Software Engineering

Cicio Ionuț

21/01/2025

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.

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%**"* [1] by using a monolith architecture.

There isn't a definitive way to answer these type of questions, but one way to go about it 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:"* [1]

- 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

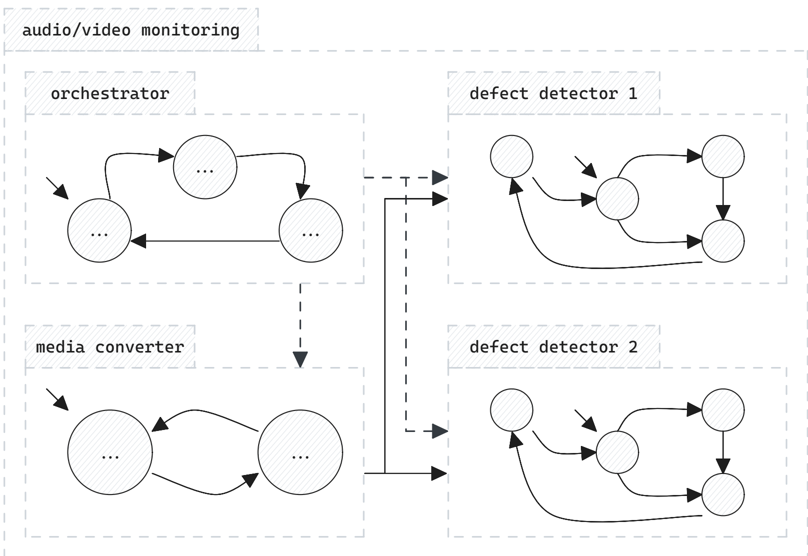


Figure 1: audio/video monitoring system

To derive conclusions the system can be **simulated** by modeling its components as **Markov decision processes**.

1.2 Formal theory

1.2.1 Markov chain

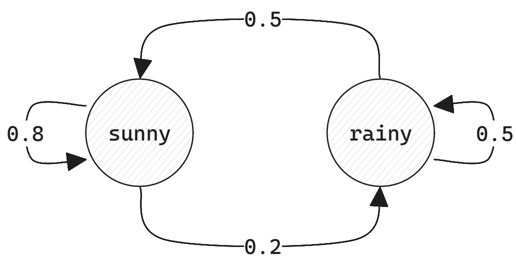
A Markov chain M is described by a set of **states** S and the **transition probability** $p : S \times S \to [0, 1]$ such that $p(s'|s)$ is the probability to transition to state s' if the current state is s. The transition probability p is constrained by Equation 1

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

For example, the weather can be modeled with $S = \{\text{sunny}, \text{rainy}\}$ and p such that

$p =$

	sunny	rainy
sunny	0.8	0.2
rainy	0.5	0.5



If a Markov chain M transitions at discrete time steps, i.e. the time steps $t_0, t_1, t_2, ...$ are a **countable**, then it's called a DTMC (discrete-time Markov chain), otherwise it's called a CTMC (continuous-time Markov chain).

1.2.2 Markov decision process

A Markov decision process (MDP), despite sharing the name, is **different** from a Markov chain, because transitions are influenced by an external environment. A MDP M is a tuple (U, X, Y, p, g) s.t.

- 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 such that $p(x'|x, u)$ is the probability to **transition** from state x to state x' when the **input value** is u
- $g : X \to Y$ is the **output function**
- and let $x_0 \in X$ be the **initial state**

The same constrain in Equation 1 holds for MDPs, with an important difference: **for each input value**, the sum of the transition probabilities for **that input value** must be 1.

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

1.2.3 Example

The development process of a company can be modeled as a MDP $M = (U, X, Y, p, g)$ s.t.

- $U = \{\varepsilon\}$[1]
- $X = \{0, 1, 2, 3, 4\}$
- $Y = \text{Cost} \times \text{Duration}$
- $x_0 = 0$

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

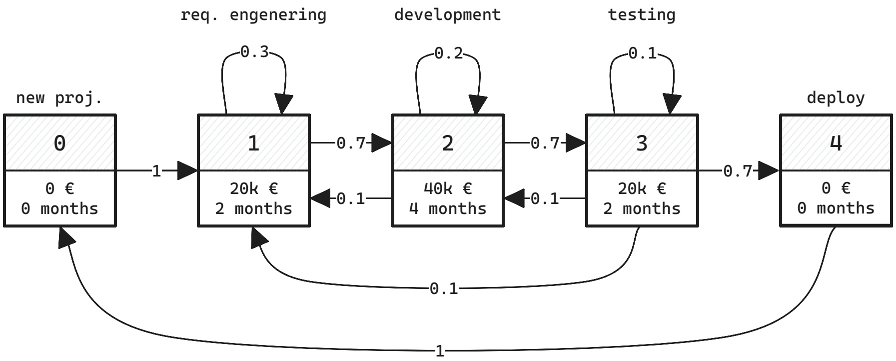


Figure 2: the model of a team's development process

$p =$

ε	**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

Only **1 transition matrix** is defined, as $|U| = 1$ (there's 1 input value). If U had multiple input values, like $\{\text{apple}, \text{banana}, \text{orange}\}$, then 3 transition matrices would have been required, one **for each input value**.

[1]If U is empty M can't transition, at least 1 input is required, i.e. ε

1.3 Tips and tricks

1.3.1 Average

Given a set of values $X = \{x_1, ..., x_n\} \subset \mathbb{R}$ the average $\overline{x}_n = \frac{\sum_{i=0}^{n} x_i}{n}$ can be computed with a simple procedure

```
float average(std::vector<float> X) {
    float sum = 0;
    for (auto x_i : X)
        sum += x_i;

    return sum / X.size();
}
```

The problem with this procedure is that, by adding up all the values before the division, the **numerator** could **overflow**, even if the value of $\overline{x}_n$ fits within the IEEE-754 limits. Nonetheless, $\overline{x}_n$ can be calculated incrementally.

$$\begin{gathered}\overline{x}_{n+1} = \frac{\sum_{i=0}^{n+1} x_i}{n+1} = \frac{\left(\sum_{i=0}^{n} x_i\right) + x_{n+1}}{n+1} = \frac{\sum_{i=0}^{n} x_i}{n+1} + \frac{x_{n+1}}{n+1} = \\ \frac{\left(\sum_{i=0}^{n} x_i\right) n}{(n+1)n} + \frac{x_{n+1}}{n+1} = \frac{\sum_{i=0}^{n} x_i}{n} \cdot \frac{n}{n+1} + \frac{x_{n+1}}{n+1} = \\ \overline{x}_n \cdot \frac{n}{n+1} + \frac{x_{n+1}}{n+1}\end{gathered} \tag{4}$$

With this formula the numbers added up are smaller: $\overline{x}_n$ is multiplied by $\frac{n}{n+1} \sim 1$, and, if x_{n+1} fits in IEEE-754, then $\frac{x_{n+1}}{n+1}$ can also be encoded.

```
float incr_average(std::vector<float> X) {
    float average = 0;
    for (size_t n = 0; n < X.size(); n++)
        average =
            average * ((float)n / (n + 1)) + X[n] / (n + 1);

    return average;
}
```

In `examples/average.cpp` the procedure `average()` returns `Inf` and `incr_average()` successfuly computes the average.

1.3.2 Welford's online algorithm (standard deviation)

In a similar fashion, it could be faster and require less memory to calculate the **standard deviation** incrementally. Welford's online algorithm can be used for this purpose.

$$
\begin{aligned}
M_{2,n} &= \sum_{i=1}^{n} (x_i - \overline{x}_n)^2 \\
M_{2,n} &= M_{2,n-1} + (x_n - \overline{x}_{n-1})(x_N - \overline{x}_n) \\
\sigma_n^2 &= \frac{M_{2,n}}{n} \\
s_n^2 &= \frac{M_{2,n}}{n-1}
\end{aligned}
\tag{5}
$$

Given M_2, the standard deviation can be calculated as $\sqrt{\frac{M_{2,n}}{n}}$ if $n > 0$.

```
real_t Stat::stddev_welford() const {
    return sqrt(n > 0 ? m_2__ / n : 0);
}
```

Listing 1: `mocc/stat.hpp`

1.3.3 Euler method for ordinary differential equations

When an ordinary differential equation can't be solved analitically, the solution must be approximated. There are many techniques: one of the simplest ones (yet less accurate and efficient) is the forward Euler method, described by the following equation:

$$
y_{n+1} = y_n + \Delta \cdot f(x_n, y_n) \tag{6}
$$

Let the function y be the solution to the following problem

$$
\begin{cases} y(x_0) = y_0 \\ y'(x) = f(x, y(x)) \end{cases} \tag{7}
$$

Let $y(x_0) = y_0$ be the initial condition of the system, and $y' = f(x, y(x))$ be the known **derivative** of y (y' is a function of x and $y(x)$). To approximate y, a Δ is chosen (the smaller, the more precise the approximation), then $x_{n+1} = x_n + \Delta$. Now, understanding Equation 6 should be easier: the value of y at the next **step** is the current value of y plus the value of its derivative y' (multiplied by Δ). In Equation 6 y' is multiplied by Δ because when going to the next step, all the derivatives from x_n to x_{n+1} must be added up, and it's done by adding up

$$
(x_{n+1} - x_n) \cdot f(x_n, y_n) = \Delta \cdot f(x_n, y_n) \tag{8}
$$

Where $y_n = y(x_n)$. Given this theoretical understanding, implementing the code should be simple enough.

The following program approximates $y = x^2$ with $\Delta = 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}$, knowing that $y' = 2x$.

```
#define SIZE 4
float derivative(float x) { return 2 * x; }

int main() {
    size_t x[SIZE];

    for (size_t i = 0; i < SIZE; i++) {
        x[i] = 0;
        float delta = 1. / (i + 1);
        for (float t = 0; t <= 10; t += delta)
            x[i] += delta * derivative(t);
    }

    return 0;
}
```

Listing 2: `examples/euler.cpp`

When plotting the results, it can be observed that the approximation is close, but not very precise. The error analysis in the Euler method is beyond this guide's scope.

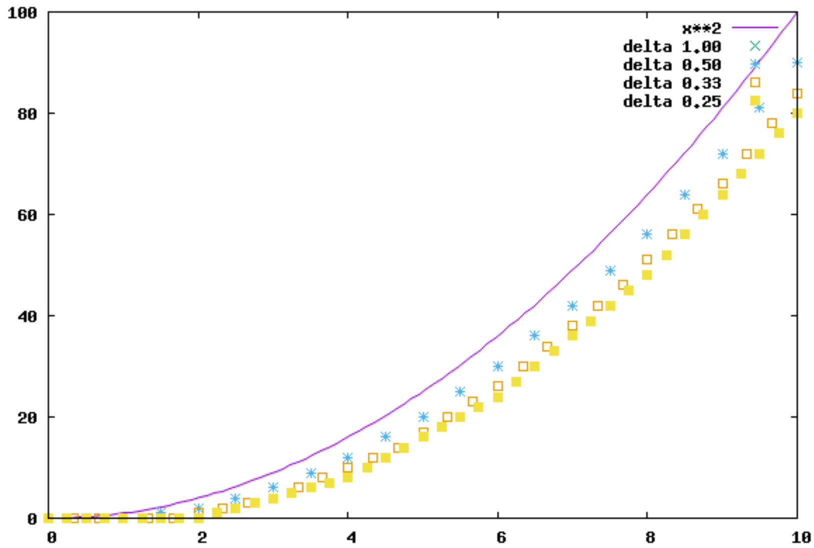


Figure 3: examples/euler.png

2 How to C++

This section covers the basics assuming the reader already knows C.

2.1 The random library

The C++ standard library offers tools to easily implement MDPs.

2.1.1 Random engines

In C++ there are many ways to **generate random numbers** [2]. Generally it's **not recommended** to use `random()` ① . It's recommended to use a **random generator** ⑤, because it's fast, deterministic (given a **seed**, the sequence of generated numbers is the same) and can be used with **distributions**. A `random_device` is a non deterministic generator: it uses a **hardware entropy source** (if available) to generate the random numbers.

```
#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;
    int seed = random_device(); ④
    std::default_random_engine r_engine(seed); ⑤
    std::cout << r_engine() ⑥ << std::endl;
}
```

Listing 3: `examples/random.cpp`

The typical course of action is to instantiate a `random_device` ②, and use it to generate a seed ④ for a `random_engine`. Given that random engines can be used with distributions, they're really useful to implement MDPs.

From this point on, `std::default_random_engine` will be reffered to as `urng_t` (uniform random number generator type).

```
#include <random>
// works like typedef in C
using urng_t = std::default_random_engine;

int main() {
    std::random_device random_device;
    urng_t urng(random_device());
}
```

2.1.2 Operator overloading *(quick note)*

In Listing 3, to generate a random number, `random_device()` ③ and `r_engine()` ⑥ are used like functions, but they aren't functions, they're instances of a `class`. That's because in `C++` you can define how a certain operator (like `+`, `+=`, `<<`, `>>`, `[]`, `()` etc..) should behave when used on a instance of the `class`. It's called **operator overloading**, a relatively common feature:

- in `Python` operation overloading is done by implementing methods with special names, like `__add__()` [3]
- in `Rust` it's done by implementing the `Trait` associated with the operation, like `std::ops::Add` [4].
- `Java` and `C` don't have operator overloading

For example, `std::cout` is an instance of the `std::basic_ostream` `class`, which overloads the method "`operator<<()`" [5].

2.1.3 Distributions

Just the capability to generate random numbers isn't enough, these numbers need to be manipulated to fit certain needs. Luckly, `C++` covers **basically all of them**. For example, the MDP in Figure 2 can be easily simulated with the following code code:

```cpp
#include <iostream>
#include <random>
using urng_t = std::default_random_engine;

int main() {
    std::random_device random_device;
    urng_t urng(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](urng);
        std::cout << state << std::endl;
    }

    return 0;
}
```

Listing 4: `examples/transition_matrix.cpp`

2.1.3.1 Uniform discrete [6]

Let's consider a simple exercise

> To test a system S it's requried to build a generator that sends value v_t to S every T_t seconds. For each send, the value of T_t is an **integer** chosen uniformly in the range $[20, 30]$.

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

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

The interval T_t can be easily generated ② without needing to remember any formula or trick. The behaviour of T_t is defined only once ①, so it can be easily changed without introducing bugs or inconsistencies. It's also worth to take a look at the implementation of the exercise above (with the addition that $v_t = T_t$), as it comes up very often in software models.

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

using urng_t = std::default_random_engine;

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

    size_t T = random_T(urng), 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(urng);
        next_request_time = time + T;
    }

    return 0;
}
```

Listing 5: `examples/interval_generator.cpp`

The `uniform_int_distribution` has many other uses, for example, it could uniformly generate a random state in a MDP. Let `STATES_SIZE` be the number of states

```
uniform_int_distribution<> random_state(0, STATES_SIZE - 1 ①);
```

`random_state` generates a random state when used. Be careful! Remember to use `STATES_SIZE - 1` ①, because `uniform_int_distribution` is inclusive. Forgettig `-1` can lead to very sneaky bugs, like random segfaults at different instructions. It's very hard to debug unless using `gdb`. The `uniform_int_distribution` can also generate negative integers, for example $z \in \{x \mid x \in \mathbb{Z} \wedge x \in [-10, 15]\}$.

2.1.3.2 Uniform continuous [7]

It's the same as above, with the difference that it generates **real** numbers in the range $[a, b) \subset \mathbb{R}$.

2.1.3.3 Bernoulli [8]

Let's consider the following exercise

> To model a network protocol P it's required to model a request. The request can randomly fail with probability $p = 0.001$.

Traditionally, a random real numer $r \in [0, 1]$ is generated, and it's checked if r is above or below p.

```
std::uniform_real_distribution<> random_r(0, 1);
r = random_r(urng);
if (r > 0.001)
  fail();
```

The `std::bernoulli_distribution` works better for this specification

```
std::bernoulli_distribution random_fail(0.001);
if (random_fail(urng))
  fail();
```

2.1.3.4 Normal

2.1.3.5 Exponential

The Exponential distribution is very useful when simulating user requests (generally, the interval between requests to a servers is described by a Exponential distribution, you just have to specify λ)

2.1.3.6 Poisson

2.1.3.7 Geometric

2.1.3.8 Discrete distribution

Let's consider the following exercise.

> To choose the architecture for an e-commerce it's required to implement a model C of the customers that simulates the requests. After interviewing 678 people it's determined that 232 of them would buy a hat, 158 would buy a hoodie and the other 288 would buy a mug.

A discrete distribution can be used for this case. Let's say that "hat" = 0, "hoodie" = 1 and mug = 2.

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

int main() {
  std::random_device random_device;
  urng_t urng(random_device());
  std::discrete_distribution<> random_item = {232, 158, 288};

  for (int request = 0; request < 1000; request++) {
    size_t item = random_item(urng);
    switch (item) {
      case 0:
        std::cout << "hat" << std::endl;
        break;
      case 1:
        std::cout << "hoodie" << std::endl;
        break;
      case 2:
        std::cout << "mug" << std::endl;
        break;
    }
  }

  return 0;
}
```

With the discrete distribution, the generated items are proportional to the data.

2.2 Dynamic structures

2.2.1 Manual memory allocation *(and how to avoid it)*

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

To avoid manual memory allocation, most of the time it's enough to use the structures in the standard library, like `std::vector<T>`.

2.2.2 `std::vector<T>()`

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 Standard I/O

2.3.2 Files

Working with files is way to easy in `C++`

```
#include <ofstream>
#include <ifstream>

int main(){
    ofstream output("output.txt");
    output << "some text" << std::endl;
    output.close();

    ifstream inputs("inputs.txt");
    int number;
    while (inputs >> number) {
        // do stuff with number...
    }
    inputs.close();

    return 0;
}
```

3 Debugging with gdb

It's super useful! Trust me, if you learn this everything is way easier

First of all, use the `-ggdb` flags to compile the code. Remember to not use any optimization like `-O3`… using optimizations makes the program harder to debug.

```
DEBUG_FLAGS := -lm -std=c++11 -ggdb3 -Wall -Wextra -pedantic
```

Then it's as easy as running `gdb ./main`

- TODO: could be useful to write a script if too many args
- TODO: just bash code to compile and run
- TODO (just the most useful stuff, thechnically not enough):
  - r
  - c
  - n
  - c 10
  - enter (last instruction)
  - b
    - on lines
    - on symbols
    - on specific files
  - clear
  - display
  - set print pretty on

4 Examples

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

4.1 First examples

This section puts together the **formal definitions** and the `C++` knowledge to implement some simple MDPs.

4.1.1 A simple Markov decision process [1100]

The first MDP $M = (U, X, Y, p, g)$ is such that

- $U = \{\varepsilon\}$ (see Section 1.2.3)
- $X = [0, 1] \times [0, 1]$, each state is a pair ③ of **real** numbers 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 continuous** distribution ②
- $g : X \to Y : (r_0, r_1) \mapsto (r_0, r_1)$ outputs the current state ④
- $x_0 = (0, 0)$ is the initial state ③

```
#include <random>

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

int main() {
    std::random_device random_device;
    urng_t urng(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(urng); ②
        log << time << ' ';
        for (auto r : state) log << r << ' '; t ④
        log << std::endl;
    }

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

Listing 6: `software/1100/main.cpp`

4.1.2 Markov decision processes network pt.1 [1200]

This example has 2 MDPs M_0, M_1 like the one in the first example Section 4.1.1, 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

$$\begin{gathered} p : X \times X \times U \to [0,1] \\ (x_0, x_1), (x'_0, x'_1), (u_0, u_1) \mapsto \ldots \end{gathered} \tag{9}$$

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

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

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

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

Listing 7: `software/1200/main.cpp`

4.1.3 Markov decision processes network pt.2 [1300]

The same as above, but with a different connection

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

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

        mdps[1].state[0] =
            mdps[1].state[0] + mdps[1].state[1];
        mdps[1].state[1] =
            -mdps[1].state[0] + mdps[1].state[1];
    }
}
```

Listing 8: `software/1300/main.cpp`

4.1.4 Markov decision processes 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)

4.2 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.

```cpp
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 9: `software/2000/main.cpp`

4.3 Control center

4.3.1 No network `[3100]`

4.3.2 Network monitor

4.3.2.1 No faults `[3200]`

4.3.2.2 Faults & no repair `[3300]`

4.3.3 Faults & repair `[3400]`

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

4.4 Statistics

4.4.1 Expected value `[4100]`

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

4.4.2 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?)

4.5 Development process simulation

An MDP can be implemented by using a **transition matrix** (like in Section 1.2.3). The simplest implemenation can be done by using a `std::discrete_distribution` by using the trick in Listing 4.

4.5.1 Random transition matrix [5100]

This example builds a **random transition matrix**.

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

int main() {
    std::random_device random_device;
    urng_t urng(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(urng);

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

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

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

Listing 10: `software/5100/main.cpp`

A **transition matrix** is a `vector<discrete_distribution<>>` ② just like in Listing 4. 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 4.

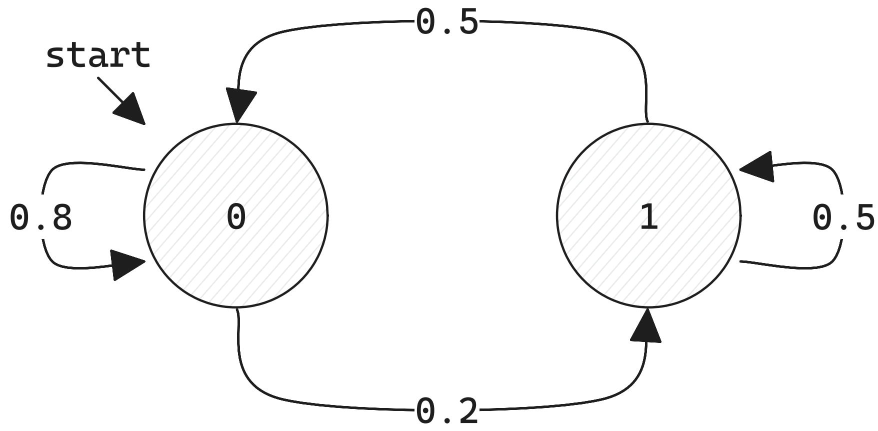


Figure 4: A simple Markov Chain

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 4, 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.

4.5.2 `[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;
```

```
    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 11: `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.

4.5.3 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 techniques like simulated annealing, but this one is simple enough for us).

4.5.4 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).

4.6 Complex systems

4.6.1 Insulin pump [6100]

4.6.2 Buffer [6200]

4.6.3 Server [6300]

5 Exam

5.1 Development team (time & cost)

5.2 Backend load balancing

5.2.1 Env

5.2.2 Dispatcher, Server and Database

5.2.3 Response time

5.3 Heater simulation

5.4 Task management

6 MOCC library

Model CheCking

6.1 Observer Pattern

Basically: the “Observer Pattern” [9] can be used because a MDP is like an entity that “is notified” when something happens (receives an input, in fact, in the case of MDPs, another name for input is “action”), and notifies other entities (output, or reward)

6.2 `C++` generics & virtual methods

Generics allow to connect MDPs more safely, as the inputs and outputs are typed! (It’s still not fault-proof)

7 Extras

7.1 VDM (Vienna Development Method)

7.1.1 It's cool, I promise

- Alloy? Maybe it's a good alternative, haven't tried it enough

7.1.2 VDM++ to design valid UMLs

7.2 Advanced testing techinques (in `Rust` & `C`)

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

7.2.1 Mocking (mockall)

7.2.2 Fuzzying (cargo-fuzz)

7.2.3 Property-based testing

7.2.4 Test augmentation (Miri, Loom, Valgrind)

7.2.5 Performance testing

- Rust is very focused on performance
- TODO: non-functional requirements

7.3 UI testing?

7.3.1 Playwright

7.4 Model checking with Bevy (`Rust`)

Bibliography

[1] 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

[2] "Pseudo-random number generation." [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random

[3] "operator — Standard operators as functions." [Online]. Available: https://docs.python.org/3/library/operator.html

[4] "Module ops." [Online]. Available: https://doc.rust-lang.org/std/ops/index.html#examples

[5] "std::basic_ostream." [Online]. Available: https://en.cppreference.com/w/cpp/io/basic_ostream/operator_ltlt

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

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

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

[9] "Observer Pattern." [Online]. Available: https://refactoring.guru/design-patterns/observer