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

A tutorial version of MØD is available to install via a Docker Image if you wish to try it during this tutorial session. Docker is required to run this version. Both a link to installing Docker and the Docker Image of MØD can be found by following a link to our Google site 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 transformation 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 catalyze the production of longer-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

This section covers what you need to know about graph theory for the purposes of this tutorial. This document nor the presentation does not dive into the mathematical and technical details behind using graphs to depict chemistry, instead focusing on how we can use it to develop and aid our understanding of chemical networks and how that applies to MØD. If you wish to understand more about the subject and how it is applied within MØD, there is extensive documentation, which you can find at the link below:

https://jakobandersen.github.io/mod/graphModel/index.html

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.

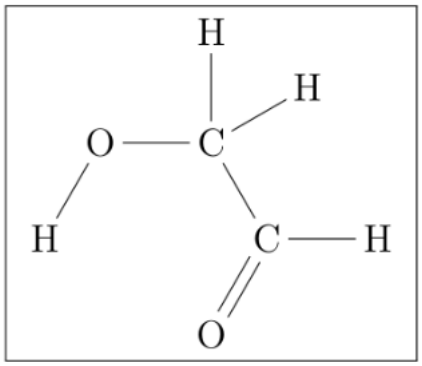
Writing a SMILES string in MØD is simple, create a variable and use the MØD function, Graph.fromSMILES(). Within the parentheses are two arguments, the SMILES string itself is simply confined within " ", and an optional, name=" " argument, can be added. Below is a snippet of example code for writing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

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

glycolaldehyde.print()
```

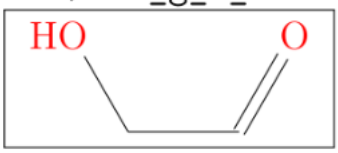
As mentioned above, the implicit hydrogens have been omitted; however, the result would be the same if they were explicitly included within the SMILES string. Alternatively, graphDFS is also a capable alternative to SMILES, but does require the addition of hydrogen atoms. Each individual atom must be 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`

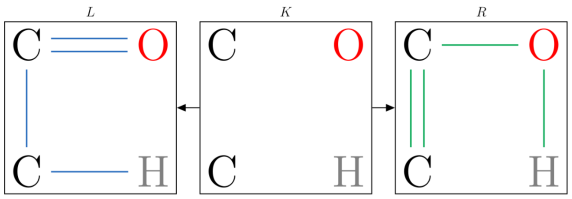


File: `out/003_g_0_11310100`

2.2 Writing Reactions as Graph Transformations

So now that we have a basic understanding of a chemical molecule represented as a graph, how can we transform one graph into another, like a molecule changing during a chemical reaction? A graph transform reorganises the location of edges (bonds) and the information of a node (the atom) by changing the character (atomic symbol). Obviously the an atom cannot change into another atom in a chemical reaction, but it can change charge, and this can be represented in the node.

A graph transformation (see below) is written as a rule to govern the expansion of the system. 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 - this is referred to as sharing a common subgraph. However, the left does not need to contain the nodes and edges of the entire reactant molecule, only a portion (subgraph/functional group), as long as they are in the same order/structure (isomorphic), the rule will apply. This tunable aspect of rule specificity 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.

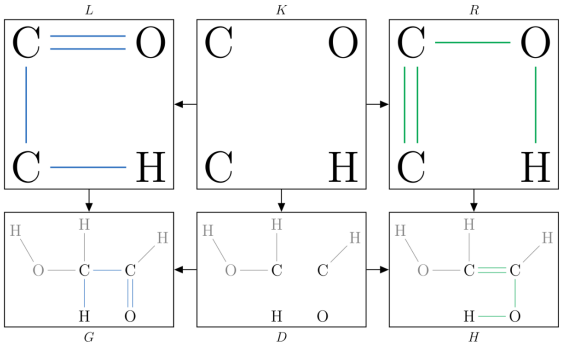
```
# String containing a GML rule representing keto-enol isomerisation
ketoEnolGML = """rule [
    ruleID "Keto-enol isomerisation"
    left [
        # edge defines a bond, and must have a start (source) and end (target)
        # the label defines the type of bond
        edge [ source 1 target 4 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 "-" ]
    ]
]"""

```

```
25 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.

```
1 # Load the rule from above, adding _F to denote it as the forward reaction
2 ketoEnol_F = Rule.fromGMLString(ketoEnolGML)
3
4 # Now we can add the inverse argument, creating a reversed version of the rule
5 # Adding _B to denote it as the backwards reaction
6 ketoEnol_B = Rule.fromGMLString(ketoEnolGML, inverse=True)
```

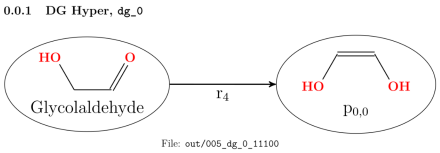
In the image below, the rule we created called "Keto-enol isomerisation" has been applied in its 'forward' variation to the molecule we previously created, glycolaldehyde. This hopefully helps you visualise how a rule works.



2.3 Derivation Graphs

Once we have a set of graph transformation 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:
        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. Code below details how you can add this to your strategy so that a chain length of 8 atoms is never produced. Using graphDFS, we can specify the subgraph we wish to avoid by adding wildcards, *, which means that 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()
```

Within some of the code snippets in this section you may have noticed syntax such as ’

4 Hyperflow Queries

After generating an entire chemical space, we can begin to find individual pathways based on an objective function defined by the user. The following example will use an objective function that searches for the shortest possible pathway.

For additional information about the theory and implementation of this in MØD, please see the following link https://jakobandersen.github.io/mod/hyperflowModel/index.html

4.1 Initialising a Hyperflow Problem

To use hyperflow, we need to have a derivation graph to use it on, we have already discussed and know how to create a derivation graph with the formose chemistry graph transformation rules, but we will repeat it in a code snippet for you here for completeness. This snippet also shows you how to save a derivation graph, should you need it later.

```
# Loading in the rules and molecules
include("grammar_formose.py")

strat = (
    # The universe in this case defines molecules that MOD does not apply rules to
    # Formose in this case cannot react with itself or undergo an aldol addition
        addUniverse(formaldehyde)
        >> addSubset(glycolaldehyde)
        # Constrain the reactions:
        # No molecules with more than 20 atoms can be created.
        >> rightPredicate[lambda derivation:
    all(g.numVertices <= 20 for g in derivation.right)]
    (repeat(inputRules)
```

```
        )
)

# Create DG
dg = DG(graphDatabase=inputGraphs)
dg.build().execute(strat)
dg.print()

# Save DG
dg.dump("dg_autocat.dg")
```

Next we can load in the derivation graph that we wish to use. Saving and loading the graph is not a necessary step but it is useful to know, instead of creating a new one each time you wish to perform some operations on it. It is also possible to query the number of edges and vertices in the derivation graph to get a sense of scale.

```
# Include the grammar used to create the DG (includes the molecules and the rules used to create t
include("grammar_formose.py")

# Define the filename of the DG to load.
dg_name = "dg_ex_3"

# Load the DG in question from the previous DG dump file.
# inputGraphs and inputRules are parsed from the grammar file that was loaded earlier
dg = DG.load(inputGraphs, inputRules, f"{dg_name}.dg")

# Explore the size of the dg
def display_size_dg(dg):
    # print the number of vertices of the dg
    print("Number of vertices")
    print(dg.numVertices)
    # print the number of edges of the dg
    print("Number of edges")
    print(dg.numEdges)

display_size_dg(dg)
```

4.2 Pathway Search

Now we can look for a specific pathway within our generated chemical reaction space. This uses integer linear programming (ILP) to constrain and optimise the system given a required goal. The isEdgeUsed variant of the objective function uses a binary count system to note whether an edge is used in a pathway or not (effectively an on/off switch), automatically searching for the lowest number from a continuously switched-on pathway.

4.2.1 Example 1: Cycles

```
# Include the grammar used to create the dg
include("grammar_formose.py")

# Define the name of the dg to load
# This dg has been created before for this purpose;
# the script to make it is called make_autocat_dg.py
dg_name = "dg_autocat"

# Load the dg in question from the previous dg dump file
dg = DG.load(inputGraphs, inputRules, f"{dg_name}.dg")

# A flow model is the system of linear equations fed into an ILP solver
flow_name = "flow"

# Define the flow model
flow = hyperflow.Model(dg)

# To find a pathway through the network,
# we need inflows (sources) and outflows (sinks)
# Define the input molecules for the network
flow.addSource(formaldehyde)
flow.addSource(glycolaldehyde)

# Define the output molecules of the network
flow.addSink(glycolaldehyde)

# Add constraints on the amount of molecules used
```

```
28 flow.addConstraint(inFlow[formaldehyde] == 2)
29 flow.addConstraint(inFlow[glycolaldehyde] == 1)
30 flow.addConstraint(outFlow[glycolaldehyde] == 2)
31
32 # Disable flow reversal (always a positive driving force):
33 flow.allowIOReversal = False
34
35 # Set the objective function
36 # Queries whether the edge (molecule) is used in a reaction
37 flow.objectiveFunction = isEdgeUsed
38
39 # Find a flow solution from our function and constraints
40 flow.findSolutions()
41
42 # Show solution information in the terminal
43 flow.solutions.list()
44
45 # Print solutions
46 flow.solutions.print()
47
48 # save the flow as a file
49 flow.dump(f"{dg_name}_{flow_name}.flow")
```

4.2.2 Example 2: Autocatalysis

From this point on, all code snippets replace the code from lines 33 to 46 in Section 4.2.1. This time, we are searching the network for autocatalytic cycles using the built-in autocatalysis search function. Working in conjunction with the isEdgeUsed, it finds the smallest possible autocatalytic cycle.

```
1 # Enables the autocatalysis search for autocatalytic cylces
2 flow.overallAutocatalysis.enable()
3
4 # Set the objective function
5 flow.objectiveFunction = isEdgeUsed
6
7 # Now continues the same as above
8 flow.findSolutions()
```

```
flow.solutions.list()
flow.solutions.print()
flow.dump(f"{dg_name}_{flow_name}.flow")
```

4.3 Constraining the Solution

Suppose we wish to find an autocatalytic cycle that only uses molecules of a certain size or functional group. Then, similar to the predicates in Section 3.2, we can create a constraint to which the solver can adhere.

```
# Forbid molecules with more than 4 Carbon atoms in the flow solution
for v in dg.vertices:
    if v.graph.vLabelCount("C") > 4:
        flow.addConstraint(vertexFlow[v] == 0)

# Continuing on with an objective function and solutions
flow.objectiveFunction = isEdgeUsed

flow.findSolutions()
flow.solutions.list()
flow.solutions.print()
flow.dump(f"{dg_name}_{flow_name}.flow")
```

4.4 Alternative Objective Function

There is also the edgeFlow variant of the objective function. This minimises the actual weight of the edges, so how many times it is used in a pathway, which depends on the number of molecules specified at the beginning in the flow constraints.

```
# Try different objective functions
flow.objectiveFunction = edgeFlow

# Run flow
flow.findSolutions()
```

```
flow.solutions.list()
flow.solutions.print()
flow.dump(f"{dg_name}_{flow_name}.flow")
```

4.5 The Objective Function as a Linear Equation

By replacing just the objective function in the previous code, we can create a linear equation for the flow to attempt to minimise (this is the default in the flow query module) by combining the two previous functions.

```
# You can formulate any linear equation as an objective function:
flow.objectiveFunction = isEdgeUsed * 1000 + edgeFlow

# Running the flow again
flow.findSolutions()
flow.solutions.list()
flow.solutions.print()
flow.dump(f"{dg_name}_{flow_name}.flow")
```

5 Stochastic Simulations

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 (ODEs), there is no variation in species concentrations between separate simulations. In a stochastic simulation, a reaction - and the molecules related to it - are treated as discrete events with a probability of occurrence based on their reaction constants. As time progresses within the simulation, the probability of whether this reaction has occurred or not may cause the count of the tracked molecule to change. At any given time point, there may be no reaction events or many. Making each simulation slightly different, as this could change dramatically between individual simulations. In an ODE, there is no probability, and therefore no randomness, producing the same result each time.

But why would we want to introduce an element of randomness to our simulations? In biochemical networks (and chemical networks with low molecule counts), 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.

5.1 Setting Up a Simulation

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")
# Contains analysis functions for processing results later
include("analysis.py")


input_molecules = [formaldehyde, glcolaldehyde]
```

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, the following code is appended to the end of the previous snippet. 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 multiple simulations, each producing different results.

```
# 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 replace the stochastic simulation code
# from the snippet in Section 5.4 above.

# Simulation parameters
# Duration of each simulation
SIMULATION_TIME = 100
# Number of independent simulations to run
NUMBER_OF_SIMULATIONS = 50


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

# Run multiple independent simulations for statistical analysis
# Each simulation starts with the same initial conditions but follows
# a different stochastic trajectory
for index in range(NUMBER_OF_SIMULATIONS):
    print(f"Starting simulation {index+1}/{NUMBER_OF_SIMULATIONS}")

    # Create a new simulator for each run
    sim = Simulator(
        # Starting molecules
        graphDatabase=[formaldehyde, glycolaldehyde],
        # How to grow the network
        expandNetwork=Simulator.ExpandByStrategy(expansion_strategy),
        # Initial counts
        initialState=init_state,
        # Kinetics model
        draw=Simulator.DrawMassAction(reactionRate=reaction_rate)
        )

    # Run the simulation for the specified time
    trace = sim.simulate(time=SIMULATION_TIME)
```

```
41
42      # Run the simulation for the specified time
43      trace = sim.simulate(time=SIMULATION_TIME)
44
45      # Extract simulation data into a structured format for analysis
46      # This converts the raw simulation trace into a SimulationCache object
47      # that contains all the relevant information about the simulation
48      simulation = extract_simulation(
49          trace,
50          sim.dg,
51          params={
52              'time_limit': SIMULATION_TIME,
53              'formaldehyde_init': FORMALDEHYDE_INIT,
54              'glycolaldehyde_init': GLYCOLALDEHYDE_INIT
55          },
56          verbose=False
57      )
58
59      # Store the extracted simulation data
60      simulations.append(simulation)
```

Now we can begin to analyse the data we just produced by using some very helpful Python packages.

```
1   # Create a SimulationStatistics object to analyze all simulations
2   # This object provides methods for statistical analysis across multiple runs
3   statistics = SimulationStatistics(simulations)
4
5   # Perform comprehensive statistical analysis
6   # This generates plots, calculates statistics, and saves results to files
7   statistics.statistical_analysis(
8       output_dir='analysis_results',  # Directory to save analysis results
9       uncertainty_type='ci',          # Use confidence intervals for uncertainty
10      confidence_level=0.95,          # 95% confidence level
11      top_n_species=5,                # Top 5 species by average concentration
12  )
```