A_C \rightarrow C \leftarrow B$ (corresponding pullback graph is denoted with a star $*$ on the diagram), (2) find the final pullback complement from $* \rightarrow A_C \rightarrowtail A$. We obtain $A \times^p_C B$, $x : A \times^p_C B \rightarrow A$ and $y : A \times^p_C B \rightharpoonup B$ as the result of this procedure.

$$\begin{tikzcd}
A \arrow[dd, bend right, "f"'] & A \times^p_C B \arrow[l, "x"'] \arrow[dd, bend left, "y"] \\
A_C \arrow[u, tail] \arrow[d] & * \arrow[l] \arrow[u, tail] \arrow[d] \\
C & B \arrow[l, "g"]
\end{tikzcd}$$

Appendix C Propagation procedure

```
Data: $G, G^-, G^- \to G$
Result: $(V, E)$, where $V$ is a set of updated graphs, $E$ is a set of updated typing
        homomorphisms
$V := \{G^-\}$;
$E := \emptyset$;
visited := $\emptyset$;
current_level := predecessors(G);
while current_level $\neq \emptyset$ do
    next_level := $\emptyset$;
    for $H \in$ current_level do
        visited := visited $\cup \{H\}$;
        $H \to G$ := compose homomorphisms along the path from H to G;
        $H^-, H^- \to H, H^- \to G$ := find the pullback from $H \leftarrow G \to G^-$;
        $V := V \cup \{H^-\}$;
        for $S \in$ successors(H) do
            if $S \in$ visited then
                find a unique homomorphism $H^- \to S$ using the fact that $S \leftarrow S^- \to G-$ is the
                  pullback from $S \to G \leftarrow G^-$;
                $E := E \cup \{H^- \to S\}$;
            else
                $H^- \to S$ := compose $H^- \to H$ and $H \to S$;
                $E := E \cup \{H^- \to S\}$;
            end
        end
        for $P \in$ predecessors(H) do
            if $P \in$ visited then
                find a unique homomorphism $P^- \to H^-$ using the fact that $H \leftarrow H^- \to G^-$ is
                  the pullback from $H \to G \leftarrow G^-$;
                $E := E \cup \{P^- \to H^-\}$;
            else
        end
        next_level := next_level $\cup \{P \in$ predecessors(H) $\mid P \notin$ visited $\}$;
    end
    current_level := next_level;
end
```

Algorithm 1: Propagation procedure

Appendix D ReGraph examples

D.1 Simple graph rewriting with ReGraph

Graph rewriting is implemented as an application of a *graph rewriting rule* to a given input graph object G, the rule is defined by a span $L \leftarrow P \rightarrow R$ (more details can be found in appendix A). The library provides a data structure `regraph.rules.Rule` (we will refer to it as `Rule`) for definition of rewriting rules. In addition, `ReGraph` provides several options for the rule definition: (a) L, P and R can be defined as `NetworkX` graph objects and passed to the constructor of `Rule` together with valid homomorphisms $P \rightarrow L$ and $P \rightarrow R$; (b) a rule can be constructed from L and a sequence of primitive operations on L, for example:

```python
import NetworkX as nx
from regraph.rules import Rule

# Initialize a pattern
edge_list = [(1, 2), (3, 2), (3, 4)]
L = nx.DiGraph(edge_list)

# Initialize a rule with a pattern
rule = Rule.from_transform(L)

# Perform operations on the rule
rule.remove_edge(3, 2)
rule.clone_node(1)
rule.add_node(5)
rule.add_edge(5, 3)
rule.add_edge(2, 4)
```

This code creates a rule visualized on the figure below.

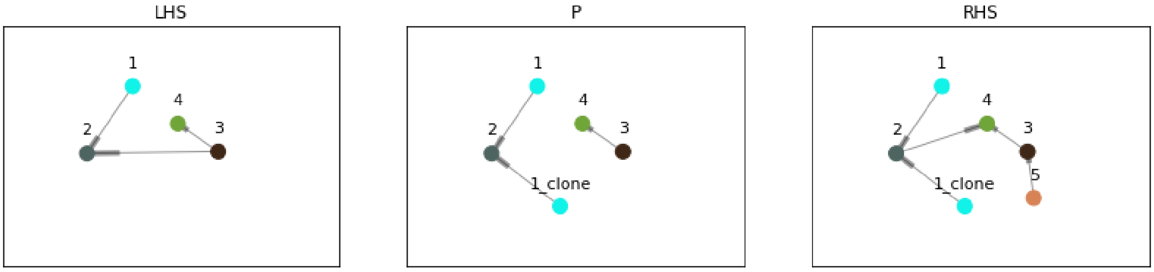
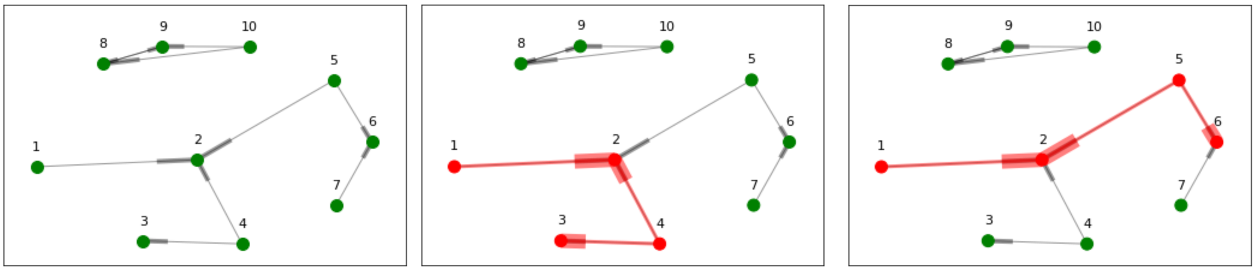


Figure 12: *Example of a rewriting rule (output of* `regraph.plotting.plot_rule`*)*: illustrated rule deletes an edge between nodes corresponding to the nodes ‘4’ and ‘3’ in the left-hand side of the rule, clones a node corresponding to the node ‘1’, adds a new node (labeled ‘5’ on the figure) together with a new edge to the node corresponding to ‘3’, and, finally, adds a new edge between the nodes corresponding to ‘2’ and ‘4’.

Matching of a rule

Matching of L in G is defined by a homomorphism $L \rightarrowtail G$. A pattern graph L may have several instances of such matching in a graph G. `ReGraph` provides a function `regraph.primitives.find_matching` that finds all the instances of L. This function returns a list whose elements are dictionaries representing instances of the matching. Figure below illustrates instances of the matching of L from the previous example found in a graph from the subfigure a.



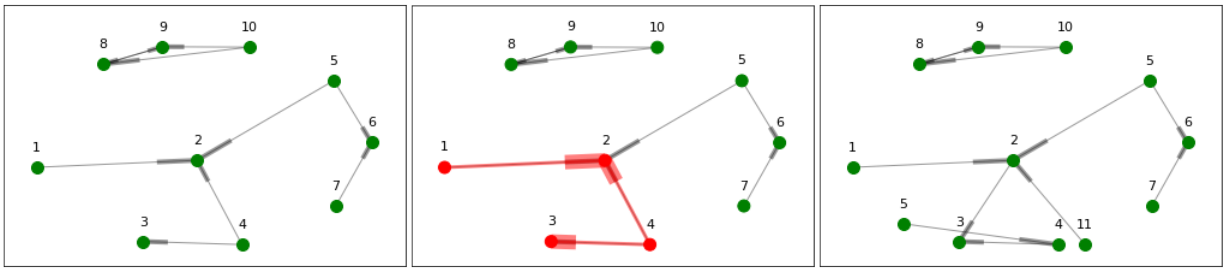
(a) Input graph G

(b) Instances of matching $L \rightarrowtail G$

Figure 13: Example of a matching in a graph with several instances (output of `regraph.plotting.plot_instance` function)

Rewriting procedure

Graph rewriting is performed with the `regraph.primitives.rewrite` function. It takes as an input a graph, an instance of the matching (dictionary that defines $L \rightarrowtail G$) and a rewriting rule (an instance of `Rule`).



(a) Input graph G

(b) Instance of matching

(c) Resulting graph

Figure 14: Result of rewriting of the graph G with the previously defined rule

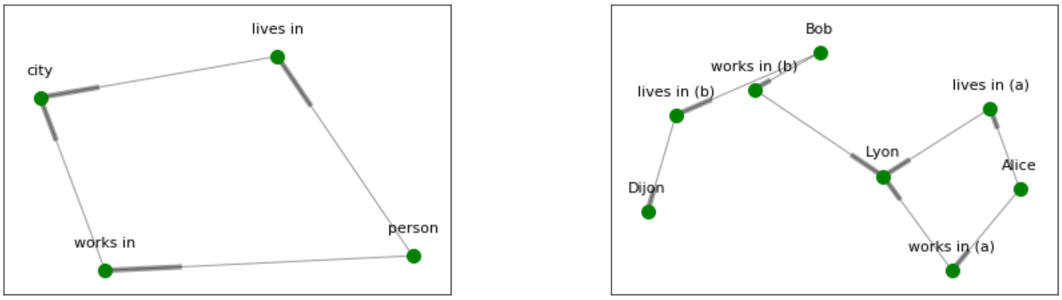
`ReGraph` allows to perform rewriting of a graph object both in-place and by returning a new object corresponding to the result of rewriting.

D.2 Graph hierarchy and rewriting in the hierarchy

Initialization of a hierarchy

Consider the following example of a simple graph hierarchy (related to the example from the figure 1). The two graphs M and MM are being created and added to the hierarchy. Afterwards a typing homomorphism between M and MM is added, so that every node of G is typed by some node in T.

```python
# Initialize graphs
mm = nx.DiGraph()
mm.add_edges_from([
    ("person", "works in"), ("person", "lives in"),
    ("works in", "city"), ("lives in", "city")
])

m = nx.DiGraph()
m.add_edges_from([
    ("Alice", "lives in (a)"),
    ("lives in (a)", "Lyon"),
    ("Alice", "works in (a)"),
    ("works in (a)", "Lyon"),
    ("Bob", "lives in (b)"),
    ("lives in (b)", "Dijon"),
    ("Bob", "works in (b)"),
    ("works in (b)", "Lyon")
])

hierarchy = Hierarchy()
hierarchy.add_graph("MM", mm)
hierarchy.add_graph("M", m)
hierarchy.add_typing(
    "M", "MM",
    {"Alice": "person",
     "Bob": "person",
     "Lyon": "city",
     "Dijon": "city",
     "lives in (a)": "lives in",
     "works in (a)": "works in",
     "lives in (b)": "lives in",
     "works in (b)": "works in"},
    total=True, ignore_attrs=True
)
```



(a) Graph MM

(b) Graph M

Figure 15: Plots illustrating graphs M and MM from the constructed hierarchy.

Type-respecting rewriting

Let us now define the following rewriting rule:

```python
# Initialize the left-hand side
lhs = nx.DiGraph()
lhs.add_edges_from([(1, 2), (2, 3)])

# Initialize the preserved part
p = nx.DiGraph()
p.add_nodes_from([1, 2, 3])
p.add_edges_from([(1, 2)])

# Initialize the right-hand side
rhs = nx.DiGraph()
rhs.add_nodes_from([1, 2, 3, "Paris"])
rhs.add_edges_from([(1, 2), (2, "Paris")])

# By default if `p_lhs` and `p_rhs` are not provided
# to a rule, it tries to construct this homomorphisms
# automatically by matching the names. In this case we
# have defined lhs, p and rhs in such a way that that
# the names of the matching nodes correspond
rule_1 = Rule(p, lhs, rhs)

lhs_typing = {
    "MM": {
        1: "person",
        2: "works in",
        3: "city"
    }
}
rhs_typing = {
    "MM": {
        "Paris": "city"
    }
}
```

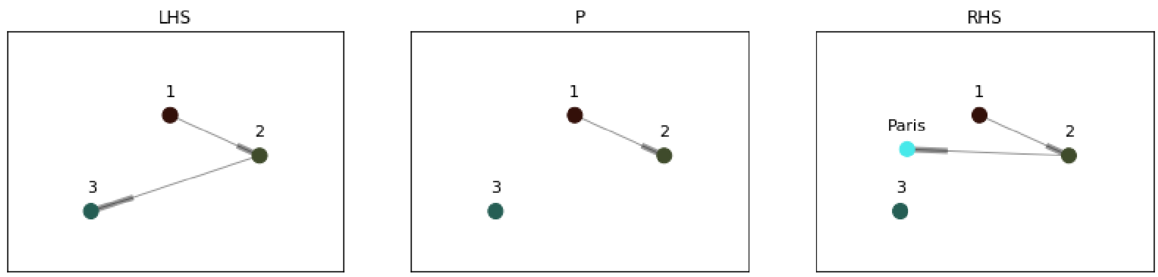
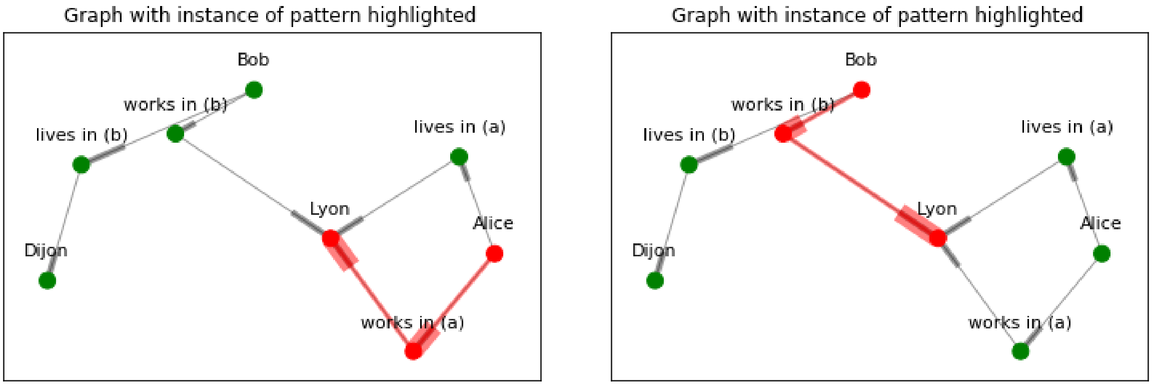


Figure 16: Rewriting rule defined in the previous listing (output of `regraph.plotting.plot_rule`)

Using this rule we want to change the city where Alice works to Paris (as in the example from 2.2). The following code illustrates how it can be done using `ReGraph`:

```python
# Find matching of lhs with lhs_typing
instances = hierarchy.find_matching("M", lhs, lhs_typing)

for instance in instances:
    plot_instance(hierarchy.node["M"].graph, lhs, instance, pos=m_pos)
```

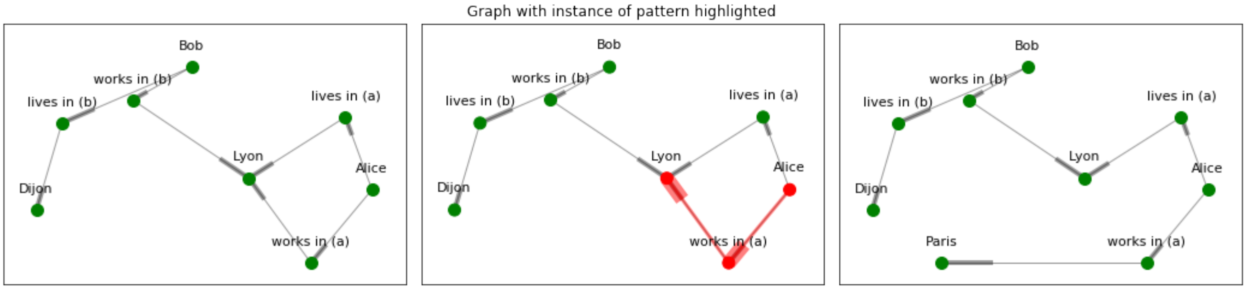


(a) (b)

Figure 17: Instances of the previously defined rule

```python
    hierarchy.rewrite(
        "M", rule_1, instances[0], lhs_typing, rhs_typing,
        total=True, inplace=True
    )
```

The figure 18 illustrated the original graph M, the instance that we want to rewrite and the rewriting result.



(a) Original graph M (b) Selected instance (c) Result of rewriting

Figure 18

Rewriting and propagation

Now consider another example of the meta-model rewriting and propagation of the changes to the graph M. In the listing below we define a rule that clones the node ‘person’ in the meta-model and creates two nodes ‘inhabitant’ and ‘employee’ connecting the first one to the node ‘lives in’ and the second one to the node ‘works in’.

```
lhs = nx.DiGraph()
lhs.add_edges_from([("person", "lives in"), ("person", "works in")])

p = nx.DiGraph()
p.add_edges_from([("employee", "works in"), ("inhabitant", "lives in")])

# Right-hand side is identical to the preserved part
rhs = copy.deepcopy(p)
p_lhs = {
    "employee": "person", "inhabitant": "person",
    "works in": "works in", "lives in": "lives in",
}

rule_2 = Rule(p, lhs, rhs, p_lhs)
```

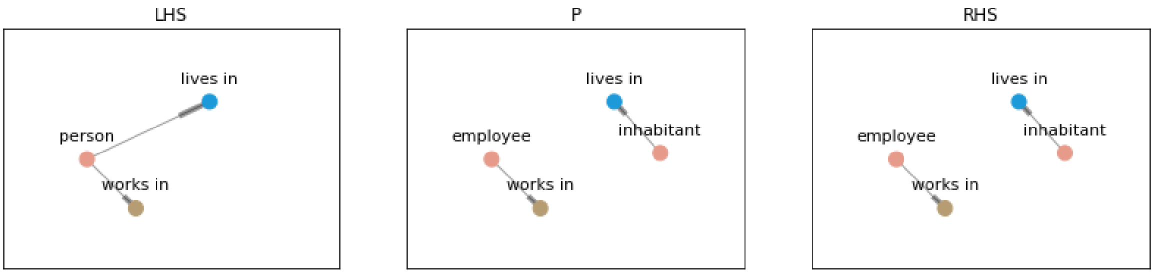
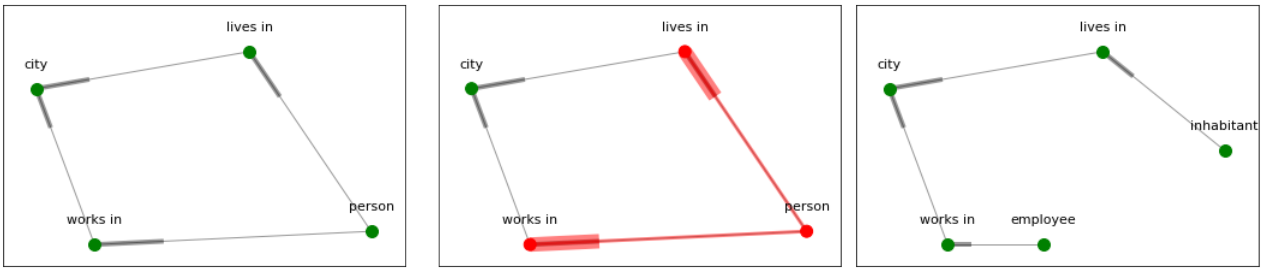


Figure 19: Rewriting rule defined in the previous listing (output of `regraph.plotting.plot_rule`)

Similarly, we search for the matching of the left-hand side of the rule in the graph MM and with the selected instance we rewrite MM in the hierarchy.

```
instances = hierarchy.find_matching("MM", lhs)
hierarchy.rewrite("MM", rule_2, instances[0], total=True, inplace=True)
```



(a) Original graph MM (b) Selected instance (c) Result of rewriting

As the result of rewriting of MM in the hierarchy graph the respective changes are propagated to M:

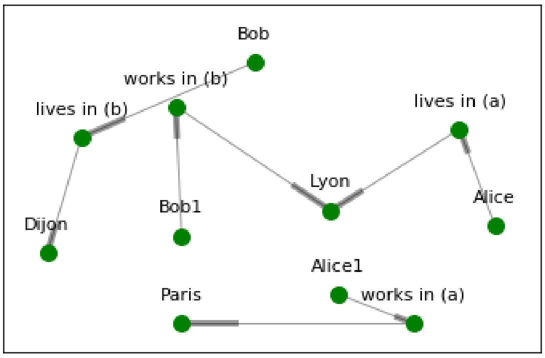


Figure 21: M after propagation of changes in MM

The following code:

```
print("Alice - type: ", hierarchy.node_type("M", "Alice")["MM"])
print("Alice1 - type: ", hierarchy.node_type("M", "Alice1")["MM"])
print("Bob - type: ", hierarchy.node_type("M", "Bob")["MM"])
print("Bob1 - type: ", hierarchy.node_type("M", "Bob1")["MM"])
```

outputs:

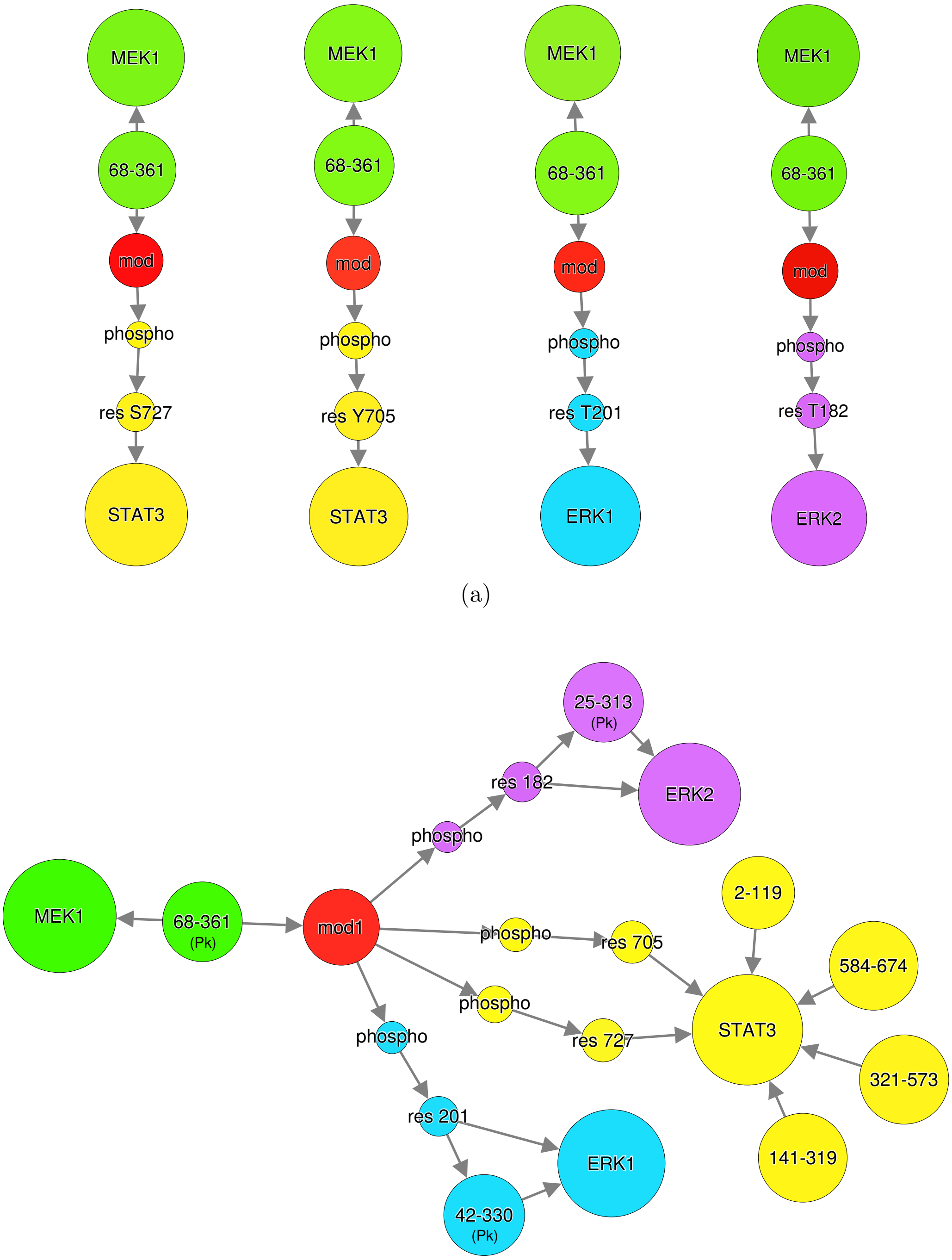
```
Alice - type: inhabitant
Alice1 - type: employee
Bob - type: inhabitant
Bob1 - type: employee
```

Appendix E Example of programmatic KAMI usage

The following listing contains an example of usage of KAMI API for programmatic input of interactions to the hierarchy.

```python
from kami.data_structures.entities import (Agent, Region,
                                           PhysicalAgent,
                                           Residue, State,
                                           PhysicalRegion,
                                           PhysicalRegionAgent)
from kami.data_structures.interactions import Modification
from kami.resolvers.black_box import create_nuggets

# Initialize an empty hierarchy
hierarchy = KamiHierarchy()

# Define interactions
mek1 = PhysicalAgent(Agent("Q02750"))
stat3 = PhysicalAgent(Agent("P40763"))
mod_state_1 = Residue("S", 727, State("phosphorylation", False))
mod1 = Modification(mek1, stat3, mod_state_1, value=True, direct=True)

mod_state_2 = Residue("Y", 705, State("phosphorylation", False))
mod2 = Modification(mek1, stat3, mod_state_2, value=True, direct=True)

erk1 = PhysicalAgent(Agent("P27361"))
mod_state_3 = Residue("T", 201, State("phosphorylation", False))
mod3 = Modification(mek1, erk1, mod_state_3, value=True, direct=True)

erk2 = PhysicalAgent(Agent("P28482"))
mod_state_4 = Residue("T", 182, State("phosphorylation", False))
mod4 = Modification(mek1, erk2, mod_state_4, value=True, direct=True)

interactions = [mod1, mod2, mod3, mod4]

# Create nuggets from the interactions
create_nuggets(
    interactions,
    hierarchy,
    add_agents=True,
    anatomize=True
)
```

In this example genes (instances of `kami.data_structures.entities.Agent`) are defined by their UniProt accession number. The resolver module of KAMI creates nuggets corresponding to these interactions. Moreover, using the agent anatomizer it adds new nodes corresponding to the genes in the action graph together with their domains. The resolver also identifies that all the interactions are phosphorylations and that the enzyme (with identifier Q02750) contains a protein kinase domain, therefore all the modification nodes in the nuggets are mapped to a single modification action in the action graph, which represents the addition of phosphate group to the substrate by the protein kinase domain. The following figure illustrates the resulting nuggets and the corresponding action graph:



(a)

(b)

Figure 22: Visualization of (a) the nuggets and (b) the action graph produced in the listing (plotted with *Gephi*). Regions are represented with the nodes containing integer intervals, we also denote with '(Pk)' protein kinase domains.