SP Exam Project

Mathias Andresen

May 13, 2021

Compiled with Visual Studio 2019 Community (version 16) in Clion. Using bundled CMake 3.17.5.

Listing 1: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.17)
   project(sp_exam_project)
   set(CMAKE_CXX_STANDARD 20)
   add_library(
       stochastic-simulation
       library/simulation.cpp
       library/simulation.cpp
       library/SymbolTable.h
10
       library/simulation_monitor.h
       library/data.h
12
       library/data.cpp
13
       library/my-thread-pool.h
14
   )
15
16
   add_executable(sp_exam_project main.cpp vessels.h)
17
   target_link_libraries(sp_exam_project PRIVATE stochastic-simulation)
```

Listing 2: main.cpp

```
#include <iostream>
   #include "library/simulation.h"
   #include <chrono>
   #include "vessels.h"
   using namespace StochasticSimulation;
   // Requirement 7 use of monitor
   class hospitalized_monitor: public simulation_monitor {
   private:
       double_t hospitalized_acc{0.0};
       double_t last_time{0.0};
12
13
       size_t max_hospitalized{0};
       void monitor(SimulationState &state) override {
16
           auto currently_hospitalized = state.reactants.get("H").amount;
17
           if (currently_hospitalized > max_hospitalized) {
               max_hospitalized = currently_hospitalized;
           }
           hospitalized_acc += (currently_hospitalized * (state.time - last_time));
24
           last_time = state.time;
25
       }
```

```
27
       double_t get_mean_hospitalized() const {
            return (hospitalized_acc / last_time);
29
       }
30
   };
31
32
   void simulate_covid() {
33
       std::cout << "Simulating covid19 example with hospitalized monitor" << std::endl;</pre>
34
       Vessel covid_vessel = seihr(10000);
36
       std::cout << covid_vessel << std::endl;</pre>
37
        covid_vessel.visualize_reactions("covid_graph.png");
38
       std::cout << "reaction graph can be seen at: covid_graph.png" << std::endl;</pre>
40
       hospitalized_monitor monitor{};
       auto trajectory = covid_vessel.do_simulation(120, monitor);
44
45
       std::cout << "Simulation done" << std::endl;</pre>
46
       std::cout << "Max hospitalized: " << monitor.max_hospitalized << std::endl;</pre>
       std::cout << "Mean hospitalized: " << monitor.get_mean_hospitalized() << std::endl;</pre>
48
       std::cout << "Writing trajectory to csv file at covid_output.csv" << std::endl;</pre>
       trajectory->write_csv("covid_output.csv");
       std::cout << "Turn it into a graph using python ./draw_graph.py covid release" << std::endl;</pre>
52
   }
53
   void simulate_covid_multiple() {
55
       std::cout << "Simulating covid19 example 30 times and calculating mean" << std::endl;</pre>
56
       Vessel covid_vessel = seihr(10000);
57
       auto trajectories = covid_vessel.do_multiple_simulations(110, 100);
59
60
       std::cout << "Simulations done" << std::endl << "Computing mean trajectory" << std::endl;</pre>
61
       auto mean = SimulationTrajectory::compute_mean_trajectory(trajectories);
63
       std::cout << "Writing mean trajectory to csv file at covid_output_multiple.csv" << std::endl;
       mean.write_csv("covid_output_multiple.csv");
       std::cout << "Turn it into a graph using python ./draw_graph.py covid
 →covid_output_multiple.csv" << std::endl;
   }
68
   void simulate_introduction() {
70
       std::cout << "Simulating introduction example" << std::endl;</pre>
71
       Vessel introduction_vessel = introduction(25, 50, 1, 0.001);
       std::cout << introduction_vessel << std::endl;</pre>
73
74
       introduction_vessel.visualize_reactions("intro_graph.png");
75
       auto trajectory = introduction_vessel.do_simulation(400);
       trajectory->write_csv("intro_output.csv");
79
   }
80
81
82
   void simulate_circadian() {
83
       std::cout << "Simulating circadian rhythm example..." << std::endl;</pre>
       Vessel oscillator = circadian_oscillator();
85
86
```

```
std::cout << oscillator << std::endl;</pre>
        oscillator.visualize_reactions("cir_graph.png");
89
        auto trajectory = oscillator.do_simulation(110);
90
91
        std::cout << "Writing csv file..." << std::endl;</pre>
92
        trajectory->write_csv("circadian_output.csv");
93
    }
94
    void simulate_circadian2() {
96
        std::cout << "Simulating circadian rhythm alternative example..." << std::endl;</pre>
97
        Vessel oscillator = circadian_oscillator2();
98
        std::cout << oscillator << std::endl;</pre>
100
        oscillator.visualize_reactions("cir2_graph.png");
101
        auto trajectory = oscillator.do_simulation(110);
103
104
        std::cout << "Writing csv file..." << std::endl;</pre>
105
106
        trajectory->write_csv("circadian2_output.csv");
    }
107
108
    void benchmark() {
109
        std::cout << "Benchmarking with circadian rhythm example (max_time=100)" << std::endl;</pre>
110
111
        auto runs{30};
112
113
        Vessel oscillator = circadian_oscillator();
114
115
        unsigned long time_acc1{0};
116
        for (int i = 0; i < runs; ++i) {</pre>
117
             auto t0 = std::chrono::high_resolution_clock::now();
             oscillator.do_simulation(100);
119
             auto t1 = std::chrono::high_resolution_clock::now();
120
121
             time_acc1 += std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
        }
123
        auto mean_time1 = time_acc1 / runs;
124
        std::cout << "Simulation 1 mean time (nanoseconds): " << mean_time1 << std::endl;</pre>
125
        unsigned long time_acc2{0};
127
        for (int i = 0; i < runs; ++i) {</pre>
128
             auto t0 = std::chrono::high_resolution_clock::now();
129
             oscillator.do_simulation2(100);
130
             auto t1 = std::chrono::high_resolution_clock::now();
131
132
             time_acc2 += std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
134
        auto mean_time2 = time_acc2 / runs;
135
        std::cout << "Simulation 2 mean time (nanoseconds): " << mean_time2 << std::endl;</pre>
136
    }
137
138
    int main() {
139
        simulate_covid();
140
    //
          simulate_covid_multiple();
141
142
          simulate_introduction();
   //
143
   //
          simulate_circadian();
144
          simulate_circadian2();
   //
145
146
  //
          benchmark();
147
```

.48 }

Listing 3: vessels.h

```
// Created by Mathias on 12-05-2021.
   #ifndef SP_EXAM_PROJECT_VESSELS_H
   #define SP_EXAM_PROJECT_VESSELS_H
   #include "library/simulation.h"
   using namespace StochasticSimulation;
10
11
   Vessel seihr(uint32_t N)
   {
13
       auto v = Vessel{};
14
       const auto eps = 0.0009; // initial fraction of infectious
15
       const auto I0 = size_t(std::round(eps*N)); // initial infectious
16
       const auto E0 = size_t(std::round(eps*N*15)); // initial exposed
17
       const auto S0 = N-I0-E0; // initial susceptible
18
       const auto R0 = 2.4; // basic reproductive number (initial, without lockdown etc)
19
       const auto alpha = 1.0 / 5.1; // incubation rate (E -> I) ~5.1 days
       const auto gamma = 1.0 / 3.1; // recovery rate (I \rightarrow R) ~3.1 days
21
       const auto beta = R0 * gamma; // infection/generation rate (S+I \rightarrow 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; // recovery/death rate in hospital (H -> R) ~10.12 days
25
26
       // Reactants
27
       auto S = v("S", S0); // susceptible
28
       auto E = v("E", E0); // exposed
29
       auto I = v("I", I0); // infectious
30
       auto H = v("H", 0); // hospitalized
       auto R = v("R", 0); // removed/immune (recovered + dead)
32
33
       // Reactions
34
       v(S >= E, I, beta/N);
       v(E >>= I, alpha);
36
       v(I >>= R, gamma);
37
       v(I >>= H, kappa);
       v(H >>= R, tau);
40
       return v;
41
   }
42
   Vessel introduction(uint32_t A_start, uint32_t B_Start, uint32_t D_amount, double_t lambda) {
44
       auto v = Vessel{};
45
       // Reactants
46
       auto A = v("A", A_start);
       auto B = v("B", B_Start);
48
       auto C = v("C", 0);
49
       auto D = v("D", D_amount);
       // Reactions
51
       v(A + B * 2 >>= C, D, lambda);
52
       return v;
   }
55
56
  /** direct encoding */
  Vessel circadian_oscillator()
```

```
{
        auto alphaA = 50.0;
        auto alpha_A = 500.0;
61
        auto alphaR = 0.01;
62
        auto alpha_R = 50.0;
63
        auto betaA = 50.0;
64
        auto betaR = 5.0;
65
        auto gammaA = 1.0;
66
        auto gammaR = 1.0;
        auto gammaC = 2.0;
68
        auto deltaA = 1.0;
69
        auto deltaR = 0.2;
70
        auto deltaMA = 10.0;
71
        auto deltaMR = 0.5;
72
        auto thetaA = 50.0;
73
        auto thetaR = 100.0;
        auto v = Vessel{};
75
        auto env = v.environment();
76
        auto DA = v("DA", 1);
77
        auto D_A = v("D_A", \theta);
78
        auto DR = v("DR", 1);
        auto D_R = v("D_R", 0);
80
        auto MA = v("MA", 0);
81
        auto MR = v("MR", 0);
        auto A = v("A", 0);
83
        auto R = v("R", 0);
84
        auto C = v("C", 0);
85
        v(A + DA >>= D_A, gammaA);
        v(D_A \gg DA + A, thetaA);
87
        v(A + DR >>= D_R, gammaR);
88
        v(D_R \gg DR + A, thetaR);
89
        v(D_A \gg MA + D_A, alpha_A);
        v(DA >>= MA + DA, alphaA);
91
        v(D_R \gg MR + D_R, alpha_R);
92
        v(DR >>= MR + DR, alphaR);
93
        v(MA >>= MA + A, betaA);
        v(MR \gg MR + R, betaR);
95
        v(A + R >>= C, gammaC);
96
        v(C >>= R, deltaA);
97
        v(A >>= env, deltaA);
        v(R >= env, deltaR);
99
        v(MA >>= env, deltaMA);
100
        v(MR >>= env, deltaMR);
101
        return v;
102
    }
103
104
    /** alternative encoding using catalysts */
105
    Vessel circadian_oscillator2()
106
    {
107
        auto alphaA = 50.0;
108
        auto alpha_A = 500.0;
109
        auto alphaR = 0.01;
110
        auto alpha_R = 50.0;
111
        auto betaA = 50.0;
112
        auto betaR = 5.0;
        auto gammaA = 1.0;
114
        auto gammaR = 1.0;
115
        auto gammaC = 2.0;
116
        auto deltaA = 1.0;
117
        auto deltaR = 0.2;
118
        auto deltaMA = 10.0;
119
```

```
auto deltaMR = 0.5;
120
        auto thetaA = 50.0;
121
        auto thetaR = 100.0;
122
        auto v = Vessel{};
123
        auto env = v.environment();
124
        auto DA = v("DA", 1);
125
        auto D_A = v("D_A", 0);
126
        auto DR = v("DR", 1);
        auto D_R = v("D_R", 0);
128
        auto MA = v("MA", 0);
129
        auto MR = v("MR", 0);
130
        auto A = v("A", 0);
131
        auto R = v("R", 0);
132
        auto C = v("C", 0);
133
        v(A + DA >>= D_A, gammaA);
134
        v(D_A >>= DA + A, thetaA);
135
        v(DR + A >>= D_R, gammaR);
136
        v(D_R >>= DR + A, thetaR);
137
        v(env >>= MA, D_A, alpha_A);
138
        v(env >>= MA, DA, alphaA);
139
        v(env >>= MR, D_R, alpha_R);
        v(env >>= MR, DR, alphaR);
141
        v(env >>= A, MA, betaA);
142
        v(env >>= R, MR, betaR);
        v(A + R >>= C, gammaC);
144
        v(C >>= R, deltaA);
145
        v(A >>= env, deltaA);
146
        v(R >>= env, deltaR);
147
        v(MA >>= env, deltaMA);
148
        v(MR >>= env, deltaMR);
149
        return v;
150
151
152
    #endif //SP_EXAM_PROJECT_VESSELS_H
153
```

Listing 4: simulation.h

```
// Created by Mathias on 09-05-2021.
#ifndef SP_EXAM_PROJECT_SIMULATION_H
#define SP_EXAM_PROJECT_SIMULATION_H
#include <string>
#include <utility>
#include <vector>
#include <set>
#include <optional>
#include <map>
#include <numeric>
#include <random>
#include <algorithm>
#include <functional>
#include <sstream>
#include <fstream>
#include <chrono>
#include <thread>
#include <future>
#include <ranges>
#include "SymbolTable.h"
#include "simulation_monitor.h"
```

```
#include "data.h"
   namespace StochasticSimulation {
28
29
       using map_type = std::map<double_t, SimulationState>;
30
       class SimulationTrajectory: public map_type {
31
       private:
32
           double_t largest_time{-1};
            static double_t compute_interpolated_value(
                    const std::string& key,
35
                    SimulationState& s0,
36
                    SimulationState& s1,
37
                    double_t x);
       public:
39
           using map_type::map;
40
           SimulationTrajectory(const SimulationTrajectory& val): map_type(val) {
                largest_time = val.largest_time;
43
           }
44
45
           SimulationTrajectory(SimulationTrajectory&& rval): map_type(std::move(rval)) {
                largest_time = std::move(rval.largest_time);
           };
           SimulationTrajectory& operator=(const SimulationTrajectory & val) {
                map_type::operator=(val);
51
                largest_time = val.largest_time;
52
           };
54
           SimulationTrajectory& operator=(SimulationTrajectory&& rval) {
55
                map_type::operator=(std::move(rval));
                largest_time = std::move(rval.largest_time);
           };
58
59
           // Requirement 9 compute mean
60
           static SimulationTrajectory
  →compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>& trajectories);
62
           void insert(SimulationState state) {
63
                if (state.time > largest_time) {
                    largest_time = state.time;
65
                }
66
67
                map_type::insert({state.time, std::move(state)});
           }
69
           // Requirement 6 output trajectory
           void write_csv(const std::string& path);
73
           double_t get_max_time() {
74
                return largest_time;
           }
76
       };
       // Requirement 1 operators for DSEL
       class Vessel {
80
       private:
81
           std::vector<Reaction> reactions{};
82
           SymbolTable<Reactant> reactants;
       public:
84
85
```

```
Vessel() = default;
            Vessel(const Vessel &val) {
88
                 reactions = val.reactions;
89
                 reactants = val.reactants;
            }
            Vessel (Vessel&& rval) {
                 reactions = std::move(rval.reactions);
                 reactants = std::move(rval.reactants);
95
            };
96
97
            Reactant& operator()(std::string name, size_t initial_amount) {
                 Reactant newReactant{std::move(name), initial_amount};
99
100
                 reactants.put(newReactant.name, newReactant);
102
                 return reactants.get(newReactant.name);
103
            }
104
105
            Reaction operator()(Reaction& reaction, double_t rate) {
                 reaction.rate = rate;
107
108
                 reactions.push_back(reaction);
110
                 return reaction;
111
            }
112
113
            Reaction operator()(Reaction&& reaction, std::initializer_list<Reactant> catalysts,
  →double rate) {
                 reaction.rate = rate;
115
                 reaction.catalysts = catalysts;
116
117
                 // Add to vessel reactions
118
                 reactions.push_back(reaction);
119
120
                 return reaction;
121
            }
122
            Reaction operator()(Reaction&& reaction, Reactant catalyst, double_t rate) {
                 reaction.rate = rate;
125
                 reaction.catalysts = {catalyst};
126
127
                 // Add to vessel reactions
128
                 reactions.push_back(reaction);
129
130
                 return reaction;
            }
132
133
134
            Reactant& environment() {
                 if (reactants.contains("__env__")) {
136
                    return reactants.get("__env__");
137
                 }
                 auto newReactant = Reactant("__env__", 0, 0);
140
141
                 reactants.put(newReactant.name, newReactant);
142
143
                 return reactants.get(newReactant.name);
144
            }
145
```

```
// Requirement 2 network graph
            void visualize_reactions(const std::string& filename);
148
149
            // Requirement 10 optimized algorithm
150
            std::shared_ptr<SimulationTrajectory> do_simulation2(double_t end_time,
151
   →simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);
152
153
            // Requirement 4 simulation
            std::shared_ptr<SimulationTrajectory> do_simulation(double_t end_time,
154
  →simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);
155
            // Requirement 8 parallelization
156
            std::vector<std::shared_ptr<SimulationTrajectory>> do_multiple_simulations(double_t
157
  →end_time, size_t simulations_to_run);
158
            // Requirement 2 pretty print
159
            friend std::ostream& operator<<(std::ostream& s, const Vessel& vessel);</pre>
160
        };
161
162
164
    }
165
166
    #endif //SP_EXAM_PROJECT_SIMULATION_H
```

Listing 5: simulation.cpp

```
// Created by Mathias on 09-05-2021.
   //
   #include <iostream>
   #include <utility>
   #include "simulation.h"
   namespace StochasticSimulation {
9
10
       // Requirement 2
12
       std::ostream &operator<<(std::ostream &s, const Vessel &vessel) {</pre>
13
            s << "{" << std::endl;
            for (const auto& reaction: vessel.reactions) {
                s << "\t" << reaction;
16
                if (&reaction != &vessel.reactions.back()) {
17
                    s << ",";
                }
                s << std::endl;
20
            }
21
            return s << "}";
       }
23
24
       // Requirement 2
25
       void Vessel::visualize_reactions(const std::string& filename) {
            std::stringstream str;
27
            SymbolTable<std::string> node_map{};
            str << "digraph {" << std::endl;</pre>
            auto i = 0;
32
            for (auto& reactant: reactants.getMap()) {
33
                if (reactant.second.name != "__env__") {
```

```
node_map.put(reactant.second.name, "s" + std::to_string(i));
                    str << node_map.get(reactant.second.name)</pre>
37
                         << "[label=\"" << reactant.second.name <<</pre>
 \rightarrow "\", shape=\"box\", style=\"filled\", fillcolor=\"cyan\"];" << std::endl;
39
                }
40
           }
41
            i = 0;
43
            for (auto& reaction: reactions) {
44
                std::string reaction_node{"r" + std::to_string(i)};
45
                str << reaction_node << "[label=\"" << reaction.rate <<</pre>
  →"\",shape=\"oval\",style=\"filled\",fillcolor=\"yellow\"];" << std::endl;</pre>
                if (reaction.catalysts.has_value()) {
48
                    for (auto& catalyst: reaction.catalysts.value()) {
49
                         str << node_map.get(catalyst.name) << " -> " << reaction_node << "</pre>
50
 →[arrowhead=\"tee\"];" << std::endl;</pre>
                    }
51
                }
                for (auto& reactant: reaction.from) {
53
                    if (reactant.name != "__env__") {
54
                         str << node_map.get(reactant.name) << " -> " << reaction_node << ";" <<</pre>
 →std::endl;
                    }
56
                }
57
                for (auto& product: reaction.to) {
                    if (product.name != "__env__") {
59
                         str << reaction_node << " -> " << node_map.get(product.name) << ";" <<</pre>
 →std::endl;
                    }
                }
62
63
                i++;
64
            }
66
            str << "}";
            std::ofstream dotfile;
            dotfile.open(filename + ".dot");
70
            dotfile << str.str();</pre>
71
           dotfile.close();
72
            std::stringstream command_builder;
74
            command_builder << "dot -Tpnq -o " << filename << " " << filename << ".dot";</pre>
75
            system(command_builder.str().c_str());
77
78
       // Requirement 10 alternative simulation
79
       std::shared_ptr<SimulationTrajectory> Vessel::do_simulation2(double_t end_time, 🗸
  →simulation_monitor &monitor) {
            SimulationTrajectory trajectory{};
81
           double_t t{0};
82
            auto thread_id = std::this_thread::get_id();
84
            auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
85
            std::default_random_engine engine(epoch * (std::hash<std::thread::id>{}(thread_id)));
86
            // Insert initial state
            trajectory.insert(SimulationState{reactants, t});
89
```

```
while (t <= end_time) {</pre>
                 for (Reaction& reaction: reactions) {
92
                     // New: using new compute delay function
93
                     reaction.compute_delay2(trajectory.at(t), engine);
                 }
                 auto r = reactions.front();
                 // Select Reaction with min delay which is not -1
99
                 for (auto& reaction: reactions) {
100
                     if (reaction.delay == -1) {
101
                         continue;
102
                     } else if (reaction.delay < r.delay) {</pre>
103
                         r = reaction;
104
                     } else if (r.delay == -1) {
                         r = reaction;
106
                     }
107
                 }
108
109
                 // Stop if we have no reactions to do, thus r.delay == -1
                 if (r.delay == -1) {
111
                     break;
112
                 }
                 auto& last_state = trajectory.at(t);
115
116
                 t += r.delay;
117
118
                 SimulationState state{last_state.reactants, t};
119
120
                         std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return
122
  →state.reactants.get(e.name).amount >= e.required;}) &&
123
                         (
                                  !r.catalysts.has_value() ||
124
                                  std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(),
125
   →[&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
                         )
126
                         ) {
                     for (auto& reactant: r.from) {
128
                         state.reactants.get(reactant.name).amount -= reactant.required;
129
130
                     for (auto& reactant: r.to) {
131
                         state.reactants.get(reactant.name).amount += reactant.required;
132
                     }
133
                 }
135
                 trajectory.insert(std::move(state));
136
137
                 monitor.monitor(trajectory.at(t));
            }
139
140
             return std::make_shared<SimulationTrajectory>(std::move(trajectory));
141
        }
143
        // Requirement 4 simulation
144
        std::shared_ptr<SimulationTrajectory> Vessel::do_simulation(double_t end_time,
145
  →simulation_monitor &monitor) {
            SimulationTrajectory trajectory{};
146
            double_t t{0};
147
```

```
auto thread_id = std::this_thread::get_id();
             auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
150
            std::default_random_engine engine(epoch * (std::hash<std::thread::id>{}(thread_id)));
151
152
            // Insert initial state
153
            trajectory.insert(SimulationState{reactants, t});
154
            while (t <= end_time) {</pre>
                 for (Reaction& reaction: reactions) {
157
                     reaction.compute_delay(trajectory.at(t), engine);
158
                 }
159
160
                 auto r = reactions.front();
161
162
                 // Select Reaction with min delay which is not -1
                 for (auto& reaction: reactions) {
164
                     if (reaction.delay == -1) {
165
                         continue;
166
167
                     } else if (reaction.delay < r.delay) {</pre>
                         r = reaction;
                     } else if (r.delay == -1 && reaction.delay != -1) {
169
                         r = reaction;
170
                     }
                 }
173
                 // Stop if we have no reactions to do, thus r.delay == -1
174
                 if (r.delay == -1) {
175
                     break;
176
                 }
177
                 auto& last_state = trajectory.at(t);
180
                 t += r.delay;
181
182
                 SimulationState state{last_state.reactants, t};
183
184
                 if (
185
                         std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return
  →state.reactants.get(e.name).amount >= e.required;}) &&
187
                                  !r.catalysts.has_value() ||
188
                                  std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(),
  →[&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
190
                         ) {
191
                     for (auto& reactant: r.from) {
                         state.reactants.get(reactant.name).amount -= reactant.required;
193
194
                     for (auto& reactant: r.to) {
195
                         state.reactants.get(reactant.name).amount += reactant.required;
                     }
197
                 }
198
                 trajectory.insert(std::move(state));
201
                 monitor.monitor(trajectory.at(t));
202
            }
203
204
             return std::make_shared<SimulationTrajectory>(std::move(trajectory));
205
        }
206
```

```
// Requirement 8 multiple at same time
        std::vector<std::shared_ptr<SimulationTrajectory>>
209
        Vessel::do_multiple_simulations(double_t end_time, size_t simulations_to_run) {
210
            std::vector<std::shared_ptr<SimulationTrajectory>> result{};
211
            result.reserve(simulations_to_run);
212
213
            auto cores = std::thread::hardware_concurrency();
            int jobs = std::min(simulations_to_run, (cores - 1));
            auto simulations_per_job = simulations_to_run / jobs;
216
217
            auto futures =
218
  →std::vector<std::future<std::vector<std::shared_ptr<SimulationTrajectory>>>>{};
219
            auto lambda = [&vessel = *this, &end_time](size_t to_run){
220
                auto simulations = std::vector<std::shared_ptr<SimulationTrajectory>>{};
                simulations.reserve(to_run);
222
223
                auto new_vessel = Vessel(vessel);
224
225
                for (int i = 0; i < to_run; ++i) {</pre>
                     simulations.push_back(new_vessel.do_simulation(end_time));
227
                }
228
                 return simulations;
230
            };
231
232
            for (int i = 0; i < jobs; ++i) {
                futures.push_back(std::async(std::launch::async, lambda, simulations_per_job));
234
235
            auto missing_simulations = simulations_to_run - (simulations_per_job * jobs);
236
            if (missing_simulations != 0) {
                 futures.push_back(std::async(std::launch::async, lambda, missing_simulations));
238
239
240
            for (auto& future: futures) {
241
                auto future_result = future.get();
242
                for (auto& res: future_result) {
243
                     result.push_back(std::move(res));
                }
            }
246
247
            return result;
248
        }
249
250
        // Private function for calculation interpolated value
251
        double_t SimulationTrajectory::compute_interpolated_value(const std::string& key,
   SimulationState& s0, SimulationState& s1, double_t x) {
253
                s0.reactants.get(key).amount
254
                + ((
255
                        ( (double_t) s1.reactants.get(key).amount - (double_t)
256
   →s0.reactants.get(key).amount) /
                        ( s1.time - s0.time )
257
                    ) * (x - s0.time));
        }
259
260
        // Requirement 9 compute mean trajectory
261
        SimulationTrajectory
262
  →SimulationTrajectory::compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>&
  →trajectories) {
```

```
auto average_delay = trajectories.front()->get_max_time() / trajectories.front()->size();
             // Get a list of all keys
265
             std::vector<std::string> reactant_keys{};
266
             for (auto& reactant: trajectories.front()->begin()->second.reactants) {
267
                 reactant_keys.push_back(reactant.second.name);
268
             }
269
270
            // Find upper bound for mean trajectory
             double_t upper_bound{-1.0};
272
             for (auto& trajectory: trajectories) {
273
                 if (upper_bound == -1.0 || trajectory->get_max_time() < upper_bound) {</pre>
274
                     upper_bound = trajectory->get_max_time();
                 }
276
             }
277
             SimulationTrajectory mean_trajectory{};
280
             for (auto& trajectory: trajectories) {
281
282
                 auto iterator = trajectory->begin();
                 SimulationState& s0 = iterator->second;
                 iterator++;
284
                 SimulationState& s1 = iterator->second;
285
                 double_t t{0};
287
288
                 while((t + average_delay) <= upper_bound) {</pre>
289
                     if (t \ge s0.time) {
                          if (t <= s1.time) {</pre>
291
                              if (!mean_trajectory.contains(t)) {
292
                                  mean_trajectory.insert((SimulationState{{}, t}));
293
                              }
295
                              for (auto& key: reactant_keys) {
296
                                  auto interpolated_value =
297
  →SimulationTrajectory::compute_interpolated_value(key, s0, s1, t);
298
                                  auto& table = mean_trajectory.at(t).reactants;
299
300
                                  if (!table.contains(key)) {
                                       Reactant reactant{key, 0.0};
302
                                       table.put(key, reactant);
303
                                  }
304
305
                                  table.get(key).amount += interpolated_value;
306
                              }
307
                              t += average_delay;
309
                          } else {
310
                              s0 = s1;
311
                              iterator++;
313
                              if (iterator != trajectory->end()) {
314
                                  s1 = iterator->second;
315
                              } else {
316
                                  break:
317
                              }
318
                         }
319
                     }
                 }
321
             }
322
```

```
for (double_t i = 0; (i + average_delay) <= upper_bound; i += average_delay) {</pre>
                 for (auto& key: reactant_keys) {
325
                      mean_trajectory.at(i).reactants.get(key).amount /= trajectories.size();
326
                 }
             }
328
329
             return std::move(mean_trajectory);
330
331
        }
332
        // Requirement 6 output to csv which can then be turned into a graph via python script
333
        void SimulationTrajectory::write_csv(const std::string &path) {
334
             std::ofstream csv_file;
335
             csv_file.open(path);
336
337
             auto reactants = at(0).reactants;
339
             for (auto& reactant : reactants) {
340
                 csv_file << reactant.second.name << ",";</pre>
341
342
             }
             csv_file << "time" << std::endl;</pre>
344
             for (auto& state : *this) {
345
                 for (auto& reactant: reactants) {
                      csv_file << state.second.reactants.get(reactant.second.name).amount << ",";</pre>
347
348
                 csv_file << state.second.time << std::endl;</pre>
349
             }
351
             csv_file.close();
352
        }
353
```

Listing 6: data.h

```
// Created by Mathias on 11-05-2021.
   #ifndef SP_EXAM_PROJECT_DATA_H
   #define SP_EXAM_PROJECT_DATA_H
6
   namespace StochasticSimulation {
       class SimulationState;
       struct Reaction;
10
       class ReactantCollection;
11
       struct Reactant {
13
           std::string name;
14
           double_t amount; // double to allow for mean values
           size_t required{1};
16
17
           Reactant(std::string name, size_t initial_amount):
18
                    name(std::move(name)),
                    amount(initial_amount)
20
           {}
           Reactant(std::string name, double_t initial_amount):
                    name(std::move(name)),
                    amount(initial_amount)
25
           {}
26
```

```
Reactant(std::string name, size_t initial_amount, size_t required):
                    name(std::move(name)),
                    amount(initial_amount),
30
                    required(required)
31
           {}
32
33
           ~Reactant() = default;
           Reactant(const Reactant& a) {
                name = a.name;
37
                amount = a.amount;
38
                required = a.required;
39
           }
41
           Reactant(Reactant&& a) {
42
                name = std::move(a.name);
                amount = std::move(a.amount);
                required = std::move(a.required);
45
           }
46
47
           Reactant& operator=(Reactant&& a) {
                name = std::move(a.name);
49
                amount = std::move(a.amount);
                required = std::move(a.required);
                return *this;
53
           }
54
           Reactant& operator=(const Reactant& a) {
56
                name = a.name;
57
                amount = a.amount;
                required = a.required;
60
                return *this;
61
           }
62
           // Requirement 1 operator for DSEL
           Reaction operator>>=(Reactant other);
           // Requirement 1 operator for DSEL
           Reaction operator>>=(ReactantCollection other);
68
69
           // Requirement 1 operator for DSEL
70
           ReactantCollection operator+(const Reactant& other);
72
           bool operator<(const Reactant& other) const;</pre>
           // Requirement 1 operator for DSEL
75
           Reactant operator*(size_t req) {
76
                required = req;
77
                return *this;
           }
80
       };
       class ReactantCollection: public std::set<Reactant> {
83
       public:
84
           using std::set<Reactant>::set;
85
           // Requirement 1 operator for DSEL
           Reaction operator>>=(Reactant other);
87
           // Requirement 1 operator for DSEL
88
```

```
Reaction operator>>=(ReactantCollection other);
        };
91
        class Reaction {
92
        public:
93
             std::set<Reactant> from;
             std::set<Reactant> to;
95
             std::optional<std::vector<Reactant>> catalysts;
             double_t rate{};
             double_t delay{-1};
98
99
            Reaction(std::set<Reactant> from, std::set<Reactant> to):
100
                     from(from),
101
                     to(to)
102
             {}
103
104
            Reaction(std::set<Reactant> from, std::set<Reactant> to, std::initializer_list<Reactant>
  →catalysts, double rate):
                     from(from),
106
107
                     to(to),
                     catalysts(catalysts),
108
                     rate(rate)
109
             {}
110
            Reaction(std::set<Reactant> from, std::set<Reactant> to, double rate):
                     from(from),
113
                     to(to),
114
                     catalysts{},
                     rate(rate)
116
             {}
117
             void compute_delay(SimulationState& state, std::default_random_engine& engine);
             void compute_delay2(SimulationState& state, std::default_random_engine& engine);
120
121
             friend std::ostream &operator<<(std::ostream &s, const Reaction &reaction);</pre>
122
        };
123
124
        struct SimulationState {
125
        public:
126
             SymbolTable<Reactant> reactants;
             double_t time;
128
129
             SimulationState(SymbolTable<Reactant> reactants, double_t time):
130
                 reactants{reactants},
131
                 time{time}
132
             {};
133
             SimulationState(const SimulationState&) = default;
135
             SimulationState(SimulationState&&) = default;
136
137
             SimulationState& operator=(const SimulationState &) = default;
             SimulationState& operator=(SimulationState&&) = default;
139
140
            ~SimulationState() = default;
141
             friend std::ostream &operator<<(std::ostream &, const SimulationState &);</pre>
143
        };
144
145
    }
146
147
    #endif //SP_EXAM_PROJECT_DATA_H
148
```

```
//
   // Created by Mathias on 11-05-2021.
3
   #include <iostream>
   #include <utility>
   #include "simulation.h"
   namespace StochasticSimulation {
       Reaction Reactant::operator>>=(StochasticSimulation::Reactant other) {
10
            return Reaction{{*this}, {std::move(other)}};
11
       }
12
       Reaction Reactant::operator>>=(ReactantCollection other) {
14
            return Reaction{{*this}}, std::move(other)};
15
16
       ReactantCollection Reactant::operator+(const Reactant& other) {
18
           return ReactantCollection{*this, other};
19
       }
20
21
       bool Reactant::operator<(const Reactant& other) const {</pre>
22
           return name < other.name;</pre>
23
       }
       Reaction ReactantCollection::operator>>=(Reactant other) {
26
            return Reaction{*this, {std::move(other)}};
27
       }
28
       Reaction ReactantCollection::operator>>=(ReactantCollection other) {
30
            return Reaction{*this, std::move(other)};
31
33
       std::ostream &operator<<(std::ostream &s, const Reaction &reaction) {</pre>
34
           s << "{ ";
35
           for (const auto& reactant: reaction.from) {
                s << reactant.name << "+";
37
38
           s << "\b" << " >>= ";
           if (reaction.catalysts.has_value()) {
                s << "(";
                for (const auto& catalyst: reaction.catalysts.value()) {
42
                    s << catalyst.name << "+";
43
                }
                s << "\b" << ") ";
45
           }
           for (const auto& reactant: reaction.to) {
                s << reactant.name << "+";
49
           s << "\b";
50
51
            return s << " - " << reaction.rate << " }";</pre>
       }
53
       void Reaction::compute_delay2(SimulationState& state, std::default_random_engine &engine) {
            size_t reactant_amount{1};
56
           size_t catalyst_amount{1};
57
58
            for (const Reactant& reactant: from) {
59
                auto amount = reactant.name == "__env__" ? 1 :
```

```
→state.reactants.get(reactant.name).amount;
                 reactant_amount *= amount;
            }
62
            // New: check if amount is 0 already
63
            if (reactant_amount == 0) {
                 delay = -1;
65
                 return;
66
            }
            if (catalysts.has_value()) {
69
                 for (auto& catalyst: catalysts.value()) {
70
                     catalyst_amount *= state.reactants.get(catalyst.name).amount;
71
                 }
            }
73
            double_t rate_k = rate * reactant_amount * catalyst_amount;
76
            if (rate_k > 0) {
77
                 delay = std::exponential_distribution<double_t>(rate_k)(engine);
78
79
            } else {
                 delay = -1;
            }
81
        }
82
        void Reaction::compute_delay(SimulationState& state, std::default_random_engine &engine) {
            size_t reactant_amount{1};
85
            size_t catalyst_amount{1};
86
            for (const Reactant& reactant: from) {
88
                 auto amount = reactant.name == "__env__" ? 1 :
  →state.reactants.get(reactant.name).amount;
                 reactant_amount *= amount;
91
            if (catalysts.has_value()) {
92
                 for (auto& catalyst: catalysts.value()) {
93
                     catalyst_amount *= state.reactants.get(catalyst.name).amount;
                 }
95
            }
            double_t rate_k = rate * reactant_amount * catalyst_amount;
99
            if (rate_k > 0) {
100
                 delay = std::exponential_distribution<double_t>(rate_k)(engine);
101
            } else {
102
                 delay = -1;
103
            }
104
106
        std::ostream &operator<<(std::ostream &s, const SimulationState& state) {</pre>
107
            s << "{" << std::endl
108
                 << "time: " << state.time << "," << std::endl
                 << "reactants: {" << std::endl;</pre>
110
            for(auto& pair: state.reactants) {
                 s << pair.first << ": " << pair.second.amount << "," << std::endl;
            }
114
            s << "}";
115
            return s;
116
117
        }
    }
118
```

```
11
   // Created by Mathias on 11-05-2021.
   #ifndef SP_EXAM_PROJECT_SIMULATION_MONITOR_H
   #define SP_EXAM_PROJECT_SIMULATION_MONITOR_H
   #include <functional>
   #include "data.h"
   namespace StochasticSimulation {
11
12
       // Requirement 7 classes for generic system state monitors
13
       class simulation_monitor {
14
       public:
15
           virtual void monitor(SimulationState& state) = 0;
       };
18
19
       class empty_simulation_monitor: public simulation_monitor {
20
           void monitor(SimulationState &state) override {
             return;
           };
23
       };
       static auto EMPTY_SIMULATION_MONITOR = empty_simulation_monitor{};
26
27
       class basic_simulation_monitor: public simulation_monitor {
28
       private:
           const std::function<void(SimulationState&)> monitor_function;
30
       public:
31
           basic_simulation_monitor(const std::function<void(SimulationState&)>& monitor_func):
               simulation_monitor{},
               monitor_function{monitor_func}
34
           {}
35
           void monitor(SimulationState& state) override {
37
               monitor_function(state);
38
           }
       };
41
42
   #endif //SP_EXAM_PROJECT_SIMULATION_MONITOR_H
```

Listing 9: SymbolTable.h

```
// //
// Created by Mathias on 10-05-2021.
//

#ifndef SP_EXAM_PROJECT_SYMBOLTABLE_H
#define SP_EXAM_PROJECT_SYMBOLTABLE_H

#include <unordered_map>
#include <string>
#include <stdexcept>
#include <utility>
#include <iterator>

namespace StochasticSimulation {
```

```
15
       // Requirement 3 generic symbol table
16
       struct SymbolTableException : public std::exception
17
18
           std::string message;
19
       public:
20
           explicit SymbolTableException(std::string message): message(std::move(message))
21
           {}
            [[nodiscard]] const char* what() const override
24
25
                return message.c_str();
26
           }
       };
28
29
       template<typename T>
       class SymbolTable {
31
           using map_type = std::unordered_map<std::string, T>;
32
       private:
33
34
           map_type map{};
       public:
           using iterator = typename map_type::iterator;
36
           using const_iterator = typename map_type::const_iterator;
           SymbolTable<T>() = default;
40
           SymbolTable<T>(const SymbolTable<T>& a) {
41
                map = a.map;
           };
43
44
           SymbolTable<T>(SymbolTable<T>&& a) {
                map = std::move(a.map);
           };
48
           ~SymbolTable() = default;
49
           SymbolTable<T>& operator=(const SymbolTable& a) {
51
                map = a.map;
                return *this;
           };
55
           SymbolTable<T>& operator=(SymbolTable&& a) {
56
                map = std::move(a.map);
57
                return *this;
           };
59
           void put(const std::string& key, T value) {
                if (!map.contains(key)) {
                    map.insert_or_assign(key, value);
63
                } else {
64
                    throw SymbolTableException("Key " + key + " already used");
           }
           T& get(const std::string& key) {
                try {
70
                    return map.at(key);
71
                } catch (std::out_of_range& e) {
72
                    throw SymbolTableException("Key " + key + " was not found");
                }
           }
75
```

```
const T& get(const std::string& key) const {
                 try {
78
                     return map.at(key);
79
                 } catch (std::out_of_range& e) {
                     throw SymbolTableException("Key " + key + " was not found");
                 }
            }
            bool contains(const std::string& key) {
85
                 return map.contains(key);
86
            }
87
            std::unordered_map<std::string, T> getMap() {
89
                 return map;
            }
            iterator begin() {
93
                 return map.begin();
94
95
            }
            iterator end() {
97
                 return map.end();
100
            const_iterator begin() const {
101
                 return map.begin();
102
            }
104
            const_iterator end() const {
105
                 return map.end();
106
            }
108
        };
109
110
    #endif //SP_EXAM_PROJECT_SYMBOLTABLE_H
112
```

Listing 10: Results

```
Introduction:\\
{
    \{ A+B >>= (D) C - 0.001 \}
}
Covid:
{
         \{ S >>= (I) E - 7.74194e - 05 \},
         \{E >>= I - 0.196078 \},
           I >>= R - 0.322581 \},
          I >>= H - 0.000290061 \},
         \{ H >>= R - 0.0988142 \}
}
Carcadian Rythm:
         \{ A+DA >>= D A - 1 \},
         \{ D A >>= A + DA - 50 \},
         \{ A+DR >>= D_R - 1 \},
         \{ D_R >>= A + DR - 100 \},
```

Pretty-print Reactions:

```
\{ D A >>= D A + MA - 500 \},
                \{ DA >>= DA + MA - 50 \},
               \{ D R >>= D_R + MR - 50 \},
                \{ DR >>= DR + MR - 0.01 \},
                MA >>= A+MA - 50 \},
                \{ MR >>= MR + R - 5 \},
                \{ A+R >>= C - 2 \},
                \{ C >>= R - 1 \},
                \left\{ \begin{array}{l} \text{A} >>= \_\text{env}\_ - 1 \end{array} \right\}, \\ \left\{ \begin{array}{l} \text{R} >>= \_\text{env}\_ - 0.2 \end{array} \right\}, \\ \left\{ \begin{array}{l} \text{MA} >>= \_\text{env}\_ - 10 \end{array} \right\}, \\ \left\{ \begin{array}{l} \text{MR} >>= \_\text{env}\_ - 0.5 \end{array} \right\} 
}
Carcadian Rythm alternative:
                \{ A+DA >>= D_A - 1 \},
                \{ D_A >>= A + DA - 50 \},
                \{ A+DR >>= D_R - 1 \},
                \{ D_R >>= A + DR - 100 \},
                \{ \__{env} >>= (D_A) MA - 500 \},
                \{ \__{env} >>= (DA) MA - 50 \},
                \{ \__{env}_{-} >>= (D_R) MR - 50 \},
               \{ -\text{env}_{-} >>= (DR) \text{ MR} - 0.01 \},
               \{ -\text{env}_{-}^{-} >>= (MA) A - 50 \},
                     _{\text{env}} >>= (MR) R - 5 },
                  A+R >>= C - 2 \},
                \{ C >>= R - 1 \},
                \left\{ \begin{array}{l} A>>= \_ \operatorname{env}_- -1 \end{array} \right\}, \\ \left\{ \begin{array}{l} R>>= \_ \operatorname{env}_- -0.2 \end{array} \right\}, \\ \left\{ \begin{array}{l} MA>>= \_ \operatorname{env}_- -10 \end{array} \right\}, \\ \left\{ \begin{array}{l} MR>>= \_ \operatorname{env}_- -0.5 \end{array} \right\} 
}
Example output from monitoring hospitalized (not the one on the graph)
Max hospitalized: 3
Mean hospitalized: 0.551814
Benchmarks:
Benchmarking with circadian rhythm example (max_time=100) (30 times each)
Simulation 1 mean time (nanoseconds): 117913398
```

Simulation 2 mean time (nanoseconds): 17683541

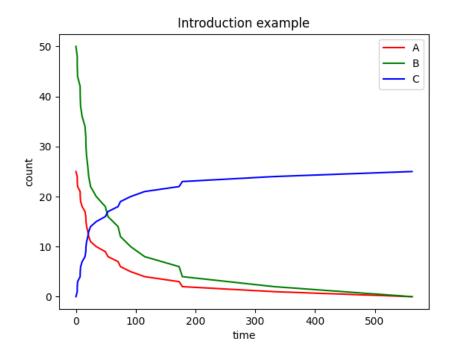


Figure 1: A(0)=25, B(0)=50, D=1

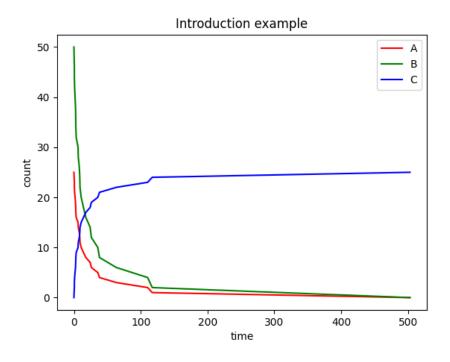


Figure 2: A(0)=25, B(0)=50, D=2

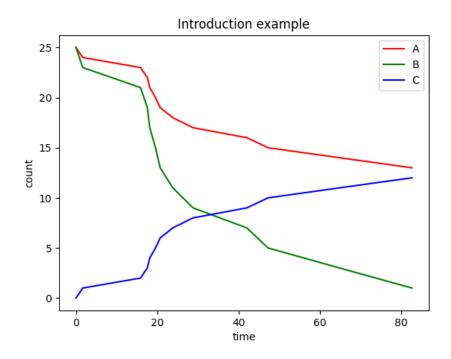


Figure 3: A(0)=25, B(0)=25, D=1

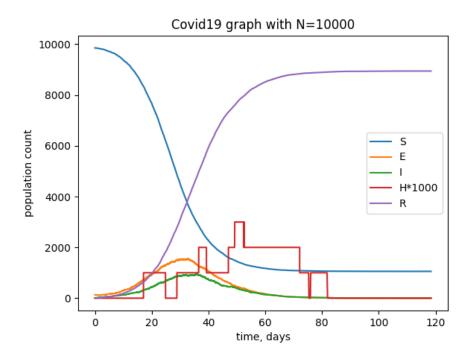


Figure 4: Sample covid trajectory

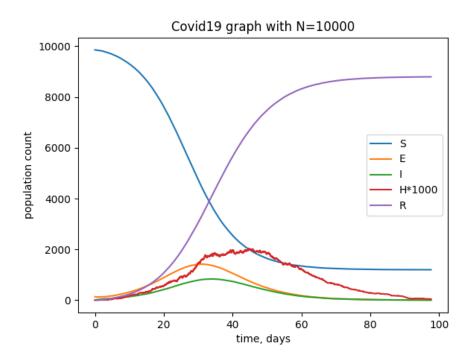


Figure 5: Mean covid trajectory of 30 simulations

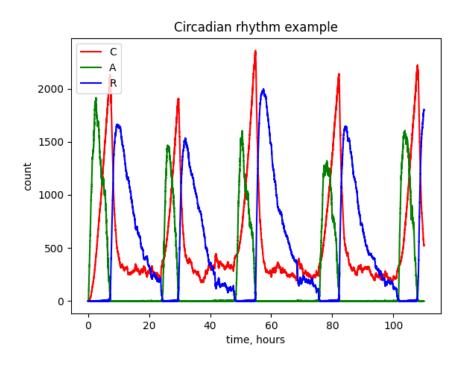


Figure 6: Sample circadian rythm trajectory

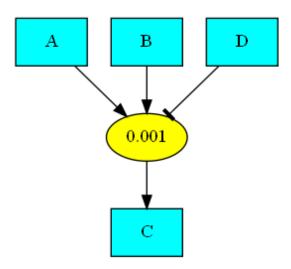


Figure 7: Intro reaction graph

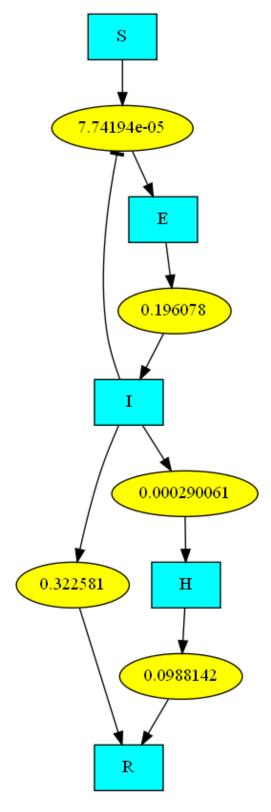


Figure 8: Covid reaction graph

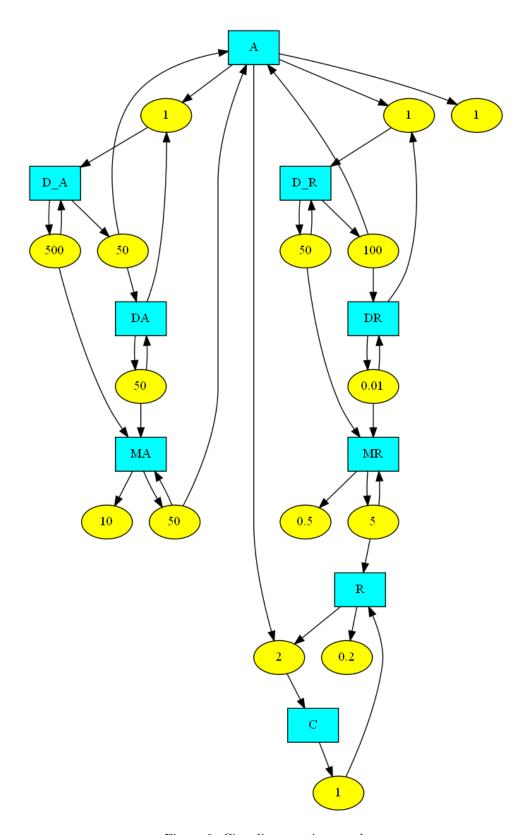


Figure 9: Circadian reaction graph

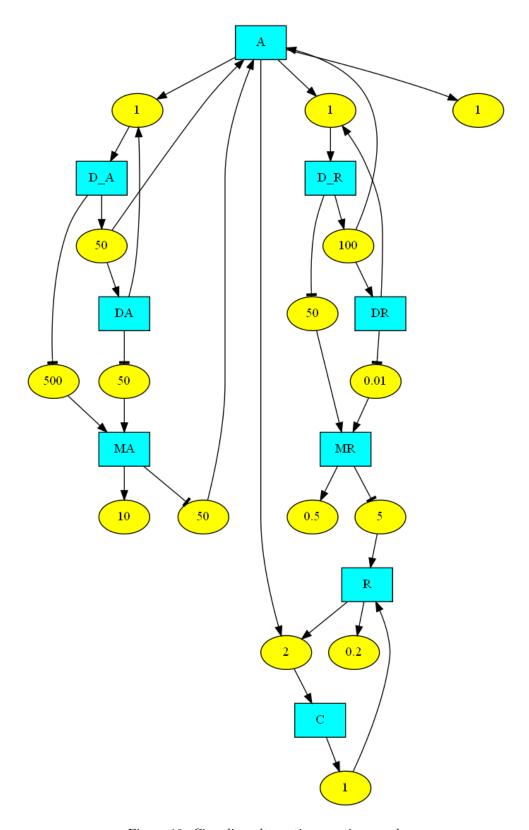


Figure 10: Circadian alternative reaction graph