SP Exam Project 2021

Malte Z. Andreasen

May 15, 2021

The program does not fulfill all requirements, unfortunately. I had trouble getting graphviz to link after compiling, which is why I only managed to produce the digraph text-output.

The calculations were not done correctly, and some of the charts did not fit into Excel whilst trying to produce the needed answers. The graphs have not been included in this pdf as I ran out of time.

Listing 1: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.19)
project(exam)

set(CMAKE_CXX_STANDARD 17)

add_executable(exam main.cpp vessel_t.h reactant_t.h rule_t.h tempRule_t.h graphMaker.h
↪stochasticSimulator.h)

target_link_libraries(exam PUBLIC pthread)
```

Listing 2: rule_t.h

```
#ifndef EXAM_RULE_T_H
#define EXAM_RULE_T_H

#include <ostream>

template<typename T>
class rule_t {
private:
    std::vector<std::shared_ptr<T>> input{};
    std::vector<std::shared_ptr<T>> output{};
    std::vector<std::shared_ptr<T>> catalysts{};
    std::string id;
    double reactionRate{};

    std::string rateToSensibleString() {
        char array[50];
        if (reactionRate > (3 * pow(10, 4)) || reactionRate < 1 * pow(10, -2)){
            sprintf(array, "%e", reactionRate);
        }
        else {
            sprintf(array, "%.3f", reactionRate);
        }
        return std::string(array);
    }

public:
    rule_t() = default;

    rule_t<T>(const rule_t<T>& other) {
        input = other.input;
        output = other.output;
        catalysts = other.catalysts;
        reactionRate = other.reactionRate;
```

```
    }

    rule_t<T>& operator=(const rule_t<T> other) {
        if (this != &other) {
            input = other.input;
            output = other.output;
            catalysts = other.catalysts;
            reactionRate = other.reactionRate;
        }
        return *this;
    }

    rule_t<T>(rule_t<T>&& other) noexcept {
        input = other.input;
        output = other.output;
        catalysts = other.catalysts;
        reactionRate = other.reactionRate;

        other.input = nullptr;
        other.output = nullptr;
        other.catalysts = nullptr;
        other.reactionRate = 0;
    }

    rule_t<T>& operator=(rule_t<T>&& other) noexcept {
        if (this != &other) {
            input = other.input;
            output = other.output;
            catalysts = other.catalysts;
            reactionRate = other.reactionRate;

            other.input = nullptr;
            other.output = nullptr;
            other.catalysts = nullptr;
            other.reactionRate = 0;
        }
        return *this;
    }

    ~rule_t() = default;

    const std::vector<std::shared_ptr<T>>& getInput() const {
        return input;
    }

    void setInput(const std::vector<std::shared_ptr<T>>& newInput) {
        rule_t::input = newInput;
    }

    const std::vector<std::shared_ptr<T>>& getOutput() const {
        return output;
    }

    void setOutput(const std::vector<std::shared_ptr<T>>& newOutput) {
        rule_t::output = newOutput;
    }

    const std::vector<std::shared_ptr<T>>& getCatalysts() const {
        return catalysts;
    }

```

```
    void setCatalysts(const std::vector<std::shared_ptr<T>>& newCatalysts) {
        rule_t::catalysts = newCatalysts;
    }

    [[nodiscard]] double getReactionRate() const {
        return reactionRate;
    }

    void setReactionRate(double newReactionRate) {
        rule_t::reactionRate = newReactionRate;
    }

    [[nodiscard]] const std::string& getId() const {
        return id;
    }

    void setId(const std::string& id) {
        rule_t::id = id;
    }

    friend std::ostream& operator<<(std::ostream& os, const rule_t& rule) {
        os << "reactionRate: " << rule.reactionRate;
        return os;
    }

    std::string toDigraphFormat(const std::string& id) {
        auto a = "\t" + id + "[label=\"" + this->rateToSensibleString() +
                 "\",shape=\"oval\",style=\"filled\",fillcolor=\"yellow\"];\n";
        for (auto reactant : catalysts) {
            auto catalyst = *reactant;
            a += "\t" + catalyst.getName() + " -> " + id + " [arrowhead=\"tee\"];\n";
        }
        for (auto reactant : input) {
            auto i = *reactant;
            if (i.getName() == "env") continue;
            a += "\t" + i.getName() + " -> " + id + ";\n";
        }
        for (auto reactant : output) {
            auto o = *reactant;
            if (o.getName() == "env") continue;
            a += "\t" + id + " -> " + o.getName() + ";\n";
        }
        return a;
    }
};


#endif //EXAM_RULE_T_H
```

Listing 3: tempRule_t.h

```
#ifndef EXAM_TEMPRULE_T_H
#define EXAM_TEMPRULE_T_H

template<typename T>
struct tempRule_t {
    std::vector<T> input{};
    std::vector<T> output{};
};


#endif //EXAM_TEMPRULE_T_H
```

Listing 4: vessel_t.h

```
#ifndef EXAM_VESSEL_T_H
#define EXAM_VESSEL_T_H


#include <utility>
#include <vector>
#include <memory>
#include <ostream>
#include "reactant_t.h"
#include "rule_t.h"

class vessel_t {
private:
    std::vector<std::shared_ptr<reactant_t>> reactants{};
    std::vector<std::pair<std::shared_ptr<rule_t<reactant_t>>, std::string>> rules{};
public:
    reactant_t& environment() {
        auto r = std::make_shared<reactant_t>("env", -1);
        reactants.push_back(r);
        return *r;
    }

    reactant_t& operator()(std::string&& name, double amount) {
        auto r = std::make_shared<reactant_t>(name, amount);
        reactants.push_back(r);
        return *r;
    }

    /// New reaction rule in vessel with no catalysts -RQ1
    std::pair<std::shared_ptr<rule_t<reactant_t>>, std::string>
↪operator()(tempRule_t<reactant_t>& tempRule, const double rate) {
        auto inputVector = std::vector<std::shared_ptr<reactant_t>>{};
        for (const auto& reactant : tempRule.input) {
            inputVector.push_back(getReactantByName(reactant.getName()));
        }

        auto outputVector = std::vector<std::shared_ptr<reactant_t>>{};
        for (const auto& reactant : tempRule.output) {
            outputVector.push_back(getReactantByName(reactant.getName()));
        }

        delete &tempRule;

        auto newRule = std::make_shared<rule_t<reactant_t>>();
        newRule->setInput(inputVector);
        newRule->setOutput(outputVector);
        newRule->setReactionRate(rate);

        auto newPair = std::pair<std::shared_ptr<rule_t<reactant_t>>, std::string>(newRule, "r"
↪+ std::to_string(rules.size()));

        rules.push_back(newPair);
        return newPair;
    }

    /// New reaction rule in vessel with one catalyst passed as reactant. -RQ1
    void operator()(tempRule_t<reactant_t>& tempRule, const reactant_t& catalyst, const double
↪rate) {
```

```
56          auto newPair = (*this)(tempRule, rate);
57          newPair.first->setCatalysts({getReactantByName(catalyst.getName())});
58      }
59
60      /// New reaction rule in vessel with multiple catalysts passed as input in tempRule. -RQ1
61      void operator()(tempRule_t<reactant_t>& tempRule, tempRule_t<reactant_t>& catalysts, const
    ↪double rate) {
62          auto newPair = (*this)(tempRule, rate);
63          auto catalystVector = std::vector<std::shared_ptr<reactant_t>>{};
64          for (const auto& catalyst : catalysts.input) {
65              catalystVector.push_back(getReactantByName(catalyst.getName()));
66          }
67          newPair.first->setCatalysts(catalystVector);
68      }
69
70      [[nodiscard]] const std::vector<std::shared_ptr<reactant_t>>& getReactants() const {
71          return reactants;
72      }
73
74      [[nodiscard]] const std::vector<std::pair<std::shared_ptr<rule_t<reactant_t>>,
    ↪std::string>>& getRules() const {
75          return rules;
76      }
77
78      std::shared_ptr<reactant_t> getReactantByName(const std::string& name) {
79          for (auto reactant : reactants) {
80              if (reactant->getName() == name) {
81                  return reactant;
82              }
83          }
84          std::cout << "ERROR!! reactant '" + name + "' not found in vessel..." << std::endl;
85          exit(1); // fatal failure...
86      }
87
88      std::pair<std::shared_ptr<rule_t<reactant_t>>, std::string> getRuleById(const std::string&
    ↪id) {
89          for (auto rule : rules) {
90              if (rule.first->getId() == id) {
91                  return rule;
92              }
93          }
94          std::cout << "ERROR!! Rule '" + id + "' not found in vessel...";
95          exit(1);
96      }
97
98      friend std::ostream& operator<<(std::ostream& os, const vessel_t& vessel) {
99
100         return os;
101     }
102
103     std::string toDigraph() {
104         std::string s = "digraph{\n";
105         for (const auto& reactant : reactants) {
106             auto r = *reactant;
107             if (r.getName() == "env") continue;
108             s.append(r.toDigraphElement());
109         }
110         for (const auto& rule : rules) {
111             auto r = *(rule.first);
112             s.append(r.toDigraphFormat(rule.second));
113         }
```

```
        s.append("}");
        return s;
    }
};

#endif //EXAM_VESSEL_T_H
```

Listing 5: stochasticSimulator.h

```
#ifndef EXAM_STOCHASTICSIMULATOR_H
#define EXAM_STOCHASTICSIMULATOR_H

#include <list>
#include <random>
#include <algorithm>
#include "vessel_t.h"

class stochasticSimulator {
private:
public:
    static std::string saveState(const std::vector<std::shared_ptr<reactant_t>>& vector, double currentTime) {
        std::string s = std::to_string(currentTime);
        for (const auto& v : vector) {
            if (v->getName() == "env") continue;
            s += "; " + std::to_string(v->getAmount());
        }
        s += "\n";
        return s;
    }

    static std::string writeHeaders(const std::vector<std::shared_ptr<reactant_t>>& vector) {
        std::string s = "time";
        for (const auto& v : vector) {
            if (v->getName() == "env") continue;
            s += "; " + v->getName();
        }
        s += "\n";
        return s;
    }

    static double randomExp(double k) {
        std::random_device d;
        std::exponential_distribution<> distribution(k);
        return distribution(d);
    }

    static std::pair<double, std::shared_ptr<rule_t<reactant_t>>>
    calculateDelayK(std::shared_ptr<rule_t<reactant_t>> rule) {
        auto lambda_k = rule->getReactionRate();
        for (const auto& r : rule->getInput()) {
            if (r->getName() == "env") continue;
            lambda_k *= r->getAmount();
        }
        for (const auto& c : rule->getCatalysts()) {
            if (c->getName() == "env") continue;
            lambda_k *= c->getAmount();
        }
        lambda_k = randomExp(lambda_k);
        return std::pair<double, std::shared_ptr<rule_t<reactant_t>>>(lambda_k, rule);
    }

```

```
53      static bool compareRulesWithDelay(std::pair<double, std::shared_ptr<rule_t<reactant_t>>> a,
54                                        std::pair<double, std::shared_ptr<rule_t<reactant_t>>> b) {
55          return a.first < b.first;
56      }
57
58      static std::pair<double, std::shared_ptr<rule_t<reactant_t>>>
59      calculateDelays(const std::vector<std::pair<std::shared_ptr<rule_t<reactant_t>>,
    ↪std::string>>& vector) {
60          auto rulesWithDelay = std::vector<std::pair<double, std::shared_ptr<rule_t<reactant_t>>>>{};
61          for (const auto& v : vector) {
62              rulesWithDelay.push_back(calculateDelayK(v.first));
63          }
64          std::sort(rulesWithDelay.begin(), rulesWithDelay.end(),
    ↪stochasticSimulator::compareRulesWithDelay);
65          return rulesWithDelay.front();
66      }
67
68      static double applyRule(const std::pair<double, std::shared_ptr<rule_t<reactant_t>>>& pair) {
69          // Rule Applicable wrt. inputs?
70          for (const auto& r : pair.second->getInput()) {
71              if (r->getAmount() <= 0) return pair.first;
72          }
73
74          // Rule Applicable wrt. catalysts?
75          for (const auto& c : pair.second->getCatalysts()) {
76              if (c->getAmount() <= 0) return pair.first;
77          }
78
79          // Rule is applicable, if we reached here. Apply.
80          for (const auto& r : pair.second->getInput()) {
81              auto current = r->getAmount();
82              r->setAmount(current - 1);
83          }
84          for (const auto& r : pair.second->getOutput()) {
85              auto current = r->getAmount();
86              r->setAmount(current + 1);
87          }
88          return pair.first;
89      }
90
91      /// Simulate according to algorithm. Doesn't work correctly, as some calculations for
    ↪reactions with
92      /// multiple of the same reactant aren't calculated correctly.
93      static void simulate(const vessel_t& vessel, const double MaxTime, const std::string&
    ↪outputFileName) {
94          std::cout << "started calculating " + outputFileName << std::endl;
95          auto relativeFileName = "../../" + outputFileName + ".csv";
96          std::ofstream file;
97          file.open(relativeFileName);
98          file << writeHeaders(vessel.getReactants());
99
100         // write initial state at t = 0
101         double currentTime{0.0};
102         file << saveState(vessel.getReactants(), currentTime);
103         while (currentTime < MaxTime) {
104             // Calculate delay for every rule and choose lowest
105             auto nextReaction = calculateDelays(vessel.getRules());
106
107             // Apply rule (if applicable)
108             currentTime += applyRule(nextReaction);
109
```

```
            // Save state
            file << saveState(vessel.getReactants(), currentTime);
        }

        file.close();
        std::cout << "Finished " + outputFileName + "!" << std::endl;
    }
};

#endif //EXAM_STOCHASTICSIMULATOR_H
```

Listing 6: reactant_t.h

```
#include <utility>
#include "rule_t.h"
#include "tempRule_t.h"

#ifndef EXAM_REACTANT_T_H
#define EXAM_REACTANT_T_H


class reactant_t {
private:
    std::string name{};
    int amount{};
public:
    [[nodiscard]] const std::string& getName() const {
        return name;
    }

    [[nodiscard]] int getAmount() const {
        return amount;
    }

    void setAmount(int newAmount) {
        reactant_t::amount = newAmount;
    }

public:
    reactant_t(std::string  name, int amount) : name(std::move(name)), amount(amount) {}

    reactant_t(const reactant_t& other) {
        name = other.name;
        amount = other.amount;
    }

    reactant_t& operator=(const reactant_t& other) {
        if (this != &other) {
            name = other.name;
            amount = other.amount;
        }
        return *this;
    }

    ~reactant_t() = default;

    friend tempRule_t<reactant_t>& operator+(const reactant_t& lhs, const reactant_t& rhs) {
        auto rule = new tempRule_t<reactant_t>;
        rule->input = {lhs, rhs};
        return *rule;
    }

```

```
50      friend tempRule_t<reactant_t>& operator+(const reactant_t& lhs, tempRule_t<reactant_t>& rhs) {
51          rhs.input.emplace_back(lhs);
52          return rhs;
53      }
54
55      friend tempRule_t<reactant_t>& operator+(tempRule_t<reactant_t>& lhs, const reactant_t& rhs) {
56          lhs.input.emplace_back(rhs);
57          return lhs;
58      }
59
60      friend tempRule_t<reactant_t>& operator>>=(const reactant_t& lhs, const reactant_t& rhs) {
61          auto rule = new tempRule_t<reactant_t>;
62          rule->input = {lhs};
63          rule->output = {rhs};
64          return *rule;
65      }
66
67      friend tempRule_t<reactant_t>& operator>>=(tempRule_t<reactant_t>& lhs, const reactant_t&
    ↪rhs) {
68          lhs.output = {rhs};
69          return lhs;
70      }
71
72      friend tempRule_t<reactant_t>& operator>>=(const reactant_t& lhs, tempRule_t<reactant_t>&
    ↪rhs) {
73          auto rule = new tempRule_t<reactant_t>;
74          rule->input = {lhs};
75          rule->output = rhs.input;
76
77          delete &rhs;
78          return *rule;
79      }
80
81      friend tempRule_t<reactant_t>& operator>>=(tempRule_t<reactant_t>& lhs,
    ↪tempRule_t<reactant_t>& rhs) {
82          auto rule = new tempRule_t<reactant_t>;
83          rule->input = lhs.input;
84          rule->output = rhs.input;
85
86          delete &lhs;
87          delete &rhs;
88
89          return *rule;
90      }
91
92      friend tempRule_t<reactant_t>& operator*(int scalar, const reactant_t& rhs) {
93          auto rule = new tempRule_t<reactant_t>;
94          for (int i = 0; i < scalar; ++i) {
95              rule->input.push_back(rhs);
96          }
97          return *rule;
98      }
99
100     std::string toDigraphElement() {
101         auto s = "\t" + name + "[label=\"" + name + "\", shape=\"box\", style=\"filled\",
    ↪fillcolor=\"cyan\"];\n";
102         return s;
103     }
104 };
105
106
```

```
#endif //EXAM_REACTANT_T_H
```

Listing 7: graphMaker.h

```
#ifndef EXAM_GRAPHMAKER_H
#define EXAM_GRAPHMAKER_H

#include <iostream>
#include <fstream>
#include "vessel_t.h"

class graphMaker {
private:
public:
    /// Converts a vessel to a digraph, ready to be pasted into http://www.webgraphviz.com/ -RQ2
    static void vesselToDigraph(vessel_t vessel, const std::string& fileName) {
        auto relativeFileName = "../../" + fileName + ".txt";
        std::ofstream file;
        file.open(relativeFileName);
        file << std::scientific;
        file << vessel.toDigraph();
        file.flush();
        file.close();
    }
};

#endif //EXAM_GRAPHMAKER_H
```

Listing 8: main.cpp

```
#include <iostream>
#include <cmath>
#include <thread>
#include "graphviz/gvc.h"
#include "vessel_t.h"
#include "graphMaker.h"
#include "stochasticSimulator.h"

/** small first example */
vessel_t tiny_example()
{
    auto lambda = 0.001;
    auto v = vessel_t{};
    auto A = v("A", 25);
    auto B = v("B", 50);
    auto C = v("C", 0);
    auto D = v("D", 2);
    v(A + 2*B >>= C, D, lambda);
    return v;
}

/** direct encoding */
vessel_t circadian_oscillator()
{
    auto alphaA = 50.0;
    auto alpha_A = 500.0;
    auto alphaR = 0.01;
    auto alpha_R = 50.0;
    auto betaA = 50.0;
    auto betaR = 5.0;
    auto gammaA = 1.0;
    auto gammaR = 1.0;
```

```
    auto gammaC = 2.0;
    auto deltaA = 1.0;
    auto deltaR = 0.2;
    auto deltaMA = 10.0;
    auto deltaMR = 0.5;
    auto thetaA = 50.0;
    auto thetaR = 100.0;
    auto v = vessel_t{};
    auto env = v.environment();
    auto DA = v("DA", 1);
    auto D_A = v("D_A", 0);
    auto DR = v("DR", 1);
    auto D_R = v("D_R", 0);
    auto MA = v("MA", 0);
    auto MR = v("MR", 0);
    auto A = v("A", 0);
    auto R = v("R", 0);
    auto C = v("C", 0);
    v(A + DA >>= D_A, gammaA);
    v(D_A >>= DA + A, thetaA);
    v(A + DR >>= D_R, gammaR);
    v(D_R >>= DR + A, thetaR);
    v(D_A >>= MA + D_A, alpha_A);
    v(DA >>= MA + DA, alphaA);
    v(D_R >>= MR + D_R, alpha_R);
    v(DR >>= MR + DR, alphaR);
    v(MA >>= MA + A, betaA);
    v(MR >>= MR + R, betaR);
    v(A + R >>= C, gammaC);
    v(C >>= R, deltaA);
    v(A >>= env, deltaA);
    v(R >>= env, deltaR);
    v(MA >>= env, deltaMA);
    v(MR >>= env, deltaMR);
    return v;
}

/** alternative encoding using catalysts */
vessel_t circadian_oscillator2()
{
    auto alphaA = 50.0;
    auto alpha_A = 500.0;
    auto alphaR = 0.01;
    auto alpha_R = 50.0;
    auto betaA = 50.0;
    auto betaR = 5.0;
    auto gammaA = 1.0;
    auto gammaR = 1.0;
    auto gammaC = 2.0;
    auto deltaA = 1.0;
    auto deltaR = 0.2;
    auto deltaMA = 10.0;
    auto deltaMR = 0.5;
    auto thetaA = 50.0;
    auto thetaR = 100.0;
    auto v = vessel_t{};
    auto env = v.environment();
    auto DA = v("DA", 1);
    auto D_A = v("D_A", 0);
    auto DR = v("DR", 1);
    auto D_R = v("D_R", 0);
```

```
    auto MA = v("MA", 0);
    auto MR = v("MR", 0);
    auto A = v("A", 0);
    auto R = v("R", 0);
    auto C = v("C", 0);
    v(A + DA >>= D_A, gammaA);
    v(D_A >>= DA + A, thetaA);
    v(DR + A >>= D_R, gammaR);
    v(D_R >>= DR + A, thetaR);
    v(env >>= MA, D_A, alpha_A);
    v(env >>= MA, DA, alphaA);
    v(env >>= MR, D_R, alpha_R);
    v(env >>= MR, DR, alphaR);
    v(env >>= A, MA, betaA);
    v(env >>= R, MR, betaR);
    v(A + R >>= C, gammaC);
    v(C >>= R, deltaA);
    v(A >>= env, deltaA);
    v(R >>= env, deltaR);
    v(MA >>= env, deltaMA);
    v(MR >>= env, deltaMR);
    return v;
}

/** covid-19 example */
vessel_t seihr(uint32_t N)
{
    auto v = vessel_t{};
    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; // basic reproductive number (initial, without lockdown etc)
    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; // recovery/death rate in hospital (H -> R) ~10.12 days
    auto S = v("S", S0); // susceptible
    auto E = v("E", E0); // exposed
    auto I = v("I", I0); // infectious
    auto H = v("H", 0); // hospitalized
    auto R = v("R", 0); // removed/immune (recovered + dead)
    v(S >>= E, I, beta/N);
    v(E >>= I, alpha);
    v(I >>= R, gamma);
    v(I >>= H, kappa);
    v(H >>= R, tau);
    return v;
}

int main() {
    // Create vessels via DSEL -RQ1
    auto tinyExample_v = tiny_example();
    auto circadianOscillator_v = circadian_oscillator();
    auto circadianOscillator2_v = circadian_oscillator2();
    auto seihr_NJ_v = seihr(589755);  // Magic number - sorry. Found in assignment.
    auto seihr_DK_v = seihr(10000); // Reduced to produce lov enough amount of output.

    // Output vessels to Graphviz digraph entities -RQ2
```

```
155     graphMaker::vesselToDigraph(tinyExample_v, "tiny-example-digraph");
156     graphMaker::vesselToDigraph(circadianOscillator_v, "CO-digraph");
157     graphMaker::vesselToDigraph(circadianOscillator2_v, "CO-2-digraph");
158     graphMaker::vesselToDigraph(seihr_NJ_v, "seihr-nj-digraph");
159     graphMaker::vesselToDigraph(seihr_DK_v, "seihr-dk-digraph");
160 
161     // Simulate the reaction rules and output to .csv -RQ4,5
162     std::thread t1(stochasticSimulator::simulate, tinyExample_v, 200, "tiny-example-data");
163     std::thread t2(stochasticSimulator::simulate, circadianOscillator2_v, 100,   ↙
↪"circadian-oscillator-data");
164     std::thread t3(stochasticSimulator::simulate, seihr_NJ_v, 200, "seihr-nj-data");
165     std::thread t4(stochasticSimulator::simulate, seihr_DK_v, 200, "seihr-dk-data");
166 
167     t1.join();
168     t2.join();
169     t3.join();
170     t4.join();
171     return 0;
172 }
```