ALIFE 2025 MØD Tutorial

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

Anne-Susann Abel[1,2], Yingjie Pan[1,3], Johannes Petersen[2,4], Mikkel Boger Posselt[2,5], Joseph Antony Smith[1,3], Manuel Uhlir[1]

Training Alliance of Computational Systems Chemistry (TACsy)
Marie Curie-Skłodowska Actions
Funded by the European Union

1: Theoretical Biochemistry Institute, Institute for Theoretical Chemistry, University of Vienna, Vienna, Austria
2: Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark
3: School of Chemistry and Chemical Engineering, University of Southampton, Southampton, UK
4: Machine Learning Research Unit, Vienna University of Technology, Vienna, Austria
5: Bioinformatics Group, Department of Computer Science, Leipzig University, Leipzig, Germany

1 Introduction

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/

MØD 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 MØD via a Docker image (instructions available on the tutorial website linked above).

1.3 What is MØD?

MØD is a computational tool developed by the University of Southern Denmark that enables the automated generation, simulation, and optimisation of chemical reaction networks, utilising graph-based rewrite rules to represent molecular transformations. The concept of graphs and rules will be explained throughout the duration of the tutorial and within this accompanying document.

The tutorial will introduce MØD as a versatile platform for exploring chemical systems. Participants will learn how to generate derivation graphs, visual representations of reaction networks, and extend their analyses to stochastic simulations and pathway optimization via the in-built flow query tool. MØD also offers a variety of functionality at a more molecular/mechanistic level, such as electron push-out diagrams for depicting the exact mechanisms behind the reaction. By bridging chemistry, computation, and network modeling, MØD provides a functional framework for investigating emergent properties of complex reaction networks.

1.4 The Formose Chemistry

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

The pathway begins with formaldehyde (CH_2O) 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 Chemistry as Graphs

2.1 Writing Molecules as SMILES Strings

Our first task in this MØD 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 MØD to process even complex molecular structures.

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.

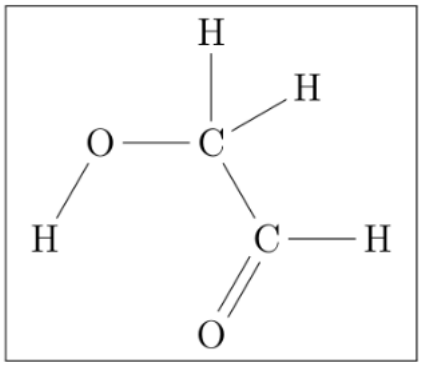
Writing a SMILES string in MØD is simple, create a variable and use the MØD function, Graph.fromSMILES(). Within the parentheses are two arguments, the SMILES string itself is simply confined within " ", and an optional, name=" " argument, can be added. Below is a snippet of example code for writing glycolaldehyde as a SMILES string.

```
# Writing a simple SMILES string in MØD
glycolaldehyde = Graph.fromsmiles("OCC=O", name="Glycolaldehyde")

glycolaldehyde.print()
```

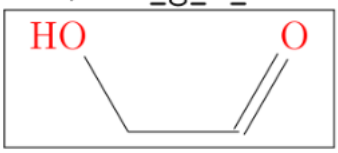
As mentioned above, the implicit hydrogens have been omitted; however, the result would be the same if they were explicitly included within the SMILES string. Alternatively, graphDFS is also a capable alternative to SMILES, but does require the addition of hydrogen atoms. Each individual atom must be enveloped by square brackets, [], while branches are enclosed in (), similar to SMILES. Below is example code to help visualise the differences.

```
# Writing a simple graphDFS in MØD
Glycolaldehyde = Graph.fromDFS("[O]([C]([C](=[O])[H])([H])[H])[H]",
name="Glycolaldehyde")

glycolaldehyde.print()
```

Within MØD, 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 MØD, atoms are represented as nodes, and the bonds as edges.



File: `out/001_g_0_10300000`

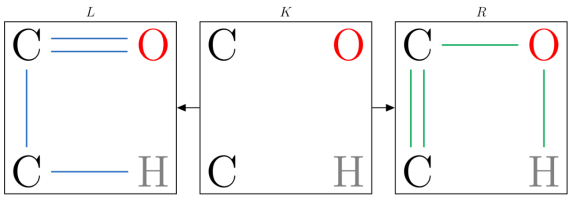


File: `out/003_g_0_11310100`

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 (see below) 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 MØD can provide.



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.

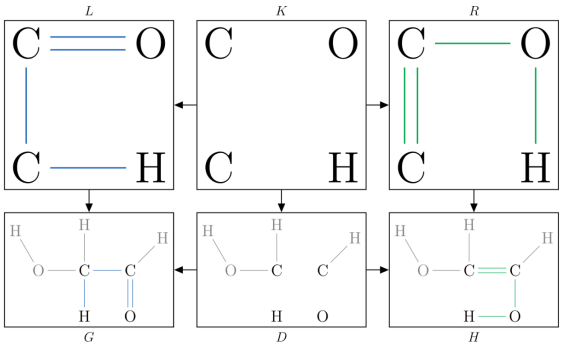
Below is a sample of code for a rule describing a keto-enol isomerisation reaction. The rule contains an ID, effectively a name, helpful for navigating rules in the printed PDF document, should you choose to print them.

```
# 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()
```

In the formose chemistry, the reactions are reversible. In MØD reversible reactions do not need to be rewritten as an entirely new section of GML, instead we can apply the invert argument to it, simply swapping the left and right sides of the rule around, allowing it to act in the reverse direction. A code snippet is supplied below.

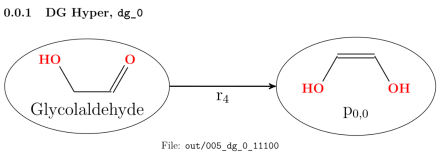
```
# Load the rule from above, adding _F to denote it as the forward reaction
ketoEnol_F = Rule.fromGMLString(ketoEnolGML)

# Now we can add the inverse argument, creating a reversed version of the rule
# Adding _B to denote it as the backwards reaction
ketoEnol_B = Rule.fromGMLString(ketoEnolGML, inverse=True)
```

In the image below, the rule we created called "Keto-enol isomerisation" has been applied in its 'forward' variation to the molecule we previously created, glycolaldehyde. This hopefully helps you visualise how a rule works.



2.3 Derivation Graphs

Once we have a set of rewrite rules that encode the 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 inform the user which rule was applied to create the resulting hyperedge and products.



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, which are automatically added to a list called inputRules
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 MØD, 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 MØD enable us to improve the results of the derivation graph by providing the user with 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 operations that impact how the graph is expanded. These operations can vary from the number of repeats (how many times MØD 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 MØD to apply the rules within the strategy. Within the Python file, we can have multiple lists of input molecules and rules. This allows us to create an initial expansion with one set of molecules and then expand upon it again with another, simply by adding another instruction. Alternatively, all molecules can be present, but the rules applied can be changed.

```
# This time, we define a strategy using a definite number of repeats
strategy = (
    addSubset(input_molecules)
        # Using the inputRules list automatically created by MØD
        >> 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 derivation graph is 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, 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 perhaps this enzyme specifically does not catalyse. This is

where a left predicate can be used, commanding the rule to specifically avoid defined subgraphs/patterns.

```
# 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:
        # num_subgraphs takes two arguments, the graphs to appl
        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

5.1 Setting Up a Simulation

MØD allows users to investigate the kinetics of their chemical systems by running stochastic simulations. The kinetic equations governing 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 MØD there are a few things we must complete for the initial setup. Importing the stochastic package from MØD itself, loading the molecules and rules, and defining the rates. We will first begin with a code block showcasing the relevant imports.

We also need to include a file provided to you called "constraints.py". This is a file of various constraints to limit the combinatorial expansion possible when expanding a network based on the formose chemistry.

```
import sys
import mod.stochsim as stoch

# Pre-made file containing the rules and molecules we previously created.
include("formose.py")
# Includes some constraints that we need to include to limit the expansion.
include("constraints.py")

input_molecules = [formaldehyde, glcolaldehyde]

# We need this for the constraints we add later to work
ls = LabelSettings(LabelType.Term, LabelRelation.Specialisation)

```

5.2 Kinetic Rates

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 the rule data in each hyperedge
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
# Feel free to change these numbers around in subsequent simulations
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,
}
```

5.3 Initial State

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 MØD, 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.

```

# 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, results would share the same initial trajectories
seed = None
```

Due to the formose chemistry's combinatorial nature, we have created some constraints for you to add to the system, without these constraints we might be here for the full week waiting for the simulation to complete. These constraints can be viewed within the "constraints.py" file in the tutorial folder. The file simply refrains the system from producing 3 and 4 molecule rings as these are thermodynamically

unfavourable, and setting a limit to the length of the chains we produce. Within the file, their application conditions are also encoded.

```

expansion_strategy = (
    rightPredicate [lambda d:
    all_constraints_apply(CONSTRAINT_FUNCTIONS, d)]
    (reaction_rules)
)
```

5.4 Running the Simulation

We are now ready to run the simulation. We first start by adding the label settings, then specifying the input molecules, input rules, initial species counts, and the kinetic rates for each rule, as arguments within the stoch.Simulator function. Finally we define a time frame for the simulation, then print the result.

```

# 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 MØD 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()
```

This sequence of code leaves us with a trace plot of the concentrations of each species changing over time in the summary file generated by MØD. By working with some Python packages, we can generate various plots showcasing statistical information about the system.

5.5 Interpretation and Analysis

A very useful use of stochastic simulations is for generating an ensemble plot. This plot uses the traces from many different simulations run consecutively, generating an average trajectory and upper and lower bounds based on the 5th and 95th percentiles. This can provide us with detailed insight into the system behaviour by increasing the sample size. We can begin to edit the code we just wrote to run more simulations, averaging them, and plotting them on the same axes.

```
# Include the callbacks.py file at the beginning of the current file with the other files.
include("constraints.py")

# The expansion strategy remains the same, but is implemented differently
expansion_strategy = (
    rightPredicate [lambda d:
    all_constraints_apply(CONSTRAINT_FUNCTIONS, d)]
    (reaction_rules)
)

# The following lines (12-33) replace the stochastic simulation code from lines (23-39) from befor

# Create an empty list for the simulation results
simulations = []

# This sets the number of successive simulations to 50
for index in range(50):
    print(f"Starting simulation {index+1}/50")
    sim = stoch.Simulator(
            labelSettings=ls,
            graphDatabase=[formaldehyde, glycolaldehyde],
```

```
            expandNetwork=stoch.ExpandByStrategy(expansion_strategy),
            initialState=init_state,
            draw=stoch.DrawMassAction(reactionRate=reaction_rate)
    )
    # Apply callbacks with parameter information
    setCallbacks(sim, verbose=False)

# Perform statistical analysis and plot the results
statistics.statistical_analysis(output_dir='analysis_results',
    # ci stands for confidence interval
    uncertainty_type='ci',
    confidence_level=0.95
)
```

Enjoy :)