Software Engineering

Cicio Ionuț

08/01/2025

Contents

1 Software models

Software projects require **design choices** that often can't be driven by experience or reasoning alone. That's why a **model** of the project is needed to compare different solutions. In this course, to describe software systems, we use **discrete time Markov chains**.

1.1 The "Amazon Prime Video" article

If you were tasked with designing the software architecture for **Amazon Prime Video** *(a live streaming service for Amazon)*, how would you go about it? What if you had the to keep the costs minimal? Would you use a distributed architecture or a monolith application?

More often than not, monolith applications are considered **more costly** and **less scalable** than the counterpart due to an inefficient usage of resources. But, in a recent article, a Senior SDE at Prime Video describes how they *"**reduced the cost** of the audio/video monitoring infrastructure by **90%**"* [3] by using a monolith architecture.

While there isn't always definitive answer, one way to go about this kind of choice is building a model of the system to compare the solutions. In the case of Prime Video, *"the audio/video monitoring service consists of three major components:"* [3]

- the **media converter** converts input audio/video streams
- the **defect detectors** analyze frames and audio buffers in real-time
- the **orchestrator** controls the flow in the service

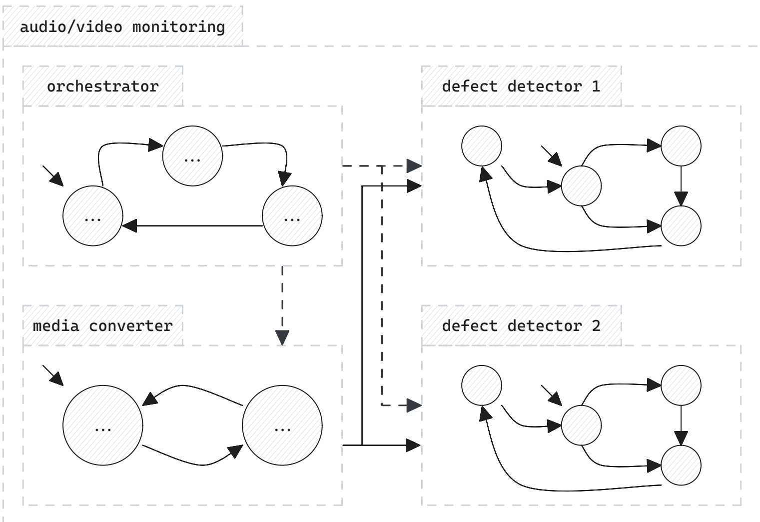


Figure 1: audio/video monitoring system

The system can be **simulated** by modeling its components as **connected probabilistic stateful automatons**.

1.2 Formal theory

1.2.1 Markov chains

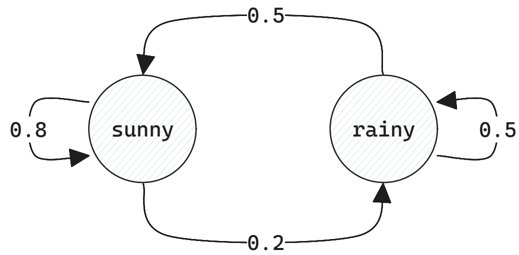
A Markov Chain is defined by a set of **states** S and the **transition probability** $p : S \times S \to [0, 1]$ such that $p(s'|s)$ is the probability to transition to state s' if the current state is s and

$$\forall s \in S \ \sum_{s' \in S} p(s'|s) = 1 \tag{1}$$

A simple model of the weather could be $S = \{\text{sunny}, \text{rainy}\}$

$p =$

	sunny	rainy
sunny	0.8	0.2
rainy	0.5	0.5



Unfortunately for us, Markov chains aren't enough: to describe complex systems (e.g. a server) we need the concepts of **input**, **output** and **time**.

1.2.2 DTMCs (Discrete Time Markov Chains)

A DTMC M is a tuple (U, X, Y, p, g) s.t.
- U, X and Y aren't empty *(otherwise stuff doesn't work)*
- U is the set of **input values**
- X is the set of **states**
- Y is the set of **output values**
- $p : X \times X \times U \to [0, 1]$ is the **transition probability**
- $g : X \to Y$ is the **output function**

The same constrain in Equation 1 holds for the DTMC, with an important difference: the **probability depends on the input value**. This means that **for each input value**, the sum of the probabilities to transition for **that input value** must be 1.

$$\forall x \in X \ \ \forall u \in U \ \sum_{x' \in X} p(x'|x, u) = 1 \tag{2}$$

Let M be a DTMC, t be a time **instant** and d a time **interval**, we have

$$\begin{aligned} X(0) &= x_0 \\ X(t+d) &= \begin{cases} x_0 & \text{with prob. } \ p(x_0 \mid X(t), U(t)) \\ x_1 & \text{with prob. } \ p(x_1 \mid X(t), U(t)) \\ \dots \end{cases} \\ Y(t) &= g(X(t)) \end{aligned} \tag{3}$$

1.2.2.1 An example of DTMC

Let's consider the development process of a team. We can define a DTMC $M = (U, X, Y, p, g)$ s.t.

- $U = \{\varnothing\}$ because M doesn't have any input and $U \neq \varnothing$
- $X = \{0, 1, 2, 3, 4\}$
- $Y = \text{Cost} \times \text{Duration}$

$$g(x) = \begin{cases} (0,0) & x = 0 \lor x = 4 \\ (20000, 2) & x = 1 \lor x = 3 \\ (40000, 4) & x = 2 \end{cases} \tag{4}$$

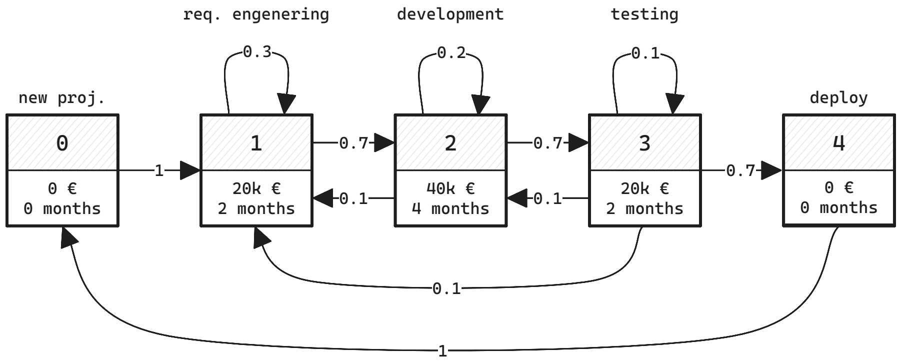


Figure 2: the model of a team's development process

$p =$

$\varnothing$	**0**	**1**	**2**	**3**	**4**
0	0	1	0	0	0
1	0	.3	.7	0	0
2	0	.1	.2	.7	0
3	0	.1	.1	.1	.7
4	1	0	0	0	0

Notice that we have only 1 table because $|U| = 1$ (we have exactly 1 input value). If we had $U = \{\text{apple}, \text{banana}, \text{orange}\}$ we would have had to describe 3 tables, one **for each input value**.

1.2.3 Network of Markov chains

To describe complex systems we don't want to model a single big DTMC (the task would be hard and error prone). What we want to do instead is model many simple DTMCs and connect them.

Let M_1, M_2 be two DTMCs, the input of M_2 depends on M_1's output

$$U_2(t) = f(Y_1(t)) \tag{5}$$

1.3 Tips and tricks

1.3.1 Calculate average incrementally

Given a set of values $V = \{v_1, ..., v_n\}$ s.t. $|V| = n$ the average $\overline{x}_n$ is s.t.

$$\overline{x}_n = \frac{\sum_{i=0}^{n} v_i}{n} \tag{6}$$

The problem with this method is that, by adding up all the values before the division, the **numerator** could easily **overflow**, as the biggest integer we can represent precisely with singe-precision floating-point number is 16777216 [4]. There is a way to calculate $\overline{x}_{n+1}$ given $\overline{x}_n$

$$\begin{gathered}
\overline{x}_{n+1} = \frac{\sum_{i=0}^{n+1} v_i}{n+1} = \\
\frac{\left(\sum_{i=0}^{n} v_i\right) + v_{n+1}}{n+1} = \\
\frac{\sum_{i=0}^{n} v_i}{n+1} + \frac{v_{n+1}}{n+1} = \\
\frac{\left(\sum_{i=0}^{n} v_i\right) n}{(n+1)n} + \frac{v_{n+1}}{n+1} = \\
\frac{\sum_{i=0}^{n} v_i}{n} \cdot \frac{n}{n+1} + \frac{v_{n+1}}{n+1} = \\
\frac{\overline{x}_n \cdot n + v_{n+1}}{n+1}
\end{gathered} \tag{7}$$

1.3.2 Welford's online algorithm

"It is often useful to be able to compute the variance in a single pass, inspecting each value x_i only once; for example, when the data is being collected without enough storage to keep all the values, or when costs of memory access dominate those of computation." (Wikpedia)

$$\begin{gathered}
M_{2,n} = \sum_{i=1}^{n} (x_i - \overline{x}_n)^2 \\
M_{2,n} = M_{2,n-1} + (x_n - \overline{x}_{n-1})(x_N - \overline{x}_n) \\
\sigma_n^2 = \frac{M_{2,n}}{n} \\
s_n^2 = \frac{M_{2,n}}{n-1}
\end{gathered} \tag{8}$$

1.3.3 Euler's method for differential equations

Got it from here baby [5]

2 C++

This section will cover the basics for the exam.

2.1 How to use the random library

The `random` standard library offers useful tools to build our models. It makes Markov chains and probabilistic models easy to implement.

2.1.1 random(), random_device() and default_random_engine()

In C++ there are many ways to **generate random numbers**, I'm gonna keep it short and sweet: don't use `random()`, use `random_device()` to generate the **seed** to instantiate a `default_random_engine()`.

```
#include <iostream>
#include <random>

int main() {
    std::cout << random() ① << std::endl;

    std::random_device random_device; ②
    std::cout << random_device() ③ << std::endl;

    std::default_random_engine r_engine(random_device() ④ );
    std::cout << r_engine() ⑤ << std::endl;
}
```

Listing 1: `examples/random.cpp`

I'll explain: `random()` ① doesn't work very well (TODO: link to the article*), you're encouraged to use the generators C++ offers... but each works in a different way, and you have to choose the best one depending on your needs. `std::random_device` ② is used to generate the **seed** ④ because it uses device randomness (if available, TODO: link to docs / article *), but it's really slow. That **seed** is used to to instantiate one of the other engines ⑤, like `::default_random_engine` (TODO: link with different engines and types). TODO BONUS: only r_eng used with distr

> If you see ③ and ⑤, random_device and r_engine aren't functions, they are instances, but you can call the `()` operator on them because C++ has operator overloading, and it allows you to call custom operators on instances... the same goes for `std::cout` and `<<`

2.1.2 Distributions

Just the capability to generate random numbers isn't enough, we often need to manipulate those numbers to fit our needs. Luckly, `C++` covers **basically all of them**. For example, we can easily simulate the DTMC in Figure 2 like this:

```cpp
#include <iostream>
#include <random>

int main() {
    std::random_device random_device;
    std::default_random_engine random_engine(random_device());

    std::discrete_distribution<> transition_matrix[] = {
        {0, 1},
        {0, .3, .7},
        {0, .2, .2, .6},
        {0, .1, .2, .1, .6},
        {1},
    };

    size_t state = 0;
    for (size_t step = 0; step < 15; step++) {
        state = transition_matrix[state](random_engine);
        std::cout << state << std::endl;
    }

    return 0;
}
```

Listing 2: `examples/transition_matrix.cpp`

2.1.2.1 `uniform_int_distribution` [1]

Let's consider a simple exercise

> To test a system S we have to build a generator that every T seconds sends a value v at S. For each send, the value of T is an **integer** chosen uniformly in the range $[20, 30]$

The code to calculate T would be `T = 20 + rand() % 11;`, which is very **error prone**, hard to remember and has no semantic value. In `C++` we can do the same thing in a **simpler** and **cleaner** way:

```cpp
std::uniform_int_distribution<> random_T(20, 30); ①
size_t T = ② random_T(random_engine);
```

Now we can easily generate the numbers we want without doing any calculations ①, and we don't have to remember how to generate T ②.

We define the behaviour of T only once ①, so we can easily change it without introducing bugs or inconsistencies. It's also worth to take a look at the implementation of the exercise above (with the addition that $v = T$), as it comes up very often in our models.

```
#include <iostream>
#include <random>

int main() {
    std::random_device random_device;
    std::default_random_engine random_engine(random_device());
    std::uniform_int_distribution<> random_T(20, 30);

    size_t T = random_T(random_engine), next_request_time = T;
    for (size_t time = 0; time < 1000; time++) {
        if (time < next_request_time)
            continue;

        std::cout << T << std::endl;
        T = random_T(random_engine);
        next_request_time = time + T;
    }

    return 0;
}
```

Listing 3: `examples/interval_generator.cpp`

The `uniform_int_distribution` has many other uses, for example, we could want to uniformly generate a random state. If `STATES_SIZE` is the number of states, then we instantiate `std::uniform_int_distribution<> random_state(0, STATES_SIZE - 1 ①);` **BE CAREFUL!** Remember to use `STATES_SIZE - 1` ①, because `uniform_int_distribution` is **inclusive**... forgettig it can lead to very sneaky bugs: it randomly segfaults at different points of the code. It's very hard to debug unless you use `gdb`.

TODO: You can also generate negative numbers TODO: the behaviour is undefined if a > b

2.1.2.2 `uniform_real_distribution` [2]

It is the same as above, with the difference that it generates **real** numbers ($\mathbb{R}$), and b is excluded

2.1.2.3 Bernoulli

TODO: ... you just have to specify the expected value, I haven't used it much up until now

2.1.3 Exponential

2.1.3.1 Poisson

The Poisson distribution is very useful when simulating user requests (generally, the number requests to a servers in a certain instant is described by a Poisson distribution, you just have to specify the expected value)

2.1.3.2 Geometric

Does the job

2.1.3.3 `discrete_distribution`

This one is **SUPER USEFUL!**, generates random integers in the range 0, number of items - 1, but it assigns a weight to each item, so each item as a certain weighted probability to be choose. It can be used in transition matrices, and for a bit more complex systems like the status of the project in Listing 9.

2.2 Dynamic structures

2.2.1 `new` and `delete` vs `malloc()` and `free()`

If you allocate with `new`, you must deallocate with `delete`, you can't mixup them with malloc and free

2.2.2 `std::vector<T>()` instead of `malloc()`

You don't have to allocate memory, basically never! You just use the structures that are implemented in the standard library, and most of the time they are enough for our use cases. They are really easy to use.

2.2.3 `std::deque<T>()`

2.2.4 Sets

Not needed as much

2.2.5 Maps

Could be useful

2.3 I/O

2.3.1 `#include <iostream>`

2.3.2 Files

3 Exercises

Each exercise has 4 digits `xxxx` that are the same as the ones in the `software` folder in the course material.

3.1 First examples

Now we have to put together our **formal definitions** and our `C++` knowledge to build some simple DTMCs and networks.

3.1.1 A simple Markov chain [1100]

Let's begin our modeling journey by implementing a DTMC M s.t.

- $U = \{\varnothing\}$ (see Section 1.2.2.1)
- $X = [0, 1] \times [0, 1]$, each state is a pair ③ of **real** ① numbers in $[0, 1]$
- $Y = [0, 1] \times [0, 1]$
- $p : X \times X \times U \to X = \mathcal{U}(0, 1) \times \mathcal{U}(0, 1)$, the transition probability is a **uniform** distribution ②
- $g : X \to Y : (r_0, r_1) \mapsto (r_0, r_1)$ outputs the current state ④
- $X(0) = (0, 0)$ ③

```
using real_t ① = double;
const size_t HORIZON = 10;

int main() {
    std::random_device random_device;
    std::default_random_engine random_engine(random_device());
    std::uniform_real_distribution<real_t> uniform(0, 1); ②

    std::vector<real_t> state(2, 0); ③
    std::ofstream log("log");

    for (size_t time = 0; time <= HORIZON; time++) {
        for (auto &r : state) r = uniform(random_engine); ②
        log << time << ' ';
        for (auto r : state) log << r << ' '; t ④
        log << std::endl;
    }

    log.close();
    return 0;
}
```

Listing 4: `software/1100/main.cpp`

3.1.2 Markov chains network pt.1 `[1200]`

In this exercise we build 2 DTMCs M_0, M_1 like the one in the first example Section 3.1.1, with the difference that, and $U_i = [0,1] \times [0,1]$:

- $U_0(t+d) = Y_1(t)$
- $U_1(t+d) = Y_0(t)$

TODO: formula to get input from other stuff and calculate the state..., maybe define

$$\begin{aligned} p : X \times X \times U \to [0,1] \\ (x_0, x_1), (x_0', x_1'), (u_0, u_1) \mapsto \ldots \end{aligned} \tag{9}$$

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

```cpp
const size_t HORIZON = 100;
struct DTMC { real_t state[2]; };

int main() {
    std::vector<DTMC> dtmcs(2, {0, 0});

    for (size_t time = 0; time <= HORIZON; time++) {
        for (size_t r = 0; r < 2; r++) {
            dtmcs[0].state[r] =
                dtmcs[1].state[r] * uniform(random_engine);
            dtmcs[1].state[r] =
                dtmcs[0].state[r] + uniform(random_engine);
        }
    }
}
```

Listing 5: `software/1200/main.cpp`

3.1.3 Markov chains network pt.2 `[1300]`

The same as above, but with a different connection

```cpp
int main() {
    std::vector<DTMC> dtmcs({{1, 1}, {2, 2}});

    for (size_t time = 0; time <= HORIZON; time++) {
        dtmcs[0].state[0] =
            .7 * dtmcs[0].state[0] + .7 * dtmcs[0].state[1];
        dtmcs[0].state[1] =
            -.7 * dtmcs[0].state[0] + .7 * dtmcs[0].state[1];

        dtmcs[1].state[0] =
            dtmcs[1].state[0] + dtmcs[1].state[1];
        dtmcs[1].state[1] =
            -dtmcs[1].state[0] + dtmcs[1].state[1];
```

```
    }
}
```

Listing 6: `software/1300/main.cpp`

3.1.4 Markov chains network pt.3 [1400]

The same as above, but with a twist (in the original uses variables to indicate each input… which is sketcy… I can do it with MOCC)

3.2 Traffic light [2000]

In this example we want to model a **traffic light**. The three versions of the system on the drive (`2100`, `2200` and `2300`) do the same thing with a different code structure.

```
const size_t HORIZON = 1000;
enum Light { GREEN = 0, YELLOW = 1, RED = 2 };

int main() {
    auto random_timer_duration =
        std::uniform_int_distribution<>(60, 120);

    Light traffic_light = Light::RED;
    size_t timer = random_timer_duration(random_engine);

    for (size_t time = 0; time <= HORIZON; time++) {
        if (timer > 0) {
            timer--;
            continue;
        }

        traffic_light =
            (traffic_light == RED
                ? GREEN
                : (traffic_light == GREEN ? YELLOW : RED));
        timer = random_timer_duration(random_engine);
    }
}
```

Listing 7: `software/2000/main.cpp`

3.3 Control center

3.3.1 No network [3100]

3.3.2 Network monitor

3.3.2.1 No faults [3200]

3.3.2.2 Faults & no repair [3300]

3.3.3 Faults & repair `[3400]`

3.3.4 Faults & repair + correct protocol `[3500]`

3.4 Statistics

3.4.1 Expected value `[4100]`

In this one we just simulate a development process (phase 0, phase 1, and phase 2), and we calculate the average …

3.4.2 Probability `[4200]`

In this one we simulate a more complex software developmen process, and we calculate the average cost (Wait, what? Do we simulate it multiple times?)

3.5 Development process simulation

One of the ways to implement a Markov Chain (like in Section 1.2.1) is by using a **transition matrix**. The simplest implemenation can be done by using a `std::discrete_distribution` by using the trick in Listing 2.

3.5.1 Random transition matrix [5100]

In this example we build a **random transition matrix**.

```
const size_t HORIZON = 20, STATES_SIZE = 10;

int main() {
    std::random_device random_device;
    std::default_random_engine random_engine(random_device());
    auto random_state = ①
        std::uniform_int_distribution<>(0, STATES_SIZE - 1);
    std::uniform_real_distribution<> random_real_0_1(0, 1);

    std::vector<std::discrete_distribution<>>
        transition_matrix(STATES_SIZE); ②
    std::ofstream log("log.csv");

    for (size_t state = 0; state < STATES_SIZE; state++) {
        std::vector<real_t> weights(STATES_SIZE); ③
        for (auto &weight : weights)
            weight = random_real_0_1(random_engine);

        transition_matrix[state] = ④
            std::discrete_distribution<>(weights.begin(),
                                         weights.end());
    }

    size_t state = random_state(random_engine);
    for (size_t time = 0; time <= HORIZON; time++) {
        log << time << " " << state << std::endl;
        state = transition_matrix[state ⑤ ](random_engine); ⑥
    }

    log.close();
    return 0;
}
```

Listing 8: `software/5100/main.cpp`

A **transition matrix** is a `vector<discrete_distribution<>>` ② just like in Listing 2. Why can we do this? First of all, the states are numbered from `0` to `STATES_SIZE - 1`, that's why we can generate a random state ① just by generating a number from `0` to `STATES_SIZE - 1`.

The problem with using a simple `uniform_int_distribution` is that we don't want to choose the next state uniformly, we want to do something like in Figure 3.

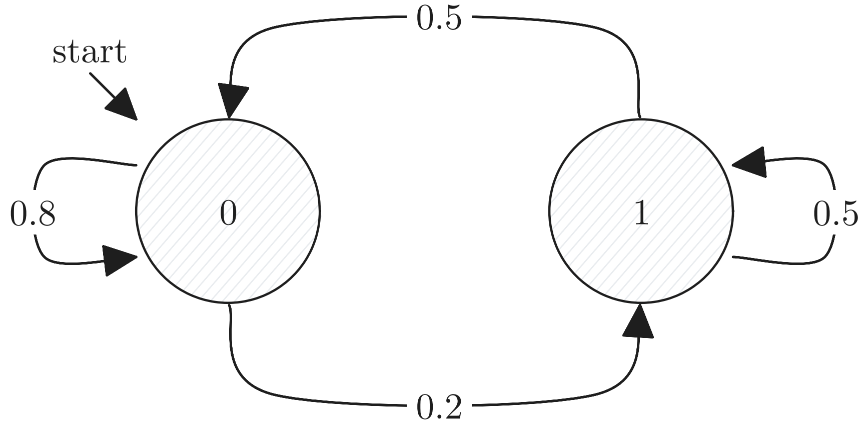


Figure 3: A simple Markov Chain

Luckly for us `std::discrete_distribution<>` does exactly what we want. It takes a list of weights $w_0, w_1, w_2, ..., w_n$ and assigns each index i the probability $p(i) = \frac{\sum_{i=0}^{n} w_i}{w_i}$ (the probability is proportional to the weight, so we have that $\sum_{i=0}^{n} p(i) = 1$ like we would expect in a Markov Chain).

To instantiate the `discrete_distribution` ④, unlike in Listing 2, we need to first calculate the weights ③, as we don't know them in advance.

To randomly generate the next state ⑥ we just have to use the `discrete_distribution` assigned to the current state ⑤.

3.5.2 `[5200]` Software development & error detection

Our next goal is to model the software development process of a team. Each phase takes the team 4 days to complete, and, at the end of each phase the testing team tests the software, and there can be 3 outcomes:

- **no error** is introduced during the phase (we can't actually know it, let's suppose there is an all-knowing "oracle" that can tell us there aren't any errors)
- **no error detected** means that the "oracle" detected an error, but the testing team wasn't able to find it
- **error detected** means that the "oracle" detected an error, and the testing team was able to find it

If we have **no error**, we proceed to the next phase… the same happens if **no error was detected** (because the testing team sucks and didn't find any errors). If we **detect an error** we either reiterate the current phase (with a certain probability, let's suppose 0.8), or we go back to one of the previous phases with equal probability (we do this because, if we find an error, there's a high chance it was introduced in the current phase, and we want to keep the model simple).

In this exercise we take the parameters for each phase (the probability to introduce an error and the probability to not detect an error) from a file.

```
#include <...>

using real_t = double;
const size_t HORIZON = 800, PHASES_SIZE = 3;

enum Outcome (1) {
    NO_ERROR = 0,
    NO_ERROR_DETECTED = 1,
    ERROR_DETECTED = 2
};

int main() {
    std::random_device random_device;
    std::default_random_engine urng(random_device());
    std::uniform_real_distribution<> uniform_0_1(0, 1);
    std::vector<std::discrete_distribution<>>
        phases_error_distribution;

    {
        std::ifstream probabilities("probabilities.csv");
        real_t probability_error_introduced,
            probability_error_not_detected;

        while (probabilities >> probability_error_introduced >>
               probability_error_not_detected)
            phases_error_distribution.push_back(
                (2) std::discrete_distribution<>({
                    1 - probability_error_introduced,
                    probability_error_introduced *
                        probability_error_not_detected,
                    probability_error_introduced *
                        (1 - probability_error_not_detected),
                }));

        probabilities.close();
        assert(phases_error_distribution.size() ==
               PHASES_SIZE);
    }

    real_t probability_repeat_phase = 0.8;

    size_t phase = 0;
```

```cpp
    std::vector<size_t> progress(PHASES_SIZE, 0);
    std::vector<Outcome> outcomes(PHASES_SIZE, NO_ERROR);

    for (size_t time = 0; time < HORIZON; time++) {
        progress[phase]++;

        if (progress[phase] == 4) {
            outcomes[phase] = static_cast<Outcome>(
                phases_error_distribution[phase](urng));
            switch (outcomes[phase]) {
            case NO_ERROR:
            case NO_ERROR_DETECTED:
                phase++;
                break;
            case ERROR_DETECTED:
                if (phase > 0 && uniform_0_1(urng) >
                                     probability_repeat_phase)
                    phase = std::uniform_int_distribution<>(
                        0, phase - 1)(urng);
                break;
            }

            if (phase == PHASES_SIZE)
                break;

            progress[phase] = 0;
        }
    }

    return 0;
}
```

Listing 9: `software/5300/main.cpp`

TODO: `class enum` vs `enum`. We can model the outcomes as an `enum` ①... we can use the `discrete_distribution` trick to choose randomly one of the outcomes ②. The other thing we notice is that we take the probabilities to generate an error and to detect it from a file.

3.5.3 Optimizing costs for the development team `[5300]`

If we want we can manipulate the “parameters” in real life: a better experienced team has a lower probability to introduce an error, but a higher cost. What we can do is:

1. randomly generate the parameters (probability to introduce an error and to not detect it)
2. simulate the development process with the random parameters

By repeating this a bunch of times, we can find out which parameters have the best results, a.k.a generate the lowest development times (there are better techinques like simulated annealing, but this one is simple enough for us).

3.5.4 Key performance index `[5400]`

We can repeat the process in exercise `[5300]`, but this time we can assign a parameter a certain cost, and see which parameters optimize cost and time (or something like that? Idk, I should look up the code again).

3.6 Complex systems

3.6.1 Insulin pump `[6100]`

3.6.2 Buffer `[6200]`

3.6.3 Server `[6300]`

4 Exam

4.1 Development team (time & cost)

4.2 Backend load balancing

4.2.1 Env

4.2.2 Dispatcher, Server and Database

4.2.3 Response time

4.3 Heater simulation

5 MOCC library

Model CheCking

5.1 Observer Pattern

5.2 `C++` generics & virtual methods

TODO…

6 Extras

6.1 VDM (Vienna Development Method)

6.1.1 It's cool, I promise

6.1.2 VDM++ to design correct UMLs

6.2 Advanced testing techinques (in `Rust`)

TODO: cite "Rust for Rustaceans" TODO: unit tests aren't the only type of test

6.2.1 Mocking (mockall)

6.2.2 Fuzzying (cargo-fuzz)

6.2.3 Property-based Testing

6.2.4 Test Augmentation (Miri, Loom)

TODO: Valgrind

6.2.5 Performance testing

TODO: non-functional requirements

6.2.6 Playwright & UI testing?

Bibliography

[1] [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution

[2] [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution

[3] Marcin Kolny, “Scaling up the Prime Video Audio-Video Monitoring Service and Reducing Costs by 90%.” Accessed: Mar. 25, 2024. [Online]. Available: https://web.archive.org/web/20240325042615/https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90#expand

[4] “Single-precision floating-point format.” [Online]. Available: https://en.wikipedia.org/wiki/Single-precision_floating-point_format

[5] Wikipedia, “Euler method.” [Online]. Available: https://en.wikipedia.org/wiki/Euler_method