Selected Topics in Programming Assignment

Jens Jacob Torvin Møller

June 14, 2024

Listing 1: ./Benchmarker.h

```cpp
//
// Created by jjtor on 11/06/2024.
//

#ifndef EKSAMENSPROJEKT_BENCHMARKER_H
#define EKSAMENSPROJEKT_BENCHMARKER_H

#include "bits/stdc++.h"

class Bench {
public:
    void start_clock(std::string name);
    void stop_clock(std::string name);
    void report();
private:
    struct tp {
        std::string name;
        std::chrono::high_resolution_clock::time_point start;
        std::chrono::high_resolution_clock::time_point end;
    };
    std::vector<tp> storage;
};

#endif //EKSAMENSPROJEKT_BENCHMARKER_H
```

Listing 2: ./GlobalState.h

```cpp
//
// Created by jjtor on 30/05/2024.
//

#ifndef EKSAMENSPROJEKT_GLOBALSTATE_H
#define EKSAMENSPROJEKT_GLOBALSTATE_H

#include <bits/stdc++.h>
#include "Molecule.h"
#include "Reaction.h"





#endif //EKSAMENSPROJEKT_GLOBALSTATE_H
```

Listing 3: ./Grapher.h

```cpp
//
// Created by jjtor on 02/06/2024.
//
```

```
#ifndef EKSAMENSPROJEKT_GRAPHER_H
#define EKSAMENSPROJEKT_GRAPHER_H

#include <bits/stdc++.h>

#include <utility>

std::string cleanString(std::string str) {
    str.erase(std::remove(str.begin(), str.end(), ':'), str.end());
    return str;
}

class Grapher{
private:
    std::string start = "digraph{";
    std::string end = "}";
    std::ofstream out;
    int delay_index = 0;
    std::vector<std::string> molecule_labels = std::vector<std::string>();
    std::vector<double> delays = std::vector<double>();
public:
    Grapher(std::string name){
    std::string path="..\\out\\out_" + cleanString(name) + ".txt";
    out = std::ofstream(path);
    out << start;
    };
    ~Grapher(){
        out << end;
        std::cout << "Grapher destroyed. ";
    }

    void Graph(std::list<stochastic::Reaction> reactions){
        for (auto r:reactions) {
            auto delay = AddDelay(r.get_current_rate_parameter());
            for (auto reactant:r.get_reactants()) {
                AddMolecule(reactant.GetName());
                for (auto p:r.get_products()) {
                    AddArrow(reactant.GetName(), p.GetName(), r.get_current_rate_parameter());
                }
            }
        }
        std::cout << "Graphing done";
    }

    void AddMolecule(std::string label){
        if (std::find(molecule_labels.begin(), molecule_labels.end(), label) != molecule_labels.end()){
            return;
        }
        molecule_labels.push_back(label);
        out << label << "[label=\"" << label << R"(",shape="box",style="filled",fillcolor="cyan"];)" << "\n";
    }
    std::string AddDelay(double delay){
        /*if (std::find(delays.begin(), delays.end(), delay) != delays.end()){
            return "r" + std::to_string(delay_index);
        }*/
        delays.push_back(delay);
        auto r = "r" + std::to_string(delay_index);
        //out << "r" + std::to_string(delay_index) << "[label=\"" << std::to_string(delay) <<
↪R"(",shape="oval",style="filled",fillcolor="yellow"];)" << "\n";
        delay_index++;
        return r;
    }

    //Takes in (label, delay) || (delay, label)
    template<class T1, class T2>
    void AddArrow(T1 source, T2 target, double delay){
        out << source << "->" << target << "[label=\"" << (double)(delay / 0.01) * 0.01 <<"\"]\n";
    }
};


#endif //EKSAMENSPROJEKT_GRAPHER_H
```

Listing 4: ./Molecule.h

```
//
// Created by jjtor on 06/05/2024.
//

#ifndef EKSAMENSPROJEKT_MOLECULE_H
#define EKSAMENSPROJEKT_MOLECULE_H
#include <stdlib.h>
#include <string>




#endif //EKSAMENSPROJEKT_MOLECULE_H
```

Listing 5: ./Observer.h

```
//
// Created by jjtor on 11/06/2024.
//

#ifndef EKSAMENSPROJEKT_OBSERVER_H
#define EKSAMENSPROJEKT_OBSERVER_H

#endif //EKSAMENSPROJEKT_OBSERVER_H
```

Listing 6: ./PrettyPrinter.h

```
//
// Created by jjtor on 31/05/2024.
//

#ifndef EKSAMENSPROJEKT_PRETTYPRINTER_H
#define EKSAMENSPROJEKT_PRETTYPRINTER_H

#include <bits/stdc++.h>
#include "StochasticSimulation.h"



/*template<class T>
std::ostream& operator<<(std::ostream& os, Molecule molecule){
    os << molecule.GetName();
    return os;
}*/

```

```
#endif //EKSAMENSPROJEKT_PRETTYPRINTER_H
```

Listing 7: ./Reaction.h

```
//
// Created by jjtor on 30/05/2024.
//

#ifndef EKSAMENSPROJEKT_REACTION_H
#define EKSAMENSPROJEKT_REACTION_H

#include <bits/stdc++.h>






#endif //EKSAMENSPROJEKT_REACTION_H
```

Listing 8: ./StochasticSimulation.h

```
//
// Created by jjtor on 06/05/2024.
//

#pragma once

#include <bits/stdc++.h>
#include <chrono>
#include <functional>
#include <algorithm>
#include "Reaction.h"
#include "Molecule.h"
#include <map>

namespace stochastic {


    class Environment {
    public:
        Environment() {};
    };

    class Reaction;

    class Molecule {
    private:
        std::string symbol;
        int current_amount;
    public:
        Molecule(std::string name, double amount) { symbol = name, current_amount = amount; }

        int get_current_amount() const { return current_amount; }

        std::string GetName() const { return symbol; }

        void set_current_amount(int val) { current_amount = val; }

        //Overloads
        Reaction operator+(Molecule molecule) const;

```

```cpp
    Reaction operator+(Reaction reaction);

    Reaction operator>>(double delay) const;

};

class GlobalState {
private:
    double time = 0;
    //std::list<Molecule> reactants; //Current molecules swimming around

    template<class T, class U>
    struct GenericLookupTable {
        std::map<T, U> table;

        auto LookUp(T search) {
            return table.find(search);

            /*if (auto it = table.find(search); it != table.end())
                return it;*/
        }

        void Insert(Molecule m) {
            table.insert({m.GetName(), m.get_current_amount()});
        }

        void Update(T element, int value) {
            auto it = LookUp(element);
            if (it != table.end()) {
                it->second += value;
            } else {
                table.insert({element, value});
                std::cout << element << " is not in symbol table. Has now been inserted";
            }
        }

    };

    std::list<Reaction> reactions;

public:
    GlobalState() {};
    Environment environment;
    GenericLookupTable<std::string, int> symbolTable = GenericLookupTable<std::string, int>();

    void AddReactant(Molecule reactant) {
        //reactants.push_back(reactant);
        symbolTable.Insert(reactant);
    }

    void AddTime(double time_to_add) { time += time_to_add; }

    double GetCurrentTime() { return time; }
};

class Reaction {
private:
    std::vector<Molecule> reactants;
    double rate_parameter;
    double delay;
    std::vector<Molecule> products;
public:
    Reaction() {
        delay = std::numeric_limits<double>::infinity();
    };

    double get_current_rate_parameter() const { return rate_parameter; }

    double get_current_delay() const { return delay; }

    void set_delay(double d) { delay = d; }

    void set_rate_parameter(double rp) { rate_parameter = rp; }

    std::vector<Molecule> &get_reactants() { return reactants; }

    std::vector<Molecule> &get_products() { return products; }

    void add_reactant(const Molecule &reactant) { reactants.push_back(reactant); }

    void add_product(const Molecule &product) { products.push_back(product); }

    //Overloads
    Reaction operator>>(double rate) {
        auto r = Reaction();
        for (const auto &reactant: this->get_reactants()) {
            r.add_reactant(reactant);
        }
        r.set_rate_parameter(rate);
        return r;
    };

    Reaction operator>>=(Molecule molecule) {
        add_product(molecule);
        return *this;
    };

    Reaction operator>>=(Reaction reaction) { //TODO: Add copy assignment constructor to reaction to copy all of this, instead of manually doing so
        auto r = Reaction();
        for (const auto &reactant: reaction.get_reactants()) {
            r.add_product(reactant);
        }
        for (const auto &reactant: this->get_reactants()) {
            r.add_reactant(reactant);
        }
        r.set_rate_parameter(this->get_current_rate_parameter());
        return r;
    };

    Reaction operator>>=(Environment env) {
        ;
        return *this;
    };
};

class Vessel {
private:
    std::string name;
    std::list<Reaction> reactions;
public:
    Vessel(std::string n) { name = n; }

    GlobalState global_state = GlobalState(); //Environment
    std::list<Reaction> &GetReactions() { return reactions; }

    std::string GetName() { return name; }

    Molecule add(std::string name, double amount) {
        auto molecule = Molecule(name, amount);
        global_state.AddReactant(molecule);
        return molecule;
    };

    void add(const Reaction reaction) {
        reactions.push_back(reaction);
    };
};

stochastic::Reaction FindSmallestDelayReaction(stochastic::Vessel &vessel);
class StochasticSimulation {
private:
    std::string path = "..\\out\\trajectory.csv";
    std::ofstream trajectory;
public:
    StochasticSimulation() {}

    void RunSimulation(Vessel vessel, double end_time);

    template<class Obs>
    void RunSimulation(Vessel vessel, double end_time, Obs observer){
        std::string mCount;
        std::string header;
        this->path = "..\\out\\trajectory_" + vessel.GetName() + ".csv";
        trajectory = std::ofstream(path);
        header += "Time,";
        trajectory << header;
        for (auto it = vessel.global_state.symbolTable.table.begin();
             it != vessel.global_state.symbolTable.table.end(); ++it) {
            if (std::next(it) != vessel.global_state.symbolTable.table.end()) {
                trajectory << it->first << ",";
            } else {
                trajectory << it->first << "\n";
            }
        }
        std::cout << "Running simulation. Time: " + std::to_string(vessel.global_state.GetCurrentTime()) + "\n";

        while (vessel.global_state.GetCurrentTime() <= end_time) {
            observer(vessel.global_state.GetCurrentTime(), vessel);
            for (const auto &[key, value]: vessel.global_state.symbolTable.table) {
                mCount += std::to_string(value) + ",";
            }
            mCount.pop_back();
            trajectory << vessel.global_state.GetCurrentTime() << ',' << mCount;
            trajectory << "\n";
            mCount.clear();
            for (auto &r: vessel.GetReactions()) {
                auto delay = ComputeReactionTime(r, vessel);
                r.set_delay(delay);
                //Fix so it is taken directly from symbol table without for-range loop
            }

        // Pick reaction with shortest delay (reaction time)
        auto min_delay_reaction = FindSmallestDelayReaction(vessel);

        vessel.global_state.AddTime(min_delay_reaction.get_current_delay()); //Line 5
        for (auto &q: min_delay_reaction.get_reactants()) {
          if (std::all_of(min_delay_reaction.get_reactants().begin(), min_delay_reaction.get_reactants().end(),
                  [&](Molecule &i) {
                    return vessel.global_state.symbolTable.LookUp(i.GetName())->second > 0;
                  })) {
            //TODO: Implement lookup/symbol table (To be..)
            vessel.global_state.symbolTable.Update(q.GetName(), -1);
          }
        }
        for (auto &p: min_delay_reaction.get_products()) {
          vessel.global_state.symbolTable.Update(p.GetName(), 1);
        }
        std::cout << "Simulation step done. Time: " + std::to_string(vessel.global_state.GetCurrentTime()) + "\n";
      }
      std::cout << "Simulation done. Time: " + std::to_string(vessel.global_state.GetCurrentTime());
    }

    template<class Obs>
    void RunSimulationParallel(Vessel vessel, double end_time, int numberOfSims, Obs observer);
    void RunSimulationParallel(Vessel vessel, double end_time, int numberOfSims);

    static double ComputeReactionTime(Reaction &reaction, Vessel &vessel);
  };


// Pretty printing
  template<class T>
  std::ostream &operator<<(std::ostream &os, std::list<T> const &container) {
    for (auto reaction: container) {
      os << "Reactants: [";
      for (auto reactant: reaction.get_reactants()) {
        os << reactant.GetName() << " ";
      }
      os << "\b \b";
      os << "] Rate parameter: ";
      os << "[" << reaction.get_current_rate_parameter() << "]";
      os << " Products: [";

      if (!reaction.get_products().empty()) {
        for (const auto p: reaction.get_products()) {
          os << p.GetName() << " ";
        }
        os << "\b \b";
      }
      os << "]\n";
    }
    return os;
  }
}
```

Listing 9: ./Vessel.h

```
//
// Created by jjtor on 30/05/2024.
```

```cpp
//

#ifndef EKSAMENSPROJEKT_VESSEL_H
#define EKSAMENSPROJEKT_VESSEL_H

#include "GlobalState.h"




#endif //EKSAMENSPROJEKT_VESSEL_H
```

Listing 10: ./Benchmarker.cpp

```cpp
//
// Created by jjtor on 14/06/2024.
//
#include "Benchmarker.h"
#include <iostream>

void Bench::start_clock(std::string name) {
    tp timepoint;
    timepoint.name = name;
    timepoint.start = std::chrono::high_resolution_clock::now();
    storage.push_back(timepoint);
}
void Bench::stop_clock(std::string name) {
    for (auto &v : storage){
        if (v.name == name){
            v.end = std::chrono::high_resolution_clock::now();
            return;
        }
    }
}

void Bench::report() {
    for (auto &v : storage){
        std::chrono::duration<double, std::milli> duration;
        duration = v.end - v.start;
        std::cout << v.name << " took " << duration.count() << " ms";
    }
}
```

Listing 11: ./GlobalState.cpp

```cpp
//
// Created by jjtor on 30/05/2024.
//
```

Listing 12: ./Grapher.cpp

```cpp
//
// Created by jjtor on 02/06/2024.
//

#include "Grapher.h"
```

Listing 13: ./main.cpp

```cpp
#include <iostream>
#include "Vessel.h"
#include "Molecule.h"
```

```cpp
#include "StochasticSimulation.h"
#include "PrettyPrinter.h"
#include "Grapher.h"
#include "Benchmarker.h"

void HospitalPeak();
stochastic::Vessel circadian_rhythm();
stochastic::Vessel seihr(uint32_t N);
stochastic::Vessel Figure1_1();
stochastic::Vessel Figure1_2();
stochastic::Vessel Figure1_3();


int main() {
    //Vessels
    auto c = circadian_rhythm();
    auto s = seihr(10000);
    auto f1 = Figure1_1();
    auto f2 = Figure1_2();
    auto f3 = Figure1_3();
    //std::cout << c.GetReactions();

    //auto grapher = Grapher("Seihr");
    //grapher.Graph(r);
    //grapher.Graph(s.GetReactions());

    auto benchmarker = Bench();

    auto sim = stochastic::StochasticSimulation();
    //sim.RunSimulation(f3, 2000);
    //sim.RunSimulation(c, 100);
    //sim.RunSimulation(s, 100);
    //HospitalPeak();

#pragma region Single Run 50 Simulations Benchmarking
    /*benchmarker.start_clock("SEIHR_SINGLE");
    for (int i = 0; i < 50; ++i) {
        std::cout << "\n" << "i = " << i;
        sim.RunSimulation(s, 100);
    }
    benchmarker.stop_clock("SEIHR_SINGLE");
    std::cout << "\n";
    benchmarker.report();*/
#pragma endregion
#pragma region Parallel Run 50 Simulations Benchmarking
    benchmarker.start_clock("SEIHR_PARALLEL");
    sim.RunSimulationParallel(s, 100, 50);
    benchmarker.stop_clock("SEIHR_PARALLEL");
    benchmarker.report();
#pragma endregion

    std::cout << "\nHello, World!" << std::endl;

    return 0;
}

//Vessels
stochastic::Vessel circadian_rhythm(){
    const auto alphaA = 50;
    const auto alpha_A = 500;
    const auto alphaR = 0.01;
    const auto alpha_R = 50;
    const auto betaA = 50;
    const auto betaR = 5;
    const auto gammaA = 1;
    const auto gammaR = 1;
    const auto gammaC = 2;
    const auto deltaA = 1;
    const auto deltaR = 0.2;
    const auto deltaMA = 10;
    const auto deltaMR = 0.5;
    const auto thetaA = 50;
    const auto thetaR = 100;


    auto v = stochastic::Vessel{"Circadian Rhythm"};

    const auto env = v.global_state.environment;

    const auto DA = v.add("DA", 1);
    const auto D_A = v.add("D_A", 0);
    const auto DR = v.add("DR", 1);
    const auto D_R = v.add("D_R", 0);
    const auto MA = v.add("MA", 0);
    const auto MR = v.add("MR", 0);
    const auto A = v.add("A", 0);
    const auto R = v.add("R", 0);
    const auto C = v.add("C", 0);


    v.add((A + DA) >> gammaA >>= D_A);
    v.add(D_A >> thetaA >>= DA + A);
    v.add((A + DR) >> gammaR >>= D_R);
    v.add(D_R >> thetaR >>= DR + A);
    v.add(D_A >> alpha_A >>= MA + D_A);
    v.add(DA >> alphaA >>= MA + DA);
    v.add(D_R >> alpha_R >>= MR + D_R);
    v.add(DR >> alphaR >>= MR + DR);
    v.add(MA >> betaA >>= MA + A);
    v.add(MR >> betaR >>= MR + R);
    v.add((A + R) >> gammaC >>= C);
    v.add(C >> deltaA >>= R);
    v.add(A >> deltaA >>= env);
    v.add(R >> deltaR >>= env);
    v.add(MA >> deltaMA >>= env);
    v.add(MR >> deltaMR >>= env);

    return v;
}
stochastic::Vessel seihr(uint32_t N) {

     auto v = stochastic::Vessel{"COVID19 SEIHR: " + std::to_string(N)};
     const auto eps = 0.0009; // initial fraction of infectious
     const auto I0 = size_t(std::round(eps * N)); // initial infectious
     const auto E0 = size_t(std::round(eps * N * 15)); // initial exposed
     const auto S0 = N - I0 - E0; // initial susceptible
     const auto R0 = 2.4;
     const auto alpha = 1.0 / 5.1; // incubation rate (E -> I) ~5.1 days
     const auto gamma = 1.0 / 3.1; // recovery rate (I -> R) ~3.1 days
     const auto beta = R0 * gamma; // infection/generation rate (S+I -> E+I)
     const auto P_H = 0.9e-3; // probability of hospitalization
     const auto kappa = gamma * P_H * (1.0 - P_H); // hospitalization rate (I -> H)
     const auto tau = 1.0 / 10.12; // removal rate in hospital (H -> R) ~10.12 days
     const auto S = v.add("S", S0); // susceptible
     const auto E = v.add("E", E0); // exposed
     const auto I = v.add("I", I0); // infectious
     const auto H = v.add("H", 0); // hospitalized
     const auto R = v.add("R", 0); // removed/immune (recovered + dead)
     v.add((S + I) >> beta / N >>= E + I); // susceptible becomes exposed by infectious
     v.add(E >> alpha >>= I); // exposed becomes infectious
     v.add(I >> gamma >>= R); // infectious becomes removed
     v.add(I >> kappa >>= H); // infectious becomes hospitalized
     v.add(H >> tau >>= R); // hospitalized becomes removed
     return v;
 }
stochastic::Vessel Figure1_1(){
    auto v = stochastic::Vessel("Fig1_1");
    const auto env = v.global_state.environment;
    const auto A = v.add("A", 100);
    const auto B = v.add("B", 0);
    const auto C = v.add("C", 1);
    const auto lambda = 0.001;
    v.add((A + C) >> lambda >>= B + C);
    return v;
}
stochastic::Vessel Figure1_2(){
    auto v = stochastic::Vessel("Fig1_2");
    const auto env = v.global_state.environment;
    const auto A = v.add("A", 100);
    const auto B = v.add("B", 0);
    const auto C = v.add("C", 2);
    const auto lambda = 0.001;
    v.add((A + C) >> lambda >>= B + C);
    return v;
}
stochastic::Vessel Figure1_3(){
    auto v = stochastic::Vessel("Fig1_3");
    const auto env = v.global_state.environment;
    const auto A = v.add("A", 50);
    const auto B = v.add("B", 50);
    const auto C = v.add("C", 1);
    const auto lambda = 0.001;
    v.add((A + C) >> lambda >>= B + C);
    return v;
}



void HospitalPeak(){
    auto observer = [](double time, stochastic::Vessel &v){
        static int max = 0;
        auto hospitalized = v.global_state.symbolTable.LookUp("H")->second;
        std::cout << "CURRENT HOSPITALIZED: " << hospitalized << "\n";
        if (hospitalized > max){
            max = hospitalized;
            std::cout << "New Peak Hospitalized: " << max << "\n";
        }
    };

    auto NDK = 5822763;
    auto NNJ = 589755;
    std::vector<int> pops = {NDK, NNJ};
    for (auto p:pops) {
        auto vessel = seihr(p);
        auto sim = stochastic::StochasticSimulation();
        sim.RunSimulation(vessel, 100, observer);
    }

}
```

Listing 14: ./Molecule.cpp

```
//
// Created by jjtor on 06/05/2024.
//

#include "Molecule.h"
```

Listing 15: ./PrettyPrinter.cpp

```
//
// Created by jjtor on 31/05/2024.
//

#include "PrettyPrinter.h"
```

Listing 16: ./Reaction.cpp

```
//
// Created by jjtor on 30/05/2024.
//

#include "Reaction.h"
#include "Molecule.h"
```

Listing 17: ./StochasticSimulation.cpp

```
//
// Created by jjtor on 06/05/2024.
//

#include "StochasticSimulation.h"
#include "GlobalState.h"
#include "Molecule.h"
#include "Reaction.h"
#include <algorithm>

//Global variables

namespace stochastic {
//Prototypes
    Reaction FindSmallestDelayReaction(Vessel &vessel);

    std::string cleanString(std::string str) {
        str.erase(std::remove(str.begin(), str.end(), ':'), str.end());
        return str;
    }

    double RandomNumberGen(double delay) {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::exponential_distribution<double> distribution(delay);
        double d = distribution(gen);
        return d;
    }
```

```cpp
    void StochasticSimulation::RunSimulation(Vessel vessel, double end_time) {
        std::string mCount;
        std::string header;
        auto vName = vessel.GetName();
        this->path = "..\\out\\trajectory_" + cleanString(vName) + ".csv";
        trajectory = std::ofstream(path);
        header += "Time,";
        trajectory << header;
        for (auto it = vessel.global_state.symbolTable.table.begin();
             it != vessel.global_state.symbolTable.table.end(); ++it) {
            if (std::next(it) != vessel.global_state.symbolTable.table.end()) {
                trajectory << it->first << ",";
            } else {
                trajectory << it->first << "\n";
            }
        }
        std::cout << "Running simulation. Time: " + std::to_string(vessel.global_state.GetCurrentTime()) + "\n";

        while (vessel.global_state.GetCurrentTime() <= end_time) {
            for (const auto &[key, value]: vessel.global_state.symbolTable.table) {
                mCount += std::to_string(value) + ",";
            }
            mCount.pop_back();
            trajectory << vessel.global_state.GetCurrentTime() << ',' << mCount;
            trajectory << "\n";
            mCount.clear();
            for (auto &r: vessel.GetReactions()) {
                auto delay = ComputeReactionTime(r, vessel);
                r.set_delay(delay);
                //Fix so it is taken directly from symbol table without for-range loop
            }

            // Pick reaction with shortest delay (reaction time)
            auto min_delay_reaction = FindSmallestDelayReaction(vessel);

            vessel.global_state.AddTime(min_delay_reaction.get_current_delay()); //Line 5
            bool valid_reaction = std::all_of(min_delay_reaction.get_reactants().begin(), min_delay_reaction.get_reactants().end(),
                                              [&](Molecule &i) {
                                                  return vessel.global_state.symbolTable.LookUp(i.GetName())->second > 0;
                                              });
            for (auto &q: min_delay_reaction.get_reactants()) {
                if (valid_reaction) {
                    vessel.global_state.symbolTable.Update(q.GetName(), -1);
                }
            }
            if (valid_reaction){
                for (auto &p: min_delay_reaction.get_products()) {
                    vessel.global_state.symbolTable.Update(p.GetName(), 1);
                }
            }
            std::cout << "Simulation step done. Time: " + std::to_string(vessel.global_state.GetCurrentTime()) + "\n";
        }
        std::cout << "Simulation done. Time: " + std::to_string(vessel.global_state.GetCurrentTime()) << "\n";

    }

    template<typename Obs>
    void StochasticSimulation::RunSimulationParallel(Vessel vessel, double end_time, int numberOfSims, Obs observer) {
        std::vector<std::thread> threads;
        std::vector<StochasticSimulation> simulations(numberOfSims);

        for (int i = 0; i < numberOfSims; ++i) {
            threads.emplace_back(&StochasticSimulation::RunSimulation, &simulations[i], std::ref(vessel), end_time, observer);
        }

        for (auto &thread: threads) {
            thread.join();
        }

    }
    void StochasticSimulation::RunSimulationParallel(Vessel vessel, double end_time, int numberOfSims) {
        std::vector<std::thread> threads;
        std::vector<StochasticSimulation> simulations(numberOfSims);

        for (int i = 0; i < numberOfSims; ++i) {
            threads.emplace_back(static_cast<void (StochasticSimulation::*)(Vessel, double)>(&StochasticSimulation::RunSimulation), &simulations[i], std::ref(vessel), end_time);
        }

        for (auto &thread: threads) {
            thread.join();
        }
    }

    double StochasticSimulation::ComputeReactionTime(Reaction &reaction, Vessel &vessel) {
        double total_amount_of_reactants = 1.0;
        for (const auto &m: reaction.get_reactants()) {
            if (vessel.global_state.symbolTable.LookUp(m.GetName())->second <= 0) {
                return std::numeric_limits<double>::infinity();
            }
            total_amount_of_reactants *= vessel.global_state.symbolTable.LookUp(m.GetName())->second;
        }
        return RandomNumberGen(reaction.get_current_rate_parameter() * total_amount_of_reactants);
    }

    Reaction FindSmallestDelayReaction(Vessel &vessel) {
        auto min_delay = std::numeric_limits<double>::infinity();
        auto min_delay_reaction = Reaction();
        for (auto &r: vessel.GetReactions()) {
            if (r.get_current_delay() < min_delay) {
                min_delay = r.get_current_delay();
                min_delay_reaction = r;
            }
        }
        return min_delay_reaction;
    }


    Reaction Molecule::operator+(Molecule molecule) const {
        auto r = Reaction();
        r.add_reactant(molecule);
        r.add_reactant(*this);
        return r;
    }


    Reaction Molecule::operator>>(double rate) const {
        auto r = Reaction();
        r.add_reactant(*this);
        r.set_rate_parameter(rate);
        return r;
    }

    Reaction Molecule::operator+(Reaction reaction) {
        auto r = Reaction();
        for (auto &reactant: reaction.get_reactants()) {
            r.add_reactant(reactant);
        }
        for (auto &product: reaction.get_products()) {
            r.add_product(product);
        }
        r.set_rate_parameter(reaction.get_current_rate_parameter());
        return r;
    };
}
```

Listing 18: ./Test/Test.cpp

```
//
// Created by jjtor on 12/06/2024.
//

#include "doctest.h"
#include "../StochasticSimulation.h"

namespace stochastic{
    TEST_CASE("Test SymbolTable") {
            auto symbolTable = GlobalState().symbolTable;
        SUBCASE("LookUpSuccess"){
            symbolTable.Update("A", 1);
            CHECK((symbolTable.LookUp("A")->second == 1));
        }
        SUBCASE("LookUpSymbolNotExist"){
            CHECK((symbolTable.LookUp("A") == symbolTable.table.end()));
        }
    }

    TEST_CASE("Pretty Print"){
        auto r = Reaction();
        r.add_reactant(Molecule("M1", 1));
        r.add_reactant(Molecule("M2", 4));
        r.add_product(Molecule("M3", 0));
        r.set_rate_parameter(1);
        Vessel v = Vessel("TestVessel");
        v.add(r);
        std::stringstream prettyPrint;
        prettyPrint << v.GetReactions();
        CHECK((prettyPrint.str() == "Reactants: [M1 M2 \b \b] Rate parameter: [1] Products: [M3 \b \b]\n"));
    }
}
```

Listing 19: ./Vessel.cpp

```
//
```