ALIFE 2025 MOD Tutorial

Optimisation and Exploration: Navigating Chemical Reaction Networks with MOD, a Graph-based Framework

Anne-Susann Abel, Yingjie Pan, Johannes Petersen, Mikkel Boger Posselt, Joseph Antony Smith, Manuel Uhlir

Theoretical Biochemistry Institute
Department of Theoretical Chemistry
University of Vienna
Austria

1 Preamble

1.1 Web Resources

Before starting, here are some online resources that you might find helpful during or after this workshop, should you wish to revisit this document.

Source code: https://github.com/jakobandersen/mod/

Documentation: https://jakobandersen.github.io/mod/

Live playground: https://cheminf.imada.sdu.dk/mod/

MOD Tutorial Website: https://sites.google.com/view/alife-mod-tutorial/start

1.2 Installation Guide

To actively partake in this tutorial, we recommend downloading and installing MOD via a Docker image (instructions available on the tutorial website linked above). This is not absolutely necessary, of course, if you only wish to observe and listen.

1.3 The Formose Chemistry

In this tutorial, we will explore the formose reaction as an example network. This reaction network is considered a possible prebiotic pathway for the synthesis of a variety of sugars.

The pathway begins with formaldehyde (CH2O) condensing into progressively longer carbon chains under basic conditions. These elongated chains start to catalyse the production of more long-chain sugars, creating an autocatalytic cycle. This yields a final product consisting of various sugar products such as variations of ribose and glucose.

2 Introducing Chemistry as Graphs

2.1 Writing Molecules as SMILES Strings

Our first MOD task in this tutorial will be defining glycoaldehyde as a SMILES string. A SMILES string is a computer-readable method of writing chemical structures, using common keyboard characters to depict a molecule using ASCII to allow MOD to process even complex molecular structures.

SMILES is a simple system. **Atoms** are represented by their atomic symbol enclosed in square brackets, []. The following elements, however, can be written without the use of square brackets: B, C, N, O, P, S, F, Cl, Br, and I. Implicit hydrogen atoms can be completely omitted. **Bonds** are represented via -, =, and #, which denote single, double, and triple bonds, respectively. **Branches** can be added to a chain by enclosing the entire branch in round brackets, (). **Cycles** are described by labelling the initial/first atom in the ring with a numerical value (usually 1) and finishing the ring with the same number. For example, cyclohexane would have the following SMILES string: C1CCCCC1. The 1 indicates that the two carbon atoms are directly connected to each other.

Writing a SMILES string in MOD is simple, create a variable and use the MOD function smiles(). The SMILES string itself is confined within “”, and an optional name can be added (this is helpful when reading the derivation graph). Below is a snippet of example code for writing glycolaldehyde as a SMILES string.

```
# Writing a simple SMILES string in MOD
glycolaldehyde = smiles("OCC=O", name="Glycolaldehyde")

glycolaldehyde.print()
```

As mentioned above, the implicit hydrogens have been omitted; however, the result would be the same if they were explicitly included within the SMILES string. Alternatively, graphDFS is also a capable alternative to SMILES.

```
# Writing a simple graphDFS in MOD
Glycolaldehyde = graphDFS("[O]([C]([C](=[O])[H])([H])[H])[H]", name="Glycolaldehyde")
```

```
3
4 glycolaldehyde.print()
```

Within MOD, SMILES strings/graphDFS are processed as graphs. But what is a graph? A graph, in this case, is not the classic x-y plots often associated with the word, but a visual depiction of the relationship between data points. There are many types of graphs, but for now, we will work with the simplest form. These simple graphs only contain data about the identity of the points or nodes within the graph. In the case of MOD, atoms are represented as nodes, and the bonds as edges.

INSERT NICE EXAMPLE HERE

This ties us in nicely to the next section.

2.2 Writing Reactions as Graph Rewrite Rules

So now that we have a basic understanding of a chemical molecule as a graph, how can we transform one graph into another, like a molecule changing during a chemical reaction? This is where rules come in. A rewrite rule, is a graph transform that can reorganise the location of edges (bonds) and the information of a node (the atom) by changing the character (atomic symbol)

A rule is comprised of 3 sections. The left (L), context (K), and right (R). For the rule to be applied to the molecule, both the left and the molecule graph must contain the same nodes and edges. However, the left does not need to contain all of the same nodes, only a portion (subgraph/functional group), as long as they are in the same order/structure (isomorphic), the rule will apply. This tuneable aspect to the specificity of a rule is a key reason for the pathway discovery aspect MOD can provide.

INSERT ANOTHER NICE EXAMPLE HERE

Rules are written in the GML (Graph Modelling Language) syntax that uses a key-value pairing system. In this use case, keys are strings that correspond to the "left", "context", and "right" sides of the rule in the top hierarchy, while "node" and "edge" correspond to atoms and bonds in the context of a molecule.

Below is a sample of code for a rule describing a keto-enol isomerisation reaction. The rule contains an ID, effectively a name, helpful for navigating rules in the printed PDF document, should you choose to print them.

```
# String containing a GML rule representing keto-enol isomerisation
ketoEnolGML = """rule [
    ruleID "Keto-enol isomerisation"
    left [
        # edge defines a bond, and must have a start (source) and end (target)
        # the label defines the type of bond
        edge [ source 1 target 4 label "-" ]
        edge [ source 1 target 2 label "-" ]
        edge [ source 2 target 3 label "=" ]
    ]
    context [
        # a node is an atom, label corresponds to the atomic symbol
        node [ id 1 label "C" ]
        node [ id 2 label "C" ]
        node [ id 3 label "O" ]
        node [ id 4 label "H" ]
    ]
    right [
        edge [ source 1 target 2 label "=" ]
        edge [ source 2 target 3 label "-" ]
        edge [ source 3 target 4 label "-" ]
    ]
]"""

ketoEnolGML.print()
```

2.3 Derivation Graphs

Once we have a set of rewrite rules encoding possible reactions and the initial molecules, we can begin applying the rules to create a derivation graph. A derivation graph differs from the graphs that depict singular molecules. This is now a hypergraph so there is a direction associated to the edges within the graph. A derivation graph depicts the entire reaction network with nodes representing full molecules and edges representing the reactions. A hyperedge also includes data that informs the user which rule(s) were applied to create the resulting hyperedge and products.

SMALL DG EXAMPLE SHOWING RULE USE AND HYPEREDGE

To apply the rules to our graphs, we need to load them into the environment. If you have been following along with the tutorial, you should have all the necessary rules and molecules already written out in your Python file.

```
# We first load the rules
aldolAdd_F = ruleGMLString(aldolAddGML)
aldolAdd_B = ruleGMLString(aldolAddGML, invert=True)
ketoEnol_F = ruleGMLString(ketoEnolGML)
ketoEnol_B = ruleGMLString(ketoEnolGML, invert=True)

# Then we define our input molecules
formaldehyde = smiles("C=O", name="Formaldehyde")
glycolaldehyde = smiles( "OCC=O", name="Glycolaldehyde")

# Followed by creating a list of molecules named input_molecules
input_molecules = [formaldehyde, glycolaldehyde]

# Now we can produce a derivation graph from these rules and molecules
# Define the DG object and the initial molecules in the reaction network
dg = DG(graphDatabase=input_molecules)
reaction_network = dg.build()

# Deleting the reaction_network locks the dg so it can be printed
del reaction_network
dg.print()
```

3 MOD, the Basics

Now that we have become a bit more accustomed to working with chemistry as graphs, we can begin to expand upon these principles. These next capabilities of MOD allow us to improve the results of the derivation graph by supplying further control over how the rules are applied.

3.1 Strategies

The expansion of a derivation graph can be controlled with a strategy. A strategy allows the user to define operators that impact how the graph is expanded. These operations can vary from the number of repeats (how many times MOD applies the rules), which rules and when, and predicates. The number of repeats can drastically change the output of the derivation graph; feel free to play around with the number of times you allow MOD to apply the rules within the strategy. Within the Python file, we can have multiple lists of input molecules. This allows us to create an initial expansion with one set of molecules and then expand upon it again with another.

```
# This time, we use the execute() function
strategy = (
    addSubset(input_molecules)
        >> repeat[4](inputRules)
)

reaction_network.execute(strategy)

del reaction_network
dg.print()
```

3.2 Predicates

Another method that has a large impact on the final graph are predicates. A predicate can be applied to either side of the rule (left or right), but it works by effectively adding further constraints to how/when a rule can be used, allowing the user to narrow down the applicability of potentially promiscuous rules.

A left predicate can restrict which molecules the rule consumes. For example, in a rule that is written to describe an enzymatic reaction that is applied to only acyclic secondary alcohols, in order to apply the rule to any secondary alcohol, only the subgroup needs to be defined within the rule. However, this rule would also apply to a cyclic secondary alcohol, which this enzyme does not catalyse according to the information available in literature. This is where a left predicate can be used, commanding the rule to specifically avoid defined carbon rings.

```
# Define the pattern we do not wish to see as a SMILES string or graphDFS
cycle = smiles("[C]1[C][C][C][C][C]1")

# Define the strategy and include initial molecules
strategy = (
    addSubset(input_molecules)
        >> leftPredicate[lambda derivation: all(num_subgraphs(g, cycle)
        < 1 for g in derivation.left)]
        (repeat[10](inputRules)
        )
)

reaction_network.execute(strategy)
del reaction_network
dg.print()
```

We can also restrict whether a rule is applied or not based on the number of certain atoms within the molecules on a side of a rule. We do this by querying the number of vertices of a certain symbol that are present in the graphs the rule is applied to by using the graph property vLabelCount.

```
# Only allow reactions where the reactants together have at most 4 oxygen atoms
strategy = (
    addSubset(input_molecules)
        >> leftPredicate[lambda derivation:
        sum([g.vLabelCount("O") for g in derivation.left])<=4](repeat[10](inputRules))
)
```

A right predicate on the other hand, can constrain the properties of the molecules we are producing. In this way, we can inhibit the rule from creating incredibly long chains that might be unlikely. For example, a reaction that produces long carbon chains, but with incredibly low likelihoods of producing anything of 8 carbons or longer. Below shows code detailing how you can add this to your strategy so a chain length of 8 atoms is never produced. By using graphDFS, we can add wildcards, *, meaning no molecule of 8 atoms in length will ever be produced.

```
# Define the pattern you do not wish to see in SMILES or graph DFS
chain = graphDFS("[*]{*}[*]{*}[*]{*}[*]{*}[*]{*}[*]{*}[*]{*}[*]{*}[*]")

chain.print() # optional

ls = LabelSettings(LabelType.Term, LabelRelation.Unification)
dg = DG(graphDatabase=input_molecules, labelSettings=ls)
reaction_network = dg.build()

# Then we build the dg as usual, defining the predicate in the strategy
strategy = (
    addSubset(input_molecules)
        >> rightPredicate[lambda derivation:
        all(chain.monomorphism(g, labelSettings=ls)
        <1 for g in derivation.right)]
        (repeat[10](inputRules))
)

reaction_network.execute(strategy)

del reaction_network
dg.print()
```

3.3 Editing the Appearance of a Derivation Graph

Derivation graphs can also be edited after expansion to organise their structure prior to printing. This is particularly useful if many reactions rely on the same molecules, such as cofactors (like NADPH) in enzymatic pathways, which can lead to busy nodes that disturb the visibility of the derivation graph. In this case, these nodes and the associated hyperedges can be hidden from the end derivation graph.

```
# First, we create a list of molecules we wish to hide from the derivation graph
hidden_molecules = []

# Then define how to print the derivation graph
Def printDG(hidden_molecules):
```

```
    p = DGPrinter()
# Explicitly stating which molecules and their vertices are allowed to be shown
    p.pushVertexVisible(lambda v: v.graph not in hidden_molecules)

dg.print(p)
```

4 Flow Queries

5 Stochastic Simulations

Once we have expanded our derivation graph, we can begin to investigate the kinetics of the system by running stochastic simulations. The kinetic mechanisms defining reactions in chemical reaction networks can be modelled by a series of differential equations that describe the production of products using terms called kinetic constants. In a simulation that uses ordinary differential equations, there is no variation in species concentrations between separate simulations. In biochemical networks, this is rarely the case, as many parameters can impact the reaction outcome. To model this more accurately, we can introduce the concept of stochasticity to the network, accounting for the inherent variability and randomness often seen in biochemical reactions conducted in the lab.

To begin a stochastic simulation in MOD there are a few things we must complete for the initial set up. Importing the stochastic package from MOD itself, loading the molecules and rules, and defining the rates. We will first begin with a code block showcasing the relevant imports.

```
import sys
import mod.stochsim as stoch

include("formose.py")

input_molecules = [formaldehyde, glcolaldehyde]

```

Now we can we can turn our attention to the kinetic rates associated with each rule.

As previously mentioned in 2.3, the hyperedges are produced by the rule, so to define the rate associated with a rule we query the rule data within the hyperedges of the derivation graph. Below, a code snippet shows how to define the rate constants for each rule as a simple first-order (rate = k[A]) kinetic equation.

```
# Associating rate to rules in a list
def reaction_rate(hyperedge):
    rule_rates = [rates[rule.name] for rule in hyperedge.rules]
    return rule_rates[0], False

# Defining the rate constant of each rule
aldol_addition_rate = 0.01
keto_enol_rate = 0.1
aldol_addition_reverse_rate = aldol_addition_rate / 2
keto_enol_reverse_rate = keto_enol_rate / 2

# Linking variables to the named rules
rates = {
    "Aldol Addition": aldol_addition_rate,
    "Aldol Addition reverse": aldol_addition_reverse_rate,
    "Keto-enol isomerization": keto_enol_rate,
    "Keto-enol isomerization reverse": keto_enol_reverse_rate,
}
```

With the reaction rates now defined, we can set up the initial state of the system. The derivation graph is generated simultaneously with the stochastic simulation, so any strategies we want to apply to the network must be specified now before starting the simulation. An important point to consider is that in a chemical reaction network, stochastic simulations are handled as discrete reaction events in MOD, so when a stochastic simulation is created, it must be initialised with an integer value as they relate to individual molecules and the reactions that occur between them. It is also valuable to use larger initial counts to avoid an extinction event, as using a single-digit number could cause the count to reach 0 quickly, thereby stopping the simulation.

```
ls = LabelSettings(LabelType.Term, LabelRelation.Specialisation)

# Initial counts (not concentrations) of each species
```

```
FORMALDEHYDE_INIT = 100
GLYCOLALDEHYDE_INIT = 1000

init_state = {
    glycolaldehyde: GLYCOLALDEHYDE_INIT,
    formaldehyde: FORMALDEHYDE_INIT
}

# The stochastic framework uses a random start seed for the Gillespie algorithm;
# if we defined a specific seed, all results would share the same trajectories
seed = None

# Stochastic Simulation
# Defining the initial state of the system and the simulation
sim = stoch.Simulator(
    labelSettings = LabelSettings(LabelType.Term, LabelRelation.Specialisation),
    # Adding the initial molecules to the simulation
    graphDatabase = input_molecules,
    # Defining the rules to expand the network with
    expandNetwork = stoch.ExpandByStrategy(inputRules),
    # Predefined initial state, starting counts of molecules
    initialState = init_state,
    # Telling MOD where to find the kinetic equations
    draw = stoch.DrawMassAction(reactionRate=reaction_rate)
)

# Simulate and draw the traces of each species, defining time in seconds
trace = sim.simulate(time=1000)

del sim
trace.print()
```